Учебник по языку программирования D

Оригинал учебника расположен на сайте tutorialspoint.com, автор не указан.

Перевод: Striver

Оглавление

D – язык объектно-ориентированного мультипарадигменного системного программирования. Фактически D разрабатывался путем реинжиниринга языка программирования C++, но это отличный язык программирования, который не только использует некоторые возможности C++, но и взял кое-что из других языков программирования, таких как Java, C#, Python и Ruby. В этом учебном пособии рассматриваются различные темы, начиная от основ языка D и заканчивая передовыми ООП-концепциями наряду с дополнительными примерами.

Читатели

Этот учебник предназначен для всех тех людей, которые ищут отправную точку, откуда можно начать изучать язык D. Как начинающие, так и продвинутые пользователи могут обращаться к этому учебнику как к материалу для изучения. Увлечённые ученики могут обращаться сюда на-ходу как к справочному материалу. Человек с логическим мышлением может получать удовольствие, обучаясь D с помощью этого учебника.

Что вы должны уметь

Прежде чем приступать к изучению этого учебника, вам желательно иметь общие представления о компьютерном программировании. Вы должны уметь пользоваться простым текстовым редактором и командной строкой.

Выполнение программ на D на-лету

Для большинства примеров, приведенных в этом учебнике, вы найдете кнопку «Попробовать», поэтому просто используйте эту возможность для выполнения программ на языке D прямо здесь и получайте удовольствие от обучения.

Попробуйте выполнить следующий пример, используя ссылку «Попробовать», расположенную в правом верхнем углу нижеприведенной области с кодом:

Такая кнопка «Попробовать» есть в примерах на сайте tutorialspoint.com и в веб-версии моего перевода на сайте striver00.ru. Здесь, в PDF-версии, такой возможности, конечно же, нет.

Также аналогичная возможность исполнения D-программ на-лету есть на сайте dpaste.dzfl.pl и прямо на титульной странице официального сайта языка D dlang.org, причём с последней версией компилятора DMD. – прим. пер.

```
import std.stdio;

void main(string[] args) {
   writeln("Hello World!");
}
```

Обзор языка D

D – язык объектно-ориентированного мультипарадигменного системного программирования, разработанный Уолтером Брайтом из Digital Mars. Его разработка началась в 1999 году и была впервые выпущена в 2001 году. Первая мажорная версия D (1.0) была выпущена в 2007 году. В настоящее время у нас есть версия D2 языка D.

D – язык с C-подобным синтаксисом, который использует статическую типизацию. В D присутсвует множество особенностей C и C++, но, тем не менее, некоторые возможности этих языков не входят в D. Вот некоторые из наиболее заметных дополнений D, отсутствующих в C/C++:

- Unit-тестирование
- True modules
- Сборка мусора
- First class arrays
- Свободный и открытый
- Ассоциативные массивы
- Динамические массивы
- Внутренние классы
- Замыкания
- Анонимные функции
- Ленивые вычисления

Несколько парадигм

D – мультипарадигменный язык программирования. В D возможно использовать следующие парадигмы:

- Императивная
- Объектно-ориентированная
- Мета-программирование
- Функциональная
- Параллельная

Пример

```
import std.stdio;

void main(string[] args) {
   writeln("Hello World!");
}
```

Изучение D

Самое важное при изучении D – это сосредоточиться на идеях и не потеряться в технических деталях языка.

Цель изучения языка программирования – стать лучшим программистом; то есть стать более эффективным при проектировании и внедрении новых систем и при поддержке старых.

Область применения D

Программирование на D имеет некоторые интересные особенности, а официальный сайт языка D утверждает, что D является удобным, мощным и эффективным. В ядро языка D добавлено множество возможностей, которые язык C предоставляет в виде стандартных библиотек, такие как массив с изменяемым размером или строки. D является отличным вторым языком для программистов среднего и продвинутого уровня. В D лучше подход к использованию памяти и управлению указателями, которые часто вызывают проблемы на C++.

Язык D предназначен в основном для новых программ, и преобразования существующих программ. Он предоставляет встроенное тестирование и верификацию кода, что идеально для нового крупного проекта, в котором большие команды напишут миллионы строк кода.

Окружение

Онлайн-опция "Попробовать"

Вам действительно не нужно настраивать какое-либо окружение, чтобы начать изучение языка программирования D. Причина очень проста, мы уже создали окружение языка D онлайн под кнопкой «Попробовать». Используя эту возможность, вы можете писать и выполнять все приведённые примеры онлайн, одновременно с изучением теории. Это придаст вам уверенности проверять всё, что вы прочитали и получать результат при различных вариантах входных данных. Не стесняйтесь изменять любой пример и выполнять его через Интернет.

Попробуйте выполнить следующий пример, используя ссылку **«Попробовать»**, расположенную в правом верхнем углу нижеприведенной области с кодом:

```
import std.stdio;

void main(string[] args) {
   writeln("Hello World!");
}
```

Для большинства примеров, приведенных в этом учебнике, вы найдете кнопку **«Попробовать»**, поэтому просто используйте её и наслаждайтесь обучением.

Как я уже писал, это замечание верно только для веб-версии учебника — прим.пер.

Локальная настройка окружения для D

Если вы всё-таки готовы настроить своё окружение для языка программирования D, вам понадобятся следующие две программы, доступные на вашем компьютере:

(a) Текстовый редактор,
(b) Компилятор D.

Текстовый редактор для программирования на D

Он будет использоваться для ввода вашей программы. Вот примеры нескольких редакторов: Блокнот Windows, команда ОС Edit, Brief, Epsilon, EMACS и vim или vi.

Экзотичненький списочек, про Brief и Epsilon я вообще раньше не слышал... В Windows я использую редактор Notepad++, подсветку D-синтаксиса он поддерживает. Под Linux существуют тысячи текстовых редакторов, и, скорее всего, любой из них подойдёт. Например те, что в конкретном дистрибутиве используются по-умолчанию (gedit, kate и т.д.) – прим. пер.

Имя и версия текстового редактора могут различаться в разных операционных системах. Например, Notepad будет использоваться в Windows, а vim или vi можно использовать как в Windows, так и в Linux или UNIX.

Файлы, созданные с помощью вашего редактора, называются исходными файлами и содержат исходный код программы. Исходные файлы для программ на D именуются с расширением "**.d**".

Перед началом программирования убедитесь, что у вас уже есть один текстовый редактор, и у вас достаточно опыта для написания компьютерной программы, сохранения её в файле, её построения и, наконец, выполнения.

Компилятор D

Большинство современных реализаций D скомпилируются непосредственно в машинный код для эффективного выполнения.

Нам доступно несколько D-компиляторов, вот их список:

- **DMD** − Компилятор Digital Mars D является официальным компилятором D от Walter Bright.
- **GDC** − Front-end для фонового компилятора GCC, построенный с использованием открытого исходного кода компилятора DMD.
- **LDC** − Компилятор на основе front-end DMD, который использует LLVM в качестве фонового компилятора.

Все вышеупомянутые различные компиляторы можно загрузить со страницы загрузки сайта языка D.

Мы будем использовать вторую версию D, и мы не рекомендуем загружать D1.

Давайте воспользуемся следующей программой helloWorld.d. Мы будем использовать её как первую программу, которую мы запустим на выбранной вами платформе.

```
import std.stdio;

void main(string[] args) {
   writeln("Hello World!");
}
```

Инсталляция D в Windows

Загрузите инсталлятор для Windows.

Эта ссылка ведет к версии 2.064, видимо это была последняя версия на момент написания учебника. Лучше скачайте последний инсталлятор со страницы загрузки, упомянутой в предыдущем подразделе. Тоже самое касается ссылок для различных дистрибутивов Linux и для Mac OS X, приведённых в следующих подразделах на этой странице – прим. пер.

Запустите загруженный исполняемый файл для установки D, которую можно выполнить, следуя инструкциям на экране.

Теперь мы можем создать и запустить d-файл, скажем helloWorld.d, переключившись на каталог, содержащий файл с помощью cd, а затем используя следующие шаги -

```
C:\DProgramming> DMD helloWorld.d
C:\DProgramming> helloWorld
```

Мы можем увидеть следующий вывод.

```
hello world
```

C:\DProgramming – это каталог, который я использую для сохранения моих примеров. Вы можете изменить его на каталог, в который вы будете сохранять D-программы.

Замечание для русскоязычных пользователей Windows. Примеры выводят результаты своей работы на системную консоль. В русскоязычной Windows кодировка такой консоли называется cp866, и она не поддерживается напрямую в языке D, весь русскоязычный текст там будет отображаться неверно. Вы можете либо выполнять примеры учебника в браузере через кнопку "Попробовать", либо исправить кодировку этой консоли. Для исправления кодировки выполните у консоли следующие действия:

1) Вызовите свойства консоли (кликните по иконке окна слева в заголовке, в появившемся меню выберите "Свойства")
*2) Откройте вкладку "Шрифт", там выберите шрифт "**Lucida Console**", нажмите OK*
*3) В самой консоли наберите команду перехода на кодировку Utf8: **chcp 65001***

После этих действий консоль будет нормально отображать юникодные русские буквы, в том числе из программ на D. – прим. пер.

Инсталляция D в Ubuntu/Debian

Загрузите инсталлятор для debian.

Запустите установку загруженного пакета для установки D, которую можно выполнить, следуя инструкциям на экране.

Теперь мы можем создать и запустить d-файл, скажем helloWorld.d, переключившись на каталог, содержащий файл с помощью cd, а затем используя следующие шаги -

```
$ dmd helloWorld.d
$ ./helloWorld
```

Мы можем увидеть следующий вывод.

```
$ hello world
```

Инсталляция D в Mac OS X

Загрузите инсталлятор для Mac.

Запустите установку загруженного пакета для установки D, которую можно выполнить, следуя инструкциям на экране.

Теперь мы можем создать и запустить d-файл, скажем helloWorld.d, переключившись на каталог, содержащий файл с помощью cd, а затем используя следующие шаги -

```
$ dmd helloWorld.d
$ ./helloWorld
```

Мы можем увидеть следующий вывод.

```
$ hello world
```

Инсталляция D в Fedora

Загрузите инсталлятор для fedora.

Запустите установку загруженного пакета для установки D, которую можно выполнить, следуя инструкциям на экране.

Теперь мы можем создать и запустить d-файл, скажем helloWorld.d, переключившись на каталог, содержащий файл с помощью cd, а затем используя следующие шаги -

```
$ dmd helloWorld.d
$ ./helloWorld
```

Мы можем увидеть следующий вывод.

```
$ hello world
```

Интегрированная среда разработки (IDE) для D

Поддержка языка D в виде плагинов присутствует в большинстве существующих IDE. В том числе:

- Visual D plugin – это плагин для Visual Studio 2008-15
- DDT – это плагин для eclipse, который предоставляет автодополнение кода, отладку с помощью GDB.
- Mono-D поддерживает автодополнение кода, отладку с поддержкой dmd/ldc/gdc. Это часть GSoC 2012.
- Code_Blocks – это многоплатформенная среда разработки, которая поддерживает создание, подсветку и отладку проектов на D.

От себя упомяну ещё про Dlang IDE – IDE, написанная на самом D — прим. пер.

Базовый синтаксис

D достаточно прост в освоении, так что давайте начнём создавать нашу первую D-программу!

Первая D-программа

Напишем простую D-программу. Все D-файлы будут иметь расширение .d. Поэтому вставьте следующий исходный код в файл test.d.

```
import std.stdio;

/* Моя первая программа на D */
void main(string[] args) {
   writeln("тест!");
}
```

Предполагая, что окружение для языка D настроено правильно, запустите программу, используя

```
$ dmd test.d
$ ./test
```

*Для быстрого запуска можно использовать команду **rdmd**, при этом программа сразу будет откомпилирована и запущена*

```
$ rdmd test.d
```

– прим. пер.

Мы можем увидеть следующий вывод.

```
тест!
```

Давайте теперь рассмотрим базовую структуру программы на D, чтобы вам было легче понять основные строительные блоки языка D.

Импорт в D

Библиотеки являются коллекциями многократно используемых частей программы, доступ к ним для нашего проекта можно получить с помощью оператора импорта import. Здесь мы импортируем стандартную библиотеку, которая предоставляет базовые операции ввода-вывода. Функция writeln, которая используется в программе, показанной выше, является функцией из стандартной библиотеки D. Она используется для печати строки текста. Содержимое библиотек в D сгруппировано в модули, в соответсвии с типами задач, для которых их предполагается использовать. Единственным модулем, который применяет эта программа, является std.stdio, предназначенный для выполнения ввода и вывода данных.

Функция main

Функция main – это то место, в котором начинается выполнение программы, и в котором определяется порядок и способы выполнения других частей программы.

Токены в D

Программа на D состоит из различных токенов, а токен может являться ключевым словом, идентификатором, константой, строковым литералом, либо символом. Например, следующий D-оператор состоит из четырех токенов –

```
writeln("тест!");
```

Отдельными токенами являются:

```
writeln (
"тест!"
)
;
```

Не нравится мне слово "Токен", но боюсь, что не могу придумать для английского слова "token" адекватного ситуации перевода. Возможно, подошло бы слово "Литерал", но по факту оно такое же кривое... – прим. пер.

Комментарии

Комментарии – это что-то вроде вспомогательного текста в вашей программе на D, и они игнорируются компилятором. Многострочный комментарий начинается с /* и заканчивается символами */, как показано ниже –

```
/* Моя первая программа на D */
```

Одиночный комментарий записывается с использованием символов // в начале комментария.

```
// моя первая программа на D
```

Идентификаторы

Идентификатор в D − это имя, используемое для идентификации переменной, функции или любого другого пользовательского элемента. Идентификатор начинается с буквы от A до Z или от a до z или с символа подчеркивания _, за которым следуют ноль или более букв, знаков подчеркивания и цифр (от 0 до 9).

D не допускает знаков, таких как @, $ и % внутри идентификаторов. D − чувствительный к регистру язык программирования. Таким образом, *Manpower* и *manpower* являются двумя разными идентификаторами в D. Вот некоторые примеры допустимых идентификаторов −

```
mohd       zara    abc   move_name  a_123
myname50   _temp   j     a23b9      retVal
```

Ключевые слова

В следующем списке показаны несколько зарезервированных слов в D. Эти зарезервированные слова нельзя использовать как константы или переменные, или в качестве имён любых других идентификаторов.

abstract	alias	align	asm
assert	auto	body	bool
byte	case	cast	catch

char	class	const	continue
dchar	debug	default	delegate
deprecated	do	double	else
enum	export	extern	false
final	finally	float	for
foreach	function	goto	if
import	in	inout	int
interface	invariant	is	long
macro	mixin	module	new
null	out	override	package
pragma	private	protected	public
real	ref	return	scope
short	static	struct	super
switch	synchronized	template	this
throw	true	try	typeid
typeof	ubyte	uint	ulong
union	unittest	ushort	version
void	wchar	while	with

Пробельные символы в D

Строка, содержащая только пробельные символы, возможно с комментарием, называется пустой строкой, и компилятор D полностью игнорирует её.

Пробельный символ (whitespace) − это термин, используемый в D для описания пробелов, символов табуляции, символов новой строки и комментариев. Пробельный символ отделяет одну часть инструкции от другой и позволяет интерпретатору идентифицировать, где начинается один элемент в инструкции, такой как int, и следующий элемент. Таким образом, в следующем утверждении −

```
local age
```

должен быть хотя бы один пробельный символ (обычно пробел) между local и age, чтобы интерпретатор мог различать их. С другой стороны, в следующем утверждении

```
int fruit = apples + oranges   //получить общее количество фруктов
```

Никаких пробельных символов не требуется между fruit и =, или между = и apples, хотя вы можете включать их, если хотите, для удобства чтения.

Переменные

Переменная – это не что иное, как название области хранения, которой могут манипулировать наши программы. Каждая переменная в D имеет определенный тип, который определяет размер и способ размещения памяти переменной; диапазон значений, которые можно запоминать в этой памяти; и набор операций, которые можно применить к переменной.

Имя переменной может состоять из букв, цифр и символа подчеркивания. Оно должно начинаться с буквы или подчеркивания. Прописные и строчные буквы различаются, потому что язык D чувствителен к регистру. Вот основные типы переменных:

Номер.	Тип и описание
1	**char** Обычно один октет (один байт). Это целочисленный тип.
2	**int** Наиболее естественный размер целого числа для машины.
3	**float** Значение с плавающей запятой одинарной точности.
4	**double** Значение с плавающей запятой двойной точности.
5	**void** Представляет отсутствие типа.

D также позволяет определять различные другие типы переменных, такие как Перечисление, Указатель, Массив, Структура, Объединение и т. д., которые мы рассмотрим в последующих главах. В этой главе изучим только основные типы переменных.

Объявление переменной в D

Объявление переменной сообщает компилятору, где и сколько пространства выделить для создания переменной. Объявление переменной состоит из типа данных и списка из одной или нескольких переменных этого типа следующим образом:

```
тип список_переменных;
```

Здесь, **тип** должен быть допустимым типом данных в D, включая char, wchar, int, float, double, bool или любой определённый пользователем объект и т. д., а **список_переменных** может состоять из одного или нескольких имен идентификаторов, разделенных запятыми. Здесь показаны некоторые допустимые объявления –

```
int    i, j, k;
char   c, ch;
```

```
float  f, salary;
double d;
```

Строка **int i, j, k;** объявляет и определяет переменные i, j и k; т.е. даёт инструкцию компилятору создать переменные с именем i, j и k типа int.

Переменные можно инициализировать (задать им начальное значение) при их объявлении. Инициализатор состоит из знака равенства, за которым следует константное выражение следующим образом:

```
тип список_переменных = значение;
```

Примеры

```
extern int d = 3, f = 5;    // объявление d и f.
int d = 3, f = 5;           // объявление и инициализация d и f.
byte z = 22;                // объявление и инициализация z.
char x = 'x';               // переменная x содержит значение 'x'.
```

Когда переменная объявляется в D, ей всегда присваивается свой «инициализатор по умолчанию», к которому можно обратиться вручную, как **T.init**, где **T** – тип (например, **int.init**). Инициализатор по умолчанию для целых типов – это 0, для Booleans – false, а для чисел с плавающей запятой – NaN.

Объявление переменной дает уверенность компилятору в том, что существует только одна переменная с данным типом и именем, чтобы компилятор перешёл к дальнейшей компиляции, не требуя полной информации о переменной. Объявление переменной имеет смысл только во время компиляции, компилятору требуется действительное объявление переменной во время компоновки программы.

Пример

Попробуйте следующий пример, где переменные были объявлены в начале программы, но определены и инициализированы внутри функции main:

```
import std.stdio;

int a = 10, b = 10;
int c;
float f;

int main () {
   writeln("Значение a : ", a);

   /* переобъявление переменных: */
   int a, b;
   int c;
   float f;

   /* Инициализация */
   a = 30;
   b = 40;
   writeln("Значение a : ", a);

   c = a + b;
   writeln("Значение c : ", c);

   f = 70.0/3.0;
   writeln("Значение f : ", f);
   return 0;
```

```
}
```

Компиляция и выполнение приведённого выше кода даст следующий результат:

```
Значение a : 10
Значение a : 30
Значение c : 70
Значение f : 23.3333
```

*Добавлю от себя (почему-то в оригинале это не было указано), что вместо типа в объявлении переменной можно ставить ключевое слово **auto**, что означает "любой тип". Тем не менее, из контекста использования переменной компилятор должен быть сам способен определить тип, который нужно подставить вместо слова auto.*

```
auto a = 10; // тип a - это int
auto b = 10.6; // тип b - это double
auto c = "строка"; // тип c - это string
```

– прим. пер.

Левые и правые значения в D

В D есть два вида выражений:

- Левые значения (**lvalue**) − Выражение, которое является lvalue, может появляться как с левой, так и с правой стороны присвоения.
- Правые значения (**rvalue**) − Выражение, которое является rvalue, может появляться с правой стороны, но не может с левой стороны присвоения.

Переменные являются lvalues и поэтому могут отображаться в левой части присвоения. Числовые литералы являются rvalues и поэтому им нельзя присваивать значения, и они не могут отображаться с левой стороны. Следующее утверждение допустимо:

```
int g = 20;
```

Но следующее утверждение не является допустимым и сгенерирует ошибку времени компиляции:

```
10 = 20;
```

Типы данных

В языке программирования D типы данных относятся к обширной системе, используемой для объявления переменных или функций различных типов. Тип переменной определяет, сколько места занимает она в хранилище и как интерпретируется хранимый битовый шаблон.

Типы в D можно классифицировать следующим образом:

Номер	Типы и описания
1	**Базовые типы** Они являются арифметическими типами и состоят из трех видов: (a) целые, (b) с плавающей точкой, и (c) символы.
2	**Перечислимые типы** Это снова арифметические типы. Они используются для определения переменных, которым можно присвоить только отдельные определённые целочисленные значения во всей программе.
3	**Тип void** Описатель типа *void* указывает, что значение не доступно.
4	**Производные типы** Они включают (a) типы указателей, (b) типы массивов, (c) типы структур, (d) типы объединений и (e) типы функций.

Типы массивов и типы структур в совокупности называются агрегатными типами. Тип функции определяет тип возвращаемого значения функции. Мы увидим базовые типы в нижеследующих подразделах, в то время как остальные типы будут рассмотрены в следующих главах.

Целые типы

В следующей таблице приведён список стандартных целочисленных типов с размерами занимаемой ими памяти и диапазонами значений:

Тип	Размер	Диапазон значений
bool	1 байт	false или true
byte	1 байт	от -128 до 127
ubyte	1 байт	от 0 до 255
int	4 байта	от -2,147,483,648 до 2,147,483,647
uint	4 байта	от 0 до 4,294,967,295
short	2 байта	от -32,768 до 32,767
ushort	2 байта	от 0 до 65,535

long	8 байт	от -9223372036854775808 до 9223372036854775807
ulong	8 байт	от 0 до 18446744073709551615

Чтобы получить точный размер типа или переменной, вы можете использовать оператор **sizeof**. *type.(sizeof)* дает размер памяти, занимаемой объектом или типом в байтах. Следующий пример получает размер типа ulong на любой машине:

```
import std.stdio;

int main() {
   writeln("Размер в байтах: ", ulong.sizeof);

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Размер в байтах: 8
```

Типы с плавающей точкой

В следующей таблице приведены стандартные типы с плавающей точкой с их размерами занимаемой памяти, диапазонами значений и назначением:

Тип	Размер	Диапазон значений	Назначение
float	4 байта	от 1.17549e-38 до 3.40282e+38	6 знаков после запятой
double	8 байт	от 2.22507e-308 до 1.79769e+308	15 знаков после запятой
real	10 байт	от 3.3621e-4932 до 1.18973e+4932	Либо самый большой тип с плавающей точкой, поддерживаемый аппаратным обеспечением, либо double; В зависимости от того, что больше
ifloat	4 байта	от 1.17549e-38i до 3.40282e+38i	тип мнимого float
idouble	8 байт	от 2.22507e-308i до 1.79769e+308i	тип мнимого double
ireal	10 байт	от 3.3621e-4932 до 1.18973e+4932	тип мнимого real
cfloat	8 байт	от 1.17549e-38+1.17549e-38i до 3.40282e+38+3.40282e+38i	комплексное число, составленное из двух float
cdouble	16 байт	от 2.22507e-308+2.22507e-308i до 1.79769e+308+1.79769e+308i	комплексное число, составленное из двух double
creal	20 байт	от 3.3621e-4932+3.3621e-4932i	комплексное число, составленное из

		до 1.18973e+4932+1.18973e+4932 i	двух real

Следующий пример выводит количество памяти, занимаемое типом float:

```
import std.stdio;

int main() {
   writeln("Размер в байтах: ", float.sizeof);

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Размер в байтах: 4
```

Символьные типы

В следующей таблице перечислены стандартные типы символов с их размерами занимаемой памяти и назначением.

Тип	Размер	Назначение
char	1 байт	кодовый блок UTF-8
wchar	2 байта	кодовый блок UTF-16
dchar	4 байта	кодовый блок UTF-32 и кодовая точка Unicode

В следующем примере выводится размер памяти, занимаемый типом char.

```
import std.stdio;

int main() {
   writeln("Размер в байтах: ", char.sizeof);

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Размер в байтах: 1
```

Тип void

Тип void указывает, что значение не доступно. Он используется в двух ситуациях:

Номер	Типы и описание
1	**Функция, возвращающая void** В D возможны различные функции, которые не возвращают значение, или вы можете сказать, что они возвращают void. Функция без возвращаемого значения имеет тип возвращаемого значения void. Например, **void exit (int status);**
2	**Аргументы функции как void**

	В D возможны различные функции, которые не принимают никаких параметров. Функция без параметров может быть объявлена, как принимающая void. Например, **int rand(void);**

На данный момент тип void может быть вам не понятен, поэтому давайте продолжим, и рассмотрим эту концепцию в следующих главах.

Перечисления

Перечисление используется для определения именованных константных значений. Перечисляемый тип объявляется с использованием ключевого слова **enum**.

Синтаксис *enum*

Простейшей формой определения перечисления является следующая:

```
enum имя_перечисления {
   список_перечисления
}
```

Где,

- *имя_перечисления* станет именем перечислимого типа.
- *список_перечисления* – список идентификаторов, разделённых запятыми.

Каждый из символов в списке перечислений обозначает целочисленное значение, большее, чем предшествующий ему символ. По умолчанию значение первого символа перечисления равно 0. Например:

```
enum Days { sun, mon, tue, wed, thu, fri, sat };
```

Пример

Следующий пример демонстрирует использование переменной-перечисления:

```
import std.stdio;

enum Days { sun, mon, tue, wed, thu, fri, sat };

int main(string[] args) {
   Days day;

   day = Days.mon;
   writefln("Сегодня: %d", day);
   writefln("Пятница: %d", Days.fri);
   return 0;
}
```

Компиляция и выполнение приведённого выше кода даст следующий результат:

```
Сегодня: 1
Пятница: 5
```

В приведенной выше программе мы видим, как можно использовать перечисление. Изначально мы создаём переменную с именем *day* с типом Days – нашего объявленного перечисления. Затем мы устанавливаем его в *mon*, используя нотацию с оператором точки. Нам нужно использовать метод writefln для вывода значения mon, которое было сохранено. Вам также нужно указать тип. Он имеет тип integer, поэтому мы используем %d для печати.

Свойства именованных перечислений

В приведенном выше примере для перечисления используется имя Days и такие типы называются именованными перечислениями (named enums). Эти именованные перечисления обладают следующими свойствами:

•**init** − Этим значением инициализируется первое значение в перечислении.

•**min** − Возвращает наименьшее значение перечисления.

•**max** − Возвращает наибольшее значение перечисления.

•**sizeof** − Возвращает размер используемой памяти для перечисления.

Изменим предыдущий пример, чтобы использовать свойства.

```
import std.stdio;

// Инициализируем sun значением 1
enum Days { sun = 1, mon, tue, wed, thu, fri, sat };

int main(string[] args) {
   writefln("Min : %d", Days.min);
   writefln("Max : %d", Days.max);
   writefln("Size of: %d", Days.sizeof);
   return 0;
}
```

Компиляция и выполнение приведённого выше кода даст следующий результат:

```
Min : 1
Max : 7
Size of: 4
```

Анонимное перечисление

Перечисление без имени называется анонимным перечислением (anonymous enum). Ниже приведён пример анонимного перечисления.

```
import std.stdio;

enum { sun , mon, tue, wed, thu, fri, sat };

int main(string[] args) {
   writefln("Воскресенье : %d", sun);
   writefln("Понедельник : %d", mon);
   return 0;
}
```

Компиляция и выполнение приведённого выше кода даст следующий результат:

```
Воскресенье : 0
Понедельник : 1
```

Анонимные перечисления работают почти так же, как и именованные, но у них нет свойств max, min и sizeof.

Синтаксис перечисления с базовым типом

Ниже приведен синтаксис для перечисления с базовым типом.

```
enum :базовыйТип {
   список_перечисления
}
```

Вот примеры некоторых возможных базовых типов: long, int, string. Пример с использованием string показан ниже.

```
import std.stdio;

enum : string {
```

```
   A = "привет",
   B = "мир",
}

int main(string[] args) {
   writefln("A : %s", A);
   writefln("B : %s", B);

   return 0;
}
```

Компиляция и выполнение приведённого выше кода даст следующий результат:

```
A : привет
B : мир
```

Больше возможностей

Перечисление в D предоставляет такие возможности, как инициализация нескольких значений в перечислении различными типами. Пример показан ниже.

```
import std.stdio;

enum {
   A = 1.2f,  // A - это 1.2f типа float
   B,         // B - это 2.2f типа float
   int C = 3, // C - это 3 типа int
   D          // D - это 4 типа int
}

int main(string[] args) {
   writefln("A : %f", A);
   writefln("B : %f", B);
   writefln("C : %d", C);
   writefln("D : %d", D);
   return 0;
}
```

Компиляция и выполнение приведённого выше кода даст следующий результат:

```
A : 1.200000
B : 2.200000
C : 3
D : 4
```

От переводчика:

Почему-то авторы не написали про такой часто используемый вариант, как объявление констант. Так что напишу от себя.

Если у анонимных перечислений все элементы явно инициализуются каким-либо значением, то фигурные скобки можно опустить. Таким образом использование ключевого слова ***enum*** *становится объявлением констант, значение которых вычисляется во время компиляции:*

```
import std.stdio;

enum
   A = 1.2f,      // A - это 1.2f типа float
   B = 6.7,       // B - это 6.7 типа double
   C = 3,         // C - это 3 типа int
   D = "привет";  // D - это "привет" типа string
```

```
int main(string[] args) {
   writefln("A : %f", A);
   writefln("B : %f", B);
   writefln("C : %d", C);
   writefln("D : %s", D);
   return 0;
}
```

Компиляция и выполнение приведённого выше кода даст следующий результат:

```
A : 1.200000
B : 6.700000
C : 3
D : привет
```

Литералы

Константные значения, которые присутствуют в программе в виде части исходного кода, называются **литералами**.

Литералы могут быть любым из базовых типов данных, и их можно классифицировать как целые числа, числа с плавающей точкой, символы, строки и логические значения.

Опять же, литералы можно использовать так же, как и обычные переменные, за исключением того, что их значения нельзя изменять после их определения.

Целые литералы

Целочисленный литерал может быть одним из следующих типов:

- **Десятичный** (Decimal) использует обычное числовое представление с учетом того, что первой цифрой не может быть 0, так как эта цифра зарезервирована для указания восьмеричной системы. Это правило не распространяется на сам 0: 0 равен нулю.
- **Восьмеричный** (Octal) использует 0 как префикс числа. *В современных версиях компилятора восьмеричные литералы не поддерживаются (их можно применять с помощью шаблона std.conv.octal стандартной библиотеки). Использование числовых литералов, начинающихся с 0, вызывает ошибку компилятора – прим. пер.*
- **Двоичный** (Binary) использует 0b или 0B как префикс.
- **Шестнадцатиричный** (Hexadecimal) использует 0x или 0X как префикс.

Целочисленный литерал также может иметь суффикс, представляющий собой комбинацию букв U и L, для беззнакового и длинного, соответственно. Суффикс может быть прописным или строчным, и составляющие его буквы могут идти в любом порядке.

Когда вы не используете суффикс, компилятор сам выбирает между int, uint, long и ulong в зависимости от размера значения.

Вот несколько примеров целочисленных литералов:

```
212         // Допустимо
215u        // Допустимо
0xFeeL      // Допустимо
078         // Недопустимо: 8 не является цифрой восьмеричной системы
032UU       // Недопустимо: нельзя повторять суффикс
```

Ниже приведены другие примеры различных типов целочисленных литералов:

```
85         // десятичный int
0x4b       // шестнадцатиричный
30         // int
30u        // unsigned int (беззнаковый)
30l        // long
30ul       // unsigned long (беззнаковый)
0b001      // двоичный
```

Литералы с плавающей точкой

Литералы с плавающей точкой можно указывать либо в десятичной системе, как в 1.568, либо в шестнадцатеричной системе, как в 0x91.bc.

В десятичной системе порядок может быть представлен путем добавления символа e или E и числа после него. Например, 2.3e4 означает «2.3 умножить на 10 в степени 4». Символ «+» можно указывать до значения порядка, но он не даст эффекта. Например, 2.3e4 и 2.3e + 4 означают одно и то же.

Символ «-», добавленный до значения порядка, придаёт значение «делится на 10 в степени». Например, 2.3e-2 означает «2.3, делённое на 10 в степени 2».

В шестнадцатеричной системе значение начинается с 0x или 0X. Порядок записывается через p или P вместо e или E. Порядок означает не «10 в степени», а «2 в степени». Например, P4 в 0xabc.defP4 означает «abc.de умножить на 2 в степени 4».

Вот несколько примеров литералов с плавающей точкой:

```
3.14159       // Допустимо
314159E-5L    // Допустимо
510E          // Недопустимо: неполный порядок
210f          // Недопустимо
.e55          // Недопустимо: отсутствует целая или дробная часть
0xabc.defP4   // Допустимо: Hexa decimal with exponent
0xabc.defe4   // Допустимо: Hexa decimal without exponent.
```

По умолчанию тип литерала с плавающей точкой устанавливается в double. Суффикс f или F означает float, а спецификатор L означает real.

Логические литералы

Существует два логических (булевых) литерала, и они входят в список ключевых слов D:

- Значение **true** представляет истину.
- Значение **false** представляет ложь.

Вы не должны считать значение true равным 1, а значение false равным 0.

Литералы символов

Литералы символов заключаются в одинарные кавычки.

Литерал символа может быть простым символом (например, 'x'), escape-последовательностью (например, '\t'), символом ASCII (например, '\x21'), символом Unicode (например, '\u011e'), или в виде именованного символа (например, '\©','\♥', '\€'). *Не очень я понял про эти "именованные символы" – прим.пер.*

В D существуют определённые символы, когда им предшествует обратная косая черта, они будут иметь особый смысл, и они используются для представления таких вещей, как новая строка (\n) или табуляция (\t). Здесь приведён список некоторых из таких кодов escape-последовательностей:

escape-последовательность	Означает
\\	символ \
\'	символ '
\"	символ "
\?	символ ?
\a	Предупреждение или звонок
\b	Backspace
\f	Form feed
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция

В следующем примере показаны несколько символов escape-последовательностей:

```
import std.stdio;

int main(string[] args) {
   writefln("Привет\tМир%c\n",'\x21');
   writefln("Хорошего дня%c",'\x21');
   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Привет  Мир!

Хорошего дня!
```

Литералы строк

Строковые литералы заключаются в двойные кавычки. Строка содержит символы, похожие на литералы символов: простые символы, escape-последовательности и универсальные символы.

Вы можете разбить длинную строку на несколько строк с использованием строковых литералов и разделить их, используя пробельные символы.

В современных версиях компилятора это работает, но выдаётся предупреждение "Deprecation" (устарело) и предлагается использовать конкатенацию с помощью оператора "~" – прим.пер.

Вот несколько примеров строковых литералов:

```
import std.stdio;

void main(string[] args) {
   writeln(q"MY_DELIMITER
      Hello World
```

```
      Have a good day
MY_DELIMITER");

   writefln("Хорошего дня%c",'\x21');
   auto str = q{int value = 20; ++value;};
   writeln(str);
}
```

В приведенном выше примере вы можете найти q"MY_DELIMITER MY_DELIMITER" для представления многострочных символов. Кроме того, вы можете увидеть, что q{} представляет собой инструкцию языка D.

К сожалению, у меня не получилось простым образом использовать русские буквы внутри этих q"MY_DELIMITER MY_DELIMITER" – часто на выходе вместо буквы появлялся её код в виде '\u041f' – прим. пер.

Операторы

Оператор – это символ, который предписывает компилятору выполнить определенные математические или логические манипуляции. В языке D большое количество встроенных операторов, и их можно разделить на следующие типы:

•Арифметические операторы

•Операторы сравнения

•Логические операторы

•Побитовые операторы

•Операторы присваивания

•Другие операторы

В этой главе по очереди описываются арифметические, сравнивающие, логические, побитовые, присваивающие и другие операторы.

Арифметические операторы

В следующей таблице показаны все арифметические операторы, поддерживаемые языком D. Предположим, что в переменной **A** содержится 10, а в переменной **B** – 20, тогда:

Оператор	Описание	Пример
+	Складывает два операнда.	A + B даст 30
-	Вычитает второй операнд из первого.	A - B даст -10
*	Перемножает два операнда.	A * B даст 200
/	Делит числитель на знаменатель.	B / A даст 2
%	Возвращает остаток от целочисленного деления.	B % A даст 0
++	Оператор приращения увеличивает целое значение на единицу.	A++ даст 11
--	Оператор декремента уменьшает целое значение на единицу.	A-- даст 9

Пример

Попробуйте следующий пример, для понимания всех арифметических операторов, доступных в языке программирования D:

```
import std.stdio;

int main(string[] args) {
   int a = 21;
   int b = 10;
   int c ;
```

```
   c = a + b;
   writefln("Строка 1: Значение c равно %d", c );
   c = a - b;
   writefln("Строка 2: Значение c равно %d", c );
   c = a * b;
   writefln("Строка 3: Значение c равно %d", c );
   c = a / b;
   writefln("Строка 4: Значение c равно %d", c );
   c = a % b;
   writefln("Строка 5: Значение c равно %d", c );
   c = a++;
   writefln("Строка 6: Значение c равно %d", c );
   c = a--;
   writefln("Строка 7: Значение c равно %d", c );
   char[] buf;
   stdin.readln(buf);
   return 0;
}
```

Компиляция и выполнение приведённого выше кода даст следующий результат:

```
Строка 1: Значение c равно 31
Строка 2: Значение c равно 11
Строка 3: Значение c равно 210
Строка 4: Значение c равно 2
Строка 5: Значение c равно 1
Строка 6: Значение c равно 21
Строка 7: Значение c равно 22
```

Операторы сравнения

В следующей таблице показаны все операторы сравнения, поддерживаемые языком D. Предположим, что в переменной **A** содержится 10, а в переменной **B** – 20, тогда:

Оператор	Описание	Пример
==	Проверяет, равны ли значения двух операндов или нет, если да, то условие становится истинным.	(A == B) – false.
!=	Проверяет, равны ли значения двух операндов или нет, если значения не равны, тогда условие становится истинным.	(A != B) – true.
>	Проверяет, что значение левого операнда больше значения правого операнда, если да, тогда условие становится истинным.	(A > B) – false.
<	Проверяет, что значение левого операнда меньше значения правого операнда, если да, тогда условие становится истинным.	(A < B) – true.
>=	Проверяет, что значение левого операнда больше или равно значению правого операнда, если да, тогда условие становится истинным.	(A >= B) – false.
<=	Проверяет, что значение левого операнда меньше или равно значению правого операнда, если да, тогда условие становится истинным.	(A <= B) – true.

Пример

Попробуйте следующий пример, для понимания всех операторов сравнения, доступных в языке программирования D:

```
import std.stdio;

int main(string[] args) {
   int a = 21;
   int b = 10;
   int c ;

   if( a == b ) {
      writefln("Строка 1 - a равно b" );
   } else {
      writefln("Строка 1 - a не равно b" );
   }

   if ( a < b ) {
      writefln("Строка 2 - a меньше, чем b" );
   } else {
      writefln("Строка 2 - a не меньше, чем b" );
   }

   if ( a > b ) {
      writefln("Строка 3 - a больше, чем b" );
   } else {
      writefln("Строка 3 - a не больше, чем b" );
   }

   /* Давайте изменим значения a и b */
   a = 5;
   b = 20;

   if ( a <= b ) {
      writefln("Строка 4 - a либо меньше, либо равно b" );
   }
   if ( b >= a ) {
      writefln("Строка 5 - b больше или равно a" );
   }
   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Строка 1 - a не равно b
Строка 2 - a не меньше, чем b
Строка 3 - a больше, чем b
Строка 4 - a либо меньше, либо равно b
Строка 5 - b больше или равно a
```

Логические операторы

В следующей таблице показаны все логические операторы, поддерживаемые языком D. Предположим, что в переменной **A** содержится 1, а в переменной **B** – 0, тогда:

Оператор	Описание	Пример
&&	Называется логическим оператором И. Если оба операнда отличны от нуля, тогда условие становится истинным.	(A && B) – false.
||	Называется логическим оператором ИЛИ. Если любой из двух операндов отличен от нуля, тогда условие становится истинным.	(A || B) – true.
!	Называется оператором логического отрицания NOT. Используется для изменения логического состояния операнда. Если условие истинно, тогда логический оператор NOT сделает результат ложным.	!(A && B) – true.

Пример

Попробуйте следующий пример, для понимания всех логических операторов, доступных в языке программирования D:

```
import std.stdio;

int main(string[] args) {
   int a = 5;
   int b = 20;
   int c ;

   if ( a && b ) {
      writefln("Строка 1 - Условие истинно" );
   }
   if ( a || b ) {
      writefln("Строка 2 - Условие истинно" );
   }
   /* Давайте изменим значения a и b */

   a = 0;
   b = 10;

   if ( a && b ) {
      writefln("Строка 3 - Условие истинно" );
   } else {
      writefln("Строка 3 - Условие ложно" );
   }

   if ( !(a && b) ) {
      writefln("Строка 4 - Условие истинно" );
   }
   return 0;
}
```

Компиляция и выполнение приведённого выше кода даст следующий результат:

```
Строка 1 - Условие истинно
Строка 2 - Условие истинно
Строка 3 - Условие ложно
Строка 4 - Условие истинно
```

Побитовые операторы

Побитовые операторы работают с битами и выполняют операции с отдельными двоичными разрядами. Таблицы истинности для операторов &, |, и ^ выглядят так:

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Предположим, что **A** = 60; И **B** = 13. В двоичном формате они будут выглядеть следующим образом:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

Побитовые операторы, поддерживаемые языком D, перечислены в следующей таблице. Предположим, что переменная **A** содержит значение 60, а переменная **B** равна 13, тогда:

Оператор	Описание	Пример
&	Бинарный оператор И копирует бит в результат, если он существует в обоих операндах.	(A & B) даст 12, или двоичное 0000 1100.
|	Бинарный оператор ИЛИ копирует бит, если он существует в любом из операндов.	(A | B) даст 61, или двоичное 0011 1101.
^	Бинарный оператор Исключающее ИЛИ копирует бит, если он установлен в одном из операндов, но не в обоих.	(A ^ B) даст 49, или двоичное 0011 0001
~	Оператор Дополнения является унарным и имеет эффект «переворачивания» бит.	(~A) даст -61, или двоичное 1100 0011 в форме дополнения двойки.
<<	Бинарный оператор Сдвига влево. Значение левого операнда перемещается влево на количество бит, заданное правым операндом.	A << 2 даст 240, или двоичное 1111 0000
>>	Бинарный оператор Сдвига вправо. Значение левого операнда перемещается вправо на количество бит, заданное правым операндом.	A >> 2 даст 15, или двоичное 0000 1111.

Пример

Попробуйте следующий пример, для понимания всех побитовых операторов, доступных в языке программирования D:

```
import std.stdio;

int main(string[] args) {
   uint a = 60; /* 60 = 0011 1100 */
   uint b = 13; /* 13 = 0000 1101 */
   int c = 0;

   c = a & b;       /* 12 = 0000 1100 */
   writefln("Строка 1 - Значение c равно %d", c );

   c = a | b;       /* 61 = 0011 1101 */
   writefln("Строка 2 - Значение c равно %d", c );

   c = a ^ b;       /* 49 = 0011 0001 */
   writefln("Строка 3 - Значение c равно %d", c );

   c = ~a;          /*-61 = 1100 0011 */
   writefln("Строка 4 - Значение c равно %d", c );

   c = a << 2;     /* 240 = 1111 0000 */
   writefln("Строка 5 - Значение c равно %d", c );

   c = a >> 2;     /* 15 = 0000 1111 */
   writefln("Строка 6 - Значение c равно %d", c );

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Строка 1 - Значение c равно 12
Строка 2 - Значение c равно 61
Строка 3 - Значение c равно 49
Строка 4 - Значение c равно -61
Строка 5 - Значение c равно 240
Строка 6 - Значение c равно 15
```

Операторы присваивания

Следующие операторы присваивания поддерживаются в языке D:

Оператор	Описание	Пример
=	Это простой оператор присваивания. Он присваивает значения из правых операндов в левый операнд.	C = A + B присвоит значение суммы A + B в C
+=	Это оператор сложения И присвоения. Он складывает правый операнд с левым операндом и присваивает результат левому операнду.	C += A эквивалетно C = C + A
-=	Это оператор вычитания И присвоения. Он вычитает	C -= A эквивалетно C

	правый операнд из левого операнда и присваивает результат левому операнду.	= C - A
*=	Это оператор умножения И присвоения. Он умножает правый операнд на левый операнд и присваивает результат левому операнду.	C *= A эквивалетно C = C * A
/=	Это оператор деления И присвоения. Он делит левый операнд на правый операнд и присваивает результат левому операнду.	C /= A эквивалетно C = C / A
%=	Это оператор получения остатка от деления И присвоения. Он получает остаток от деления левого операнда на правый операнд и присваивает результат левому операнду.	C %= A эквивалетно C = C % A
<<=	Это оператор сдвига влево И присвоения.	C <<= 2 тоже самое, что и C = C << 2
>>=	Это оператор сдвига вправо И присвоения.	C >>= 2 тоже самое, что и C = C >> 2
&=	Это оператор бинарного И с последующим присвоением.	C &= 2 тоже самое, что и C = C & 2
^=	Это оператор бинарного Исключающего ИЛИ с последующим присвоением.	C ^= 2 тоже самое, что и C = C ^ 2
|=	Это оператор бинарного ИЛИ с последующим присвоением.	C |= 2 тоже самое, что и C = C | 2

Пример

Попробуйте следующий пример, чтобы понять все операторы присваивания, доступные на языке D:

```
import std.stdio;

int main(string[] args) {
   int a = 21;
   int c ;

   c =  a;
   writefln("Строка 1 - пример ореператора = , значение c = %d", c );

   c +=  a;
   writefln("Строка 2 - пример ореператора += , значение c = %d", c );

   c -=  a;
   writefln("Строка 3 - пример ореператора -= , значение c = %d", c );

   c *=  a;
   writefln("Строка 4 - пример ореператора *= , значение c = %d", c );

   c /=  a;
```

```
   writefln("Строка 5 - пример ореператора /= , значение c = %d", c );

   c  = 200;
   c = c % a;
   writefln("Строка 6 - пример ореператора %s= , значение c = %d",'\x25', c );

   c <<=  2;
   writefln("Строка 7 - пример ореператора <<= , значение c = %d", c );

   c >>=  2;
   writefln("Строка 8 - пример ореператора >>= , значение c = %d", c );

   c &=  2;
   writefln("Строка 9 - пример ореператора &= , значение c = %d", c );

   c ^=  2;
   writefln("Строка 10 - пример ореператора ^= , значение c = %d", c );

   c |=  2;
   writefln("Строка 11 - пример ореператора |= , значение c = %d", c );

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Строка 1 - пример ореператора = , значение c = 21
Строка 2 - пример ореператора += , значение c = 42
Строка 3 - пример ореператора -= , значение c = 21
Строка 4 - пример ореператора *= , значение c = 441
Строка 5 - пример ореператора /= , значение c = 21
Строка 6 - пример ореператора %= , значение c = 11
Строка 7 - пример ореператора <<= , значение c = 44
Строка 8 - пример ореператора >>= , значение c = 11
Строка 9 - пример ореператора &= , значение c = 2
Строка 10 - пример ореператора ^= , значение c = 0
Строка 11 - пример ореператора |= , значение c = 2
```

Другие операторы

В языке D существует ещё несколько важных операторов, включая **sizeof** и **? :**

Оператор	Описание	Пример
sizeof()	Возвращает размер памяти, занимаемый переменной.	sizeof(a), где a - целое число типа int, возвращает 4.
&	Возвращает адрес переменной.	&a; даст фактический адрес переменной a.
*	Указатель на переменную.	*a; даст указатель на переменную.
? :	Условное выражение	Если условие истинно, тогда значение X: иначе значение Y.

Пример

Попробуйте следующий пример, чтобы понять эти операторы:

```
import std.stdio;
```

```
int main(string[] args) {
   int a = 4;
   short b;
   double c;
   int* ptr;

   /* пример оператора sizeof */
   writefln("Строка 1 - Размер переменной a = %d", a.sizeof );
   writefln("Строка 2 - Размер переменной b = %d", b.sizeof );
   writefln("Строка 3 - Размер переменной c= %d", c.sizeof );

   /* пример операторов & and * */
   ptr = &a; /* 'ptr' теперь содержит адрес переменной 'a'*/
   writefln("Значение a равно %d", a);
   writefln("*ptr равно %d.", *ptr);

   /* пример оператора условного выражения */
   a = 10;
   b = (a == 1) ? 20: 30;
   writefln( "Значение b равно %d", b );

   b = (a == 10) ? 20: 30;
   writefln( "Значение b равно %d", b );
   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Строка 1 - Размер переменной a = 4
Строка 2 - Размер переменной b = 2
Строка 3 - Размер переменной c= 8
Значение a равно 4
*ptr равно 4.
Значение b равно 30
Значение b равно 20
```

Приоритеты операторов в D

Приоритет оператора определяет группирование термов в выражении. Это влияет на вычисление выражения. Некоторые операторы имеют приоритет над другими.

Например, оператор умножения имеет более высокий приоритет, чем оператор сложения.

Рассмотрим выражение

x = 7 + 3 * 2.

Здесь в x присваивается 13, а не 20. Просто дело в том, что оператор * имеет более высокий приоритет, чем +, поэтому сначала вычисляется 3 * 2, а затем результат прибавляется к 7.

В этой таблице операторы с наивысшим приоритетом показаны в верхней части, а с наименьшим – внизу. Внутри выражения сначала вычисляются операторы с более высоким приоритетом.

Категория	Оператор	Ассоциативность
Суффикс	() [] -> . ++ - -	Слева направо
Унарные	+ - ! ~ ++ - - (type)* & sizeof	Справа налево
Умножение/деление	* / %	Слева направо
Сложение/вычитание	+ -	Слева направо
Сдвиг	<< >>	Слева направо
Логические	< <= > >=	Слева направо
Равенство	== !=	Слева направо
Побитовое И	&	Слева направо
Побитовое Исключающее ИЛИ	^	Слева направо
Побитовое ИЛИ	|	Слева направо
Логическое И	&&	Слева направо
Логическое ИЛИ	||	Слева направо
Условное	?:	Справа налево
Присвоение	= += -= *= /= %=>>= <<= &= ^= | =	Справа налево
Запятая	,	Слева направо

Пример

Попробуйте следующий пример, чтобы понять приоритеты операторов, доступных в языке D:

```
import std.stdio;

int main(string[] args) {
   int a = 20;
   int b = 10;
   int c = 15;
   int d = 5;
   int e;

   e = (a + b) * c / d;       // ( 30 * 15 ) / 5
   writefln("Значение (a + b) * c / d равно : %d",  e );

   e = ((a + b) * c) / d;     // (30 * 15 ) / 5
   writefln("Значение ((a + b) * c) / d равно : %d" ,  e );

   e = (a + b) * (c / d);    // (30) * (15/5)
   writefln("Значение (a + b) * (c / d) равно : %d",  e );

   e = a + (b * c) / d;      //  20 + (150/5)
   writefln("Значение a + (b * c) / d равно : %d" ,  e );

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

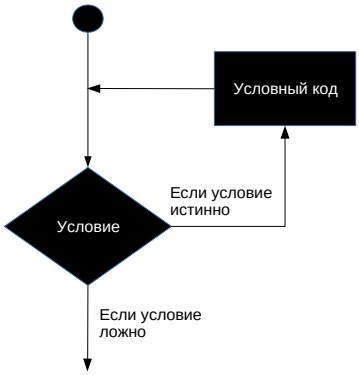
```
Значение (a + b) * c / d равно : 90
Значение ((a + b) * c) / d равно : 90
Значение (a + b) * (c / d) равно : 90
Значение a + (b * c) / d равно : 50
```

Циклы

Может возникнуть ситуация, когда вам нужно выполнить блок кода несколько раз. Обычно операторы выполняются последовательно: сначала выполняется первый оператор в функции, затем второй и т. д.

Языки программирования предоставляют различные управляющие структуры, которые позволяют выполнять программу более сложные путями.

Оператор цикла выполняет оператор или группу операторов несколько раз. Вот общая форма оператора цикла, обычно используемая в языках программирования:



Язык программирования D предоставляет следующие типы циклов. Нажмите на ссылки, чтобы изучить их подробнее.

Номер	Тип цикла и Описание
1	**Цикл while** Повторяет оператор или группу операторов, пока данное условие истинно. Условие проверяется перед выполнением тела цикла.
2	**Цикл for** Выполняет последовательность операторов несколько раз и сокращает код, управляющий переменной цикла.
3	**Цикл do...while** Аналогичен оператору while, за исключением того, что условие проверяется в конце тела цикла.
4	**Вложенные циклы** Вы можете использовать один или несколько циклов внутри другого цикла while, for, или do..while.

*Авторы не упомянули ещё один вид цикла в языке D – оператор **foreach**. Этот оператор предназначен для итерирования по содержимому составных конструкций (агрегатов), например: по массивам, по ассоциативным массивам, по символам в строке, по*

диапазонам, по кортежам, также этот оператор можно применять к классам и структурам, у которых определена специальная функция ***opApply****. Вот пример перебора символов в строке:*

```
import std.stdio;
void main() {
    foreach (char c; "12345")
    {
        writeln(c);
    }
}
```

— *доп. пер.*

Операторы управления циклом

Операторы управления циклом изменяют обычную последовательность выполнения. Когда выполнение оставляет область видимости, все автоматические объекты, созданные в этой области, уничтожаются.

D поддерживает следующие управляющие операторы:

Номер	Оператор управления и Описание
1	**Оператор break** Завершает цикл или оператор switch и передает выполнение оператору, стоящему сразу после цикла или switch.
2	**Оператор continue** Заставляет цикл пропустить оставшуюся часть его тела и сразу же перейти к условию следующего повтора цикла.

Бесконечный цикл

Цикл становится бесконечным циклом, если условие никогда не станет ложным. Традиционно для этой цели используется цикл **for**. Поскольку ни одно из трёх выражений, которые образуют цикл for, не является обязательным, вы можете сделать бесконечный цикл, оставив условное выражение пустым.

```
import std.stdio;

int main () {

   for( ; ; ) {
      writefln("Этот цикл будет работать вечно.");
   }
   return 0;
}
```

Когда условное выражение отсутствует, оно считается истинным. У вас может быть выражение инициализации и инкремента, но программисты на D чаще используют конструкцию for(;;) для обозначения бесконечного цикла.

ЗАМЕЧАНИЕ − Вы можете завершить бесконечный цикл, нажав клавиши Ctrl + C.

Цикл while

Оператор цикла **while** в языке D многократно выполняет целевой оператор, пока заданное условие имеет истинное значение.

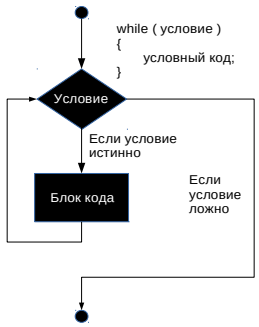
Синтаксис

Синтаксис цикла **while** в языке D:

```
while(условие) {
   тело_цикла
}
```

Здесь **тело_цикла** может представлять собой один оператор или блок операторов. **Условием** может быть любое выражение, а его истинным значением – любое ненулевое значение. Цикл повторяется, пока условие истинно.

Диаграмма потока выполнения



Здесь видно, что ключевым моментом цикла *while* является то, что цикл может никогда не запуститься. Если условие проверено и результат оказался ложным, тело цикла пропускается, и выполняется первый оператор, расположенный после цикла while.

Пример

```
import std.stdio;

int main () {
   /* Определение локальной переменной */
   int a = 10;

   /* Выполнение цикла while */
   while( a < 20 ) {
      writefln("значение a: %d", a);
      a++;
   }

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
значение a: 10
значение a: 11
значение a: 12
значение a: 13
значение a: 14
```

```
значение a: 15
значение a: 16
значение a: 17
значение a: 18
значение a: 19
```

Цикл for

Цикл **for** – это структура управления, позволяющая эффективно писать цикл, который должен выполняться определённое количество раз.

Синтаксис

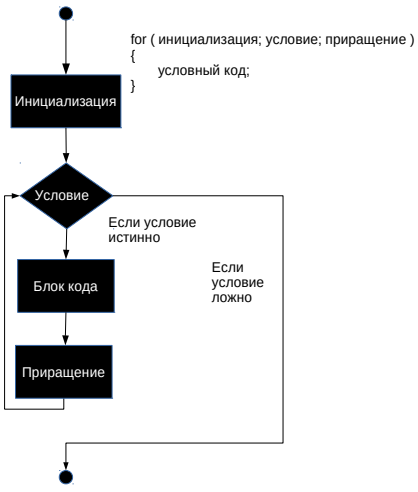
Синтаксис цикла **for** в языке D:

```
for ( инициализация; условие; приращение ) {
   тело_цикла;
}
```

Вот поток управления во время цикла for:

- Шаг **инициализации** выполняется в самом начале, и только один раз. Этот шаг позволяет вам объявлять и инициализировать любые переменные, управляющие циклом. Вы не обязаны помещать оператор сюда, до точки с запятой.
- Затем вычисляется **условие**. Если оно истинно, выполняется тело цикла. Если оно ложно, тело цикла не выполняется и поток выполнения переходит к оператору, следующему сразу после цикла for.
- После того, как тело цикла for выполнится, поток управления переходит к инструкции **приращение**. Этот оператор позволяет обновить любые переменные, управляющие циклом. Этот оператор, т.е. всё место после точки с запятой, расположенной в конце условия, можно оставить пустым.
- Условие теперь снова вычисляется. Если оно истинно, цикл выполняется, и процесс повторяется (тело цикла, затем шаг приращения, а затем снова условие). После того, как условие станет ложным, цикл for завершится.

Диаграмма потока выполнения



Пример

```
import std.stdio;

int main () {
   /* Выполнение цикла for */
   for( int a = 10; a < 20; a = a + 1 ) {
      writefln("Значение a: %d", a);
   }

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Значение a: 10
Значение a: 11
Значение a: 12
Значение a: 13
Значение a: 14
Значение a: 15
Значение a: 16
Значение a: 17
Значение a: 18
Значение a: 19
```

Цикл Do...While

В отличие от циклов **for** и **while**, которые проверяют условие в верхней части цикла, цикл **do...while** в языке D проверяет своё условие в нижней части.

Цикл **do...while** похож на цикл while, за исключением того, что цикл do...while гарантированно выполнится хотя бы один раз.

Синтаксис

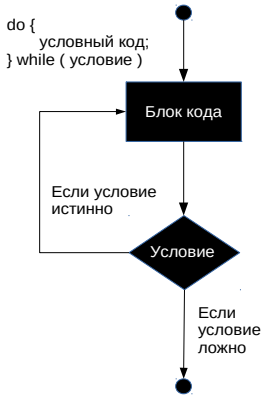
Синтаксис цкла **do...while** в языке D:

```
do {
   тело_цикла;
} while( условие );
```

Обратите внимание, что условное выражение появляется в конце цикла, поэтому **тело_цикла** выполнится один раз до проверки **условия**.

Если условие истинно, поток управления перескакивает назад, и тело_цикла выполняется снова. Этот процесс повторяется до тех пор, пока данное условие не станет ложным.

Диаграмма потока выполнения



Пример

```
import std.stdio;

int main () {
   /* Определение локальной переменной */
   int a = 10;

   /* Выполнение цикла do */
   do{
      writefln("Значение a: %d", a);
      a = a + 1;
   }while( a < 20 );

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Значение a: 10
Значение a: 11
Значение a: 12
Значение a: 13
Значение a: 14
Значение a: 15
Значение a: 16
Значение a: 17
Значение a: 18
Значение a: 19
```

Вложенные циклы

Язык D позволяет использовать один цикл внутри другого цикла. В следующем подразделе показаны несколько примеров, иллюстрирующих эту возможность.

Синтаксис

Синтаксис **вложенных циклов for** выглядит следующим образом:

```
for ( инициализация; условие; приращение ) {
   for ( инициализация; условие; приращение ) {
      тело_цикла;
   }
   остальное_тело_цикла;
```

```
}
```

Синтаксис **вложенных циклов while** выглядит следующим образом:

```
while(условие) {
   while(условие) {
      тело_цикла;
   }
   остальное_тело_цикла;
}
```

Синтаксис **вложенных циклов do...while** выглядит следующим образом:

```
do {
   тело_цикла;
   do {
      тело_цикла;
   }while( условие );

}while( условие );
```

Последним замечанием о вложенности циклов скажем, что вы можете поместить цикл любого типа внутри цикла любого другого типа. Например, цикл for может находиться внутри цикла while или наоборот.

Пример

Следующая программа использует вложенные циклы for для поиска простых чисел от 2 до 100:

```
import std.stdio;

int main () {
   /* Определение локальных переменных */
   int i, j;

   for(i = 2; i<100; i++) {
      for(j = 2; j <= (i/j); j++)
        if(!(i%j)) break; // Если найден множитель, простым не является
      if(j > (i/j)) writefln("%d является простым числом", i);
   }

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
2 является простым числом
3 является простым числом
5 является простым числом
7 является простым числом
11 является простым числом
13 является простым числом
17 является простым числом
19 является простым числом
23 является простым числом
29 является простым числом
31 является простым числом
37 является простым числом
41 является простым числом
43 является простым числом
47 является простым числом
53 является простым числом
```

```
59 является простым числом
61 является простым числом
67 является простым числом
71 является простым числом
73 является простым числом
79 является простым числом
83 является простым числом
89 является простым числом
97 является простым числом
```

Оператор break

Оператор **break** в языке D имеет следующие два варианта использования:

- Когда оператор break встречается внутри цикла, цикл немедленно прекращается, и выполнение программы продолжается со следующего оператора после цикла.
- Его можно использовать для завершения case-инструкции в операторе switch (см. в следующей главе).

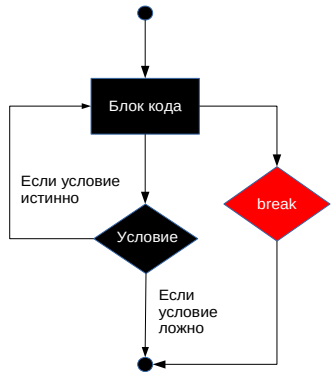
Если вы используете вложенные циклы (т.е. один цикл внутри другого цикла), оператор break останавливает выполнение самого внутреннего цикла и начинает выполнение следующей строки кода после этого блока.

Синтаксис

Синтаксис оператора **break** в D следующий:

```
break;
```

Диаграмма потока выполнения



Пример

```
import std.stdio;

int main () {
   /* Определение локальной переменной */
   int a = 10;

   /* Выполнение цикла while */
   while( a < 20 ) {
      writefln("Значение a: %d", a);
      a++;

      if( a > 15) {
```

```
         /* Завершение цикла с использованием оператора break */
         break;
      }
   }

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Значение a: 10
Значение a: 11
Значение a: 12
Значение a: 13
Значение a: 14
Значение a: 15
```

Оператор continue

Оператор **continue** в языке D работает похоже на оператор break. Однако вместо принудительного завершения, заставляет выполнить следующую итерацию цикла, игнорируя весь код после себя.

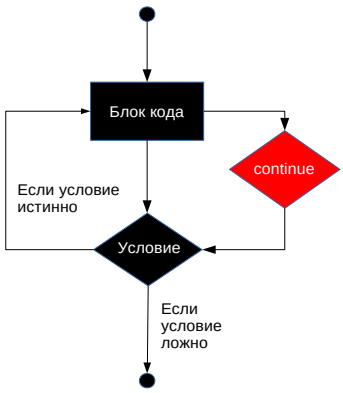
Для цикла **for** оператор **continue** вызывает выполнение частей цикла тест условия и приращение. Для цикла **while** и **do...while**, оператор **continue** вызывает переход программы к тесту условия.

Синтаксис

Синтаксис оператора **continue** в D следующий:

```
continue;
```

Диаграмма потока выполнения



Пример

```
import std.stdio;

int main () {
   /* Определение локальной переменной */
   int a = 10;

   /* Выполнение цикла do */
   do {
```

```
      if( a == 15) {
         /* пропустить итерацию */
         a = a + 1;
         continue;
      }
      writefln("Значение a: %d", a);
      a++;

   } while( a < 20 );

   return 0;
}
```

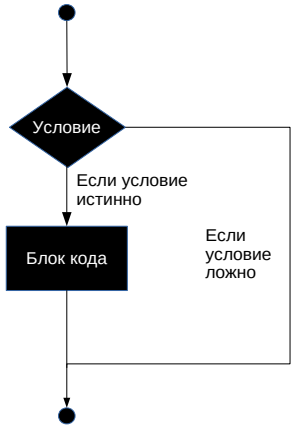
Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Значение a: 10
Значение a: 11
Значение a: 12
Значение a: 13
Значение a: 14
Значение a: 16
Значение a: 17
Значение a: 18
Значение a: 19
```

Условные операторы

Условные структуры содержат условие, которое будет вычислено, вместе с двумя наборами выполняемых операторов. Один набор операторов выполняется, если условие оказалось истинным, а другой набор операторов выполняется, если условие ложное.

Ниже приведён общий вид типичной условной структуры, которую можно найти в большинстве языков программирования:



Язык программирования D принимает любые **ненулевые** и **не-null** значения как истинные (**true**), а равные **нулю** или **null** принимаются как ложные значения (**false**).

Язык D предлагает следующие типы условных операторов.

Номер	Оператор и Описание
1	**оператор if** **Оператор if** состоит из логического выражения, за которым следуют один или несколько операторов.
2	**оператор if...else** За **оператором if** может следовать необязательный **оператор else**, который выполняется, когда логическое выражение ложно.
3	**вложенные операторы if** Вы можете использовать один оператор **if** или **if else** внутри другого оператора **if** или **if else**.
4	**оператор switch** Оператор **switch** позволяет проверять переменную на равенство по отношению к списку значений.
5	**вложенные операторы switch** Вы можете использовать один оператор **switch** внутри другого.

Оператор ? : в языке D

Мы уже упоминали оператор **условного выражения ? :** в одной из предыдущих глав, его можно использовать вместо if...else. Он имеет следующий вид:

```
Выр1 ? Выр2 : Выр3;
```

Где Выр1, Выр2 и Выр3 являются выражениями. Обратите внимание на использование и размещение двоеточия.

Значение ?-выражения определяется следующим образом:

- Вычисляется значение Выр1. Если оно истинно, то вычисляется Выр2, и его значение становится значением всего ?-выражения.
- Если Выр1 является ложным, тогда вычисляется Выр3, и его значение становится значением выражения.

Оператор if

Оператор **if** состоит из логического выражения, за которым следуют один или несколько операторов.

Синтаксис

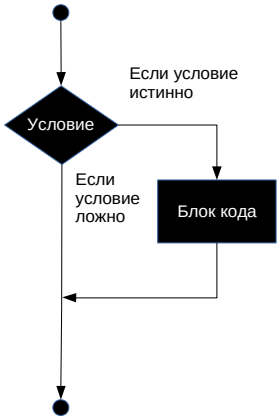
Синтаксис оператора **if** в языке D выглядит так:

```
if(логическое_выражение) {
   /* Оператор(ы) будет выполняться, если логическое выражение истинно */
}
```

Если логическое_выражение вычислено в **true**, тогда выполняется блок кода внутри оператора if. Если логическое_выражение вычислено в **false**, тогда выполняется первый оператор после оператора if (после закрытия фигурной скобки).

Язык программирования D принимает любые **ненулевые** и **не-null** значения как истинные (**true**), а равные **нулю** или **null** принимаются как ложные значения (**false**).

Диаграмма потока выполнения



Пример

```
import std.stdio;
```

```
int main () {
   /* Определение локальной переменной */
   int a = 10;

   /* Проверить логическое условие, используя оператор if */
   if( a < 20 ) {
      /* Если условие истинно, то вывести следующее */
      writefln("a меньше, чем 20" );
   }
   writefln("значение a равно : %d", a);

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
a меньше, чем 20
значение a равно : 10
```

Оператор if...else

За оператором **if** может следовать необязательный оператор **else**, который выполняется, когда логическое выражение ложно.

Синтаксис

Синтаксис оператора **if...else** в языке D выглядит так

```
if(логическое_выражение) {
   /* Оператор(ы) будет выполняться, если логическое выражение истинно */
} else {
   /* Оператор(ы) будет выполняться, если логическое выражение ложно */
}
```

Если логическое_выражение вычислено в **true**, тогда выполняется **блок кода if**, в противном случае выполняется **блок кода else**.

Язык программирования D принимает любые **ненулевые** и **не-null** значения как истинные (**true**), а равные **нулю** или **null** принимаются как ложные значения (**false**).

Диаграмма потока выполнения

Пример

```
import std.stdio;

int main () {
   /* Определение локальной переменной */
   int a = 100;

   /* Проверить логическое условие */
   if( a < 20 ) {
      /* Если условие истинно, вывести следующее */
      writefln("a меньше, чем 20" );
   } else {
      /* Если условие ложно, вывести следующее */
      writefln("a не меньше, чем 20" );
   }
   writefln("значение a равно : %d", a);

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
a не меньше, чем 20
значение a равно : 100
```

Оператор if...else if...else

За инструкцией **if** может следовать необязательный оператор **else if...else**, который очень полезен для проверки различных условий с использованием единичного оператора if ... else if.

При использовании команд if, else if, else следует иметь в виду несколько моментов:

- *if* может иметь ноль или одно *else*, и оно должно находиться после всех *else if*.
- *if* может иметь ноль или множество *else if*, и они должны находиться перед *else*.
- Как только *else if* сработает, никакие из оставшихся *else if* или *else* не проверяются.

Синтаксис

Синтаксис оператора **if...else if...else** в языке D:

```
if(логическое_выражение 1) {
   /* Выполняется, когда логическое_выражение 1 является истиной */
} else if( логическое_выражение 2) {
   /* Выполняется, когда логическое_выражение 2 является истиной */
} else if( логическое_выражение 3) {
   /* Выполняется, когда логическое_выражение 3 является истиной */
} else {
   /* Выполняется, когда ни одно из указанных выше условий не является истиной */
}
```

Пример

```
import std.stdio;

int main () {
   /* Определение локальной переменной */
   int a = 100;
```

```
   /* Проверить логическое условие */
   if( a == 10 ) {
      /* Если условие истинно, вывести следующее */
      writefln("Значение a равно 10" );
   } else if( a == 20 ) {
      /* Если условие else if истинно */
      writefln("Значение a равно 20" );
   } else if( a == 30 ) {
      /* Если условие else if истинно  */
      writefln("Значение a равно 30" );
   } else {
      /* ни одно из условий не является истиной */
      writefln("Ни одно из значений не соответствует" );
   }
   writefln("Точное значение a равно: %d", a );

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Ни одно из значений не соответствует
Точное значение a равно: 100
```

Вложенные операторы if

В программировании на D всегда допустимо вставлять друг в друга выражения if-else, что означает, что вы можете использовать один оператор if или else if внутри другого оператора if или else if.

Синтаксис

Синтаксис **вложенных if** следующий:

```
if( логическое_выражение 1) {
   /* Выполняется, когда логическое_выражение 1 является истиной */
   if(логическое_выражение 2) {
      /* Выполняется, когда логическое_выражение 2 является истиной */
   }
}
```

Вы можете вставлять **else if...else**, аналогично тому, как вы вставляете оператор *if*.

Пример

```
import std.stdio;

int main () {
   /* Определение локальных переменных */
   int a = 100;
   int b = 200;

   /* Проверить логическое условие */
   if( a == 100 ) {
      /* Если условие истинно, проверить следующее */
      if( b == 200 ) {
         /* Если условие истинно, вывести следующее */
         writefln("Значение a равно 100 и b равно 200" );
      }
   }
   writefln("Точное значение a равно : %d", a );
```

```
   writefln("Точное значение b равно : %d", b );

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Значение a равно 100 и b равно 200
Точное значение a равно : 100
Точное значение b равно : 200
```

Оператор switch

Оператор **switch** (*слово переводится как "переключатель" – прим. пер.*) позволяет проверять переменную на равенство со списком значений. Каждое значение называется случаем (**case**), а коммутируемая переменная сравнивается с каждым случаем.

Синтаксис

Синтаксис оператора **switch** в языке D следующий:

```
switch(выражение) {
   case константное_выражение  :
      оператор(ы);
      break;

   case константное_выражение  :
      оператор(ы);
      break;
      * Вы можете иметь любое количество операторов case */

   default :
      оператор(ы);
}
```

К инструкции **switch** применяются следующие правила:

*Со времени написания этого учебника тонкости в работе оператора **switch** поменялись несколько раз, поэтому я позволил себе переделать нижеследующий список – прим. пер.*

- **Выражение**, используемое в операторе **switch** может иметь целый, перечисляемый или строковый тип.
- В **switch** может быть любое количество операторов **case**. За каждым **case** следует сравниваемое значение, с последующим двоеточием.
- **Константное_выражение** для каждого **case** должно быть тем же типом данных, что и переменная в **switch**, и должно быть константой или литералом.
- Когда коммутируемая переменная оказывается равной **case**, операторы, следующие за этим **case**, выполняются до тех пор, пока не будет достигнут один из операторов, прерывающих прямое выполнение программы: **break**, **continue**, **return** или **goto case**. Присутствие одного из них в конце списка операторов обязательно. Также допустим оператор **throw**, вызывающий исключение.

•Когда достигнут оператор **break**, **switch** завершается, и поток выполнения переходит к следующей строке, следующей за оператором **switch**.

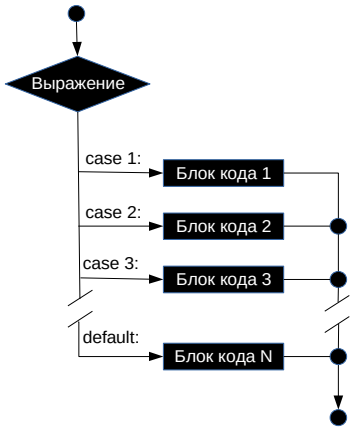
•Когда достигнут оператор **continue**, **switch** завершается, и поток выполнения переходит к следующей итерации цикла, внутри которого находится оператор **switch**.

•Когда достигнут оператор **goto case**, поток выполнения "падает" в следующий оператор **case**.

•"Тело" оператора **case** может быть пустым, тогда поток выполнения переходит к операторам в следующим за ним **case**.

•В операторе **switch** должен присутствовать случай по-умолчанию **default**, который выполняется, если ни один из случаев не является истинным. **break** или какой-либо другой прерывающий прямое выполнение оператор здесь также обязателен.

Диаграмма потока выполнения



Пример

```
import std.stdio;

int main () {
   /* Определение локальной переменной */
   char grade = 'B';
   switch(grade) {
      case 'A' :
         writefln("Великолепно!" );
         break;
      case 'B' :
      case 'C' :
         writefln("Хорошо" );
         break;
      case 'D' :
         writefln("Вы прошли" );
         break;
      case 'F' :
         writefln("Лучше повторите попытку" );
         break;
      default :
         writefln("Недопустимая оценка" );
   }
```

```
   writefln("Ваша оценка %c", grade );

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Хорошо
Ваша оценка B
```

Вложенные операторы switch

Допустимо иметь switch как часть последовательности операторов внешнего switch. Даже если case-константы внутреннего и внешнего switch содержат общие значения, конфликтов не возникает.

Синтаксис

Синтаксис для вложенного оператора switch выглядит следующим образом:

```
switch(ch1) {
   case 'A':
      writefln("Эта A является частью внешнего switch" );
      switch(ch2) {
         case 'A':
            writefln("Эта A является частью внутреннего switch" );
            break;
         case 'B': /* case-код */
            break;
         default:
            break;
      }
      break;
   case 'B': /* case-код */
      break;
   default:
      break;
}
```

Пример

```
import std.stdio;

int main () {
   /* Определение локальных переменных */
   int a = 100;
   int b = 200;

   switch(a) {
      case 100:
         writefln("Это часть внешнего switch", a );
         switch(b) {
            case 200:
               writefln("Это часть внутреннего switch", a );
               break;
            default:
               break;
         }
         break;
      default:
         break;
```

```
   }
   writefln("Точное значение a равно: %d", a );
   writefln("Точное значение b равно: %d", b );

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Это часть внешнего switch
Это часть внутреннего switch
Точное значение a равно: 100
Точное значение b равно: 200
```

Функции

В этой главе описываются функции, используемые в программировании на D.

Определение функции в D

В своей основе определение функции состоит из заголовка функции и тела функции.

СИНТАКСИС

```
возвращаемый_тип имя_функции( список_параметров ) {
   тело_функции
}
```

Вот описание всех частей функции:

- **Возвращаемый тип** – Функция может возвращать значение. Тип **возвращаемый_тип** – это тип данных значения, возвращаемого функцией. Некоторые функции выполняют свои действия без возврата значения. В этом случае в качестве **возвращаемого_типа** ставится ключевое слово **void**.
- **Имя функции** − Это фактическое имя функции. Имя функции и список параметров совместно составляют сигнатуру функции.
- **Параметры** − Параметр похож на заполнитель. Когда функция вызывается, вы передаете значение параметру. На это значение ссылается фактический параметр или аргумент. **Список_параметров** определяет тип, порядок и количество параметров функции. Параметры являются необязательными, то есть функция может не содержать никаких параметров.
- **Тело функции** − Тело функции содержит набор операторов, которые определяют, что делает функция.

Вызов функции

Вы можете вызвать функцию следующим образом:

```
имя_функции(список_параметров)
```

Типы функций в D

D поддерживает широкий спектр функций, и они перечислены ниже.

- Чистые функции
- Функции Nothrow
- Ref-Функции
- Функции Auto
- Функции с переменным числом аргументов
- Inout-Функции
- Функции-свойства

Ниже описаны различные функции.

Чистые функции

Чистые функции (pure) – это функции, которые не могут получить доступ к глобальному или статическому, изменяемому состоянию, сохранять его через свои аргументы. Таким образом появляется возможность включить оптимизацию, основанную на том факте, что чистая функция, как гарантируется, не изменяет ничего, что не передается ей, а в тех случаях, когда компилятор может гарантировать, что чистая функция не может изменить свои аргументы, она может обеспечить полную функциональную чистоту, то есть гарантировать, что функция всегда вернёт один и тот же результат для одних и тех же аргументов.

```
import std.stdio;

int x = 10;
immutable int y = 30;
const int* p;

pure int purefunc(int i,const char* q,immutable int* s) {
   //writeln("Simple print"); //нельзя вызывать "нечистую" функцию 'writeln'

   debug writeln("in foo()"); // можно, "нечистый" код допустим в операторе debug
   // x = i;  // ошибка, изменение глобального состояния
   // i = x;  // ошибка, чтение изменяемого глобального состояния
   // i = *p; // ошибка, чтение константного глобального состояния
   i = y;     // можно, чтение неизменяемого глобального состояния
   auto myvar = new int;     // Можно использовать выражение new
   return i;
}

void main() {
   writeln("Значение, возвращаемое из чистой функции : ",purefunc(x,null,null));
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Значение, возвращаемое из чистой функции : 30
```

Функции Nothrow

Функции Nothrow не выбрасывают никаких исключений, унаследованных от класса Exception.

Nothrow гарантирует, что функция не выбрасывает никаких исключений.

```
import std.stdio;

int add(int a, int b) nothrow {
   //writeln("сложение"); Это не удастся, потому что writeln может бросать исключения
   int result;

   try {
      writeln("сложение"); // компилируется
      result = a + b;
   } catch (Exception error) { // ловит все исключения
   }
```

```
   return result;
}

void main() {
   writeln("Результат сложения: ", add(10,20));
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
сложение
Результат сложения: 30
```

Ref-Функции

Ref-функции позволяют возвращать значение функции по ссылке. Это аналогично ref-параметрам функции.

```
import std.stdio;

// "greater" переводится как "больший"
ref int greater(ref int first, ref int second) {
   return (first > second) ? first : second;
}

void main() {
   int a = 1;
   int b = 2;

   greater(a, b) += 10;
   writefln("a: %s, b: %s", a, b);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
a: 1, b: 12
```

Видимо, тут необходимо небольшое пояснение. Ref-параметры дают возможность функции изменять их значение в области видимости, из которой функция вызывается. Значение, возвращённое по ссылке из ref-функции, является «левым значением» (lvalue), т. е. допустимо ставить его с левой стороны оператора присваивания, что и сделано в этом примере — прим. пер.

Auto-Функции

Auto-Функции могут возвращать значение любого типа. Нет никаких ограничений на возвращаемый тип. Ниже приведён простой пример функции типа auto.

```
import std.stdio;

auto add(int first, double second) {
   double result = first + second;
   return result;
}

void main() {
   int a = 1;
   double b = 2.5;

   writeln("add(a,b) = ", add(a, b));
```

```
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
add(a,b) = 3.5
```

Функции с переменным числом аргументов

Функции с переменным числом аргументов (variadic) являются такими функциями, в которых число параметров функции определяется во время выполнения. В языке C существует ограничение, должен быть по крайней мере один параметр. Но в языке D нет такого ограничения. Ниже приведен простой пример.

```
import std.stdio;
import core.vararg;

void printargs(int x, ...) {
   for (int i = 0; i < _arguments.length; i++) {
      write(_arguments[i]);

      if (_arguments[i] == typeid(int)) {
         int j = va_arg!(int)(_argptr);
         writefln("\t%d", j);
      } else if (_arguments[i] == typeid(long)) {
         long j = va_arg!(long)(_argptr);
         writefln("\t%d", j);
      } else if (_arguments[i] == typeid(double)) {
         double d = va_arg!(double)(_argptr);
         writefln("\t%g", d);
      }
   }
}

void main() {
   printargs(1, 2, 3L, 4.5);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
int 2
long 3
double 4.5
```

*Я попытался выяснить, как это работает, что привело меня сюда: [opennet.ru](). Также добавлю, что для variadic-функций неявно определены две переменные: массив **_arguments** из типов этих аргументов, а также **_argptr**, которую нужно вызывать через шаблон **va_arg**, определённый в модуле библиотеки core.vararg, при каждом вызове _argptr мы получаем следующий аргумент — прим. пер.*

Inout-Функции

Inout может использоваться как для параметров, так и для возвращаемых типов функций. Это как шаблон для квалификаторов изменяемости: mutable, const и immutable. Атрибут изменяемости выводится из параметра. Значит, inout переносит выведенный атрибут изменяемости на возвращаемый тип. Ниже показан простой пример, показывающий, как меняется квалификатор изменяемости.

```
import std.stdio;

inout(char)[] qoutedWord(inout(char)[] phrase) {
   return '"' ~ phrase ~ '"';
}

void main() {
   char[] a = "проверка a".dup;

   a = qoutedWord(a);
   writeln(typeof(qoutedWord(a)).stringof," ", a);

   const(char)[] b = "проверка b";
   b = qoutedWord(b);
   writeln(typeof(qoutedWord(b)).stringof," ", b);

   immutable(char)[] c = "проверка c";
   c = qoutedWord(c);
   writeln(typeof(qoutedWord(c)).stringof," ", c);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
char[] "проверка a"
const(char)[] "проверка b"
string "проверка c"
```

Функции-свойства

Свойства (properties) позволяют использовать функции-члены как данные-члены. Они используют ключевое слово @property. Свойства связаны с родственной функцией, возвращающей значение, когда оно требуется. Ниже приведен простой пример свойства.

```
import std.stdio;

// прямоугольник
struct Rectangle {
   double width; // ширина
   double height; // высота

   // площадь
   double area() const @property {
      return width*height;
   }

   void area(double newArea) @property {
      auto multiplier = newArea / area;
      width *= multiplier;
      writeln("Значение установлено!");
   }
}

void main() {
   auto rectangle = Rectangle(20,10);
   writeln("Площадь равна ", rectangle.area);

   rectangle.area = 300;
   writeln("Изменённая ширина равна ", rectangle.width);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Площадь равна 200
Значение установлено!
Изменённая ширина равна 30
```

Символы

Символы – это строительные блоки строк. В системе письма всё называется символом: буквы алфавитов, цифры, знаки препинания, символ пробела и т.д. Несколько добавляет путаницы тот факт, что строительные блоки самих символов также называются символами.

Целочисленное значение буквы **a** в нижнем регистре равно 97, а целочисленное значение цифры **1** равно 49. Эти значения были назначены просто по соглашениям, при разработке таблицы ASCII.

В следующей таблице указаны стандартные символьные типы с размерами занимаемой ими памяти и целями.

Символы представлены типом char, который может содержать только 256 различных значений. Если вы знакомы с типом char по другим языкам, вы, возможно, уже знаете, что его размера недостаточно для поддержки букв большого количества систем письменности.

Тип	Занимаемый размер	Цель
char	1 байт	кодовый блок UTF-8
wchar	2 байта	кодовый блок UTF-16
dchar	4 байта	кодовый блок UTF-32 и кодовая точка Unicode

Ниже перечислены некоторые полезные функции, работающие с символами:

- **isLower** − Определяет, является ли строчным символом.
- **isUpper** − Определяет, является ли символом верхнего регистра.
- **isAlpha** − Определяет, является ли алфавитно-цифровым символом Юникода (как правило, буквой или цифрой).
- **isWhite** − Определяет, является ли пробельным символом.
- **toLower** − Возвращает данный символ в нижнем регистре.
- **toUpper** − Возвращает данный символ в верхнем регистре.

```
import std.stdio;
import std.uni;

void main() {
   writeln("ğ - строчная буква? ", isLower('ğ'));
   writeln("Ş - строчная буква? ", isLower('Ş'));

   writeln("İ - прописная буква? ", isUpper('İ'));
   writeln("ç - прописная буква? ", isUpper('ç'));

   writeln("z - это алфавино-цифровой символ? ",       isAlpha('z'));
   writeln("Перевод строки является пробельным символом? ",  isWhite('\n'));

   writeln("Нижнее подчёркивание является пробельным символом? ", isWhite('_'));

   writeln("Буква Ğ в нижнем регистре: ", toLower('Ğ'));
   writeln("Буква İ в нижнем регистре: ", toLower('İ'));
```

```
   writeln("Буква ş в верхнем регистре: ", toUpper('ş'));
   writeln("Буква ı в верхнем регистре: ", toUpper('ı'));
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
ğ - строчная буква? true
Ş - строчная буква? false
İ - прописная буква? true
ç - прописная буква? false
z - это алфавино-цифровой символ? true
Перевод строки является пробельным символом? true
Нижнее подчёркивание является пробельным символом? false
Буква Ğ в нижнем регистре: ğ
Буква İ в нижнем регистре: i
Буква ş в верхнем регистре: Ş
Буква ı в верхнем регистре: I
```

Чтение символов в D

Мы можем читать символы, используя *readf*, как показано ниже.

```
readf(" %s", &letter);
```

Поскольку программирование на D поддерживает Юникод, то чтобы читать символы Юникода, нам нужно прочитывать дважды и записывать дважды, чтобы получить ожидаемый результат. Это не работает в онлайн-компиляторе. Пример показан ниже.

```
import std.stdio;

void main() {
   char firstCode;
   char secondCode;

   write("Пожалуйста, введите букву: ");
   readf(" %s", &firstCode);
   readf(" %s", &secondCode);

   writeln("Буква, которая была прочитана: ", firstCode, secondCode);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Пожалуйста, введите букву: ğ
Буква, которая была прочитана: ğ
```

Что-то нормально у меня эта программа так и не заработала. Возможно, в Linux было бы больше толку – прим. пер.

Строки

D предоставляет следующие два типа строковых представлений:

•Массив символов

•Тип string

Массив символов

Мы можем представить массив символов в одной из двух форм, как показано ниже. В первой форме размер указан напрямую, а во второй форме используется метод dup, который создаёт перезаписываемую копию строки "Доброе утро".

```
char[21]  greeting1 = "Всем привет";
char[] greeting2 = "Доброе утро".dup;
```

Пример

Вот простой пример, использующий приведённые выше простые формы массива символов.

```
import std.stdio;

void main(string[] args) {
   char[21] greeting1 = "Всем привет";
   writefln("%s", greeting1);

   char[] greeting2 = "Доброе утро".dup;
   writefln("%s", greeting2);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Всем привет
Доброе утро
```

Тип string

Строки string встроены в язык D. Эти строки совместимы с массивом символов, показанным выше. В следующем примере показано простое объявление строки.

```
string greeting1 = "Hello all";
```

Насколько мне известно, это практически то же, что и массивы, только неизменяемые. Тип ***string*** *является псевдонимом для выражения immutable char[] – прим. пер.*

Пример

```
import std.stdio;

void main(string[] args) {
   string greeting1 = "Всем привет";
   writefln("%s", greeting1);

   char[] greeting2 = "Доброе утро".dup;
   writefln("%s", greeting2);
```

```
   string greeting3 = greeting1;
   writefln("%s", greeting3);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Всем привет
Доброе утро
Всем привет
```

Соединение строк (конкатенация)

Для соединения строк в языке D используется символ тильды (~).

Пример

```
import std.stdio;

void main(string[] args) {
   string greeting1 = "Доброе";
   char[] greeting2 = "утро".dup;

   char[] greeting3 = greeting1~" "~greeting2;
   writefln("%s", greeting3);

   string greeting4 = "утро";
   string greeting5 = greeting1~" "~greeting4;
   writefln("%s", greeting5);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Доброе утро
Доброе утро
```

Длина строки

Длину строки в байтах можно получить с помощью функции length.

Пример

```
import std.stdio;

void main(string[] args) {
   string greeting1 = "Доброе";
   writefln("Длина строки greeting1 равна %d", greeting1.length);

   char[] greeting2 = "утро".dup;
   writefln("Длина строки greeting2 равна %d", greeting2.length);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Длина строки greeting1 равна 12
Длина строки greeting2 равна 8
```

Цифры или буквы латинского алфавита обычно занимают по одному байту. Буквы кириллицы в кодировке UTF-8 занимают по 2 байта. Существуют алфавиты, символы которых могут занимать ещё больше байт, например иероглифы некоторых китайских диалектов. – прим. пер.

Сравнение строк

В языке D сравнивать строки довольно просто. Для сравнения строк вы можете использовать операторы ==, <, и >.

Пример

```
import std.stdio;

void main() {
   string s1 = "Привет";
   string s2 = "Мир";
   string s3 = "Мир";

   if (s2 == s3) {
      writeln("s2: ",s2," и s3: ",s3, "  - это одно и тоже!");
   }

   if (s1 < s2) {
      writeln("'", s1, "' предшествует '", s2, "'.");
   } else {
      writeln("'", s2, "' предшествует '", s1, "'.");
   }
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
s2: Мир и s3: Мир  - это одно и тоже!
'Мир' предшествует 'Привет'.
```

Замещение строк

Мы можем заменить часть строки, представленной в виде изменяемого массива, используя операцию среза string[].

Пример

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

void main() {
   char[] s1 = "привет, Земля  ".dup;
   char[] s2 = "космос".dup;

   s1[14..26] = s2[0..12];
   writeln(s1);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
привет, космос
```

Индексные методы

Индексные методы для нахождения подстроки в строке, включают indexOf (поиск первого вхождения) и lastIndexOf (поиск последнего вхождения), и объясняются на следующем примере.

Пример

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

void main() {
   char[] s1 = "привет, Мир ".dup;

   writeln("indexOf подстроки вет внутри привет, Мир равен ", std.string.indexOf(s1,
"вет"));
   writeln(s1);
   writeln("lastIndexOf подстроки И внутри привет, Мир равен " ,
std.string.lastIndexOf(s1, "И", CaseSensitive.no));
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
indexOf подстроки вет внутри привет, Мир равен 6
привет, Мир
lastIndexOf подстроки И внутри привет, Мир равен 16
```

Управление регистрами

Методы, используемые для изменения регистров, показаны в следующем примере.

Пример

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

void main() {
   char[] s1 = "привет, Мир ".dup;
   writeln("Строка s1 с заглавной буквы равна ", capitalize(s1));

   writeln("Строка s1 в верхнем регистре равна ", toUpper(s1));

   writeln("Строка s1 в нижнем регистре равна ", toLower(s1));
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Строка s1 с заглавной буквы равна Привет, мир
Строка s1 в верхнем регистре равна ПРИВЕТ, МИР
Строка s1 в нижнем регистре равна привет, мир
```

Ограничение по символам

Ограничение по символам в строках показано в следующем примере.

Пример

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

void main() {
   string s = "П123Привет1";

   string result = munch(s, "0123456789П");
   writeln("Оставить только начальные символы: ", result);

   result = squeeze(s, "0123456789П");
   writeln("Отсечь начальные символы: ", result);
```

```
    s = "  Привет, Мир  ";
    writeln("Избавиться от начальных и конечных пробелов: ", strip(s));
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Оставить только начальные символы: П123П
Отсечь начальные символы: ривет1
Избавиться от начальных и конечных пробелов: Привет, Мир
```

*К сожалению, **munch**, **squeeze** и ещё несколько не показанных здесь функций из модуля **std.string** в 2017 году объявлены устаревшими, и в 2018 будут удалены из стандартной библиотеки. Вместо них предлагается использовать регулярные выражения. – прим. пер.*

Массивы

Язык D предоставляет структуру данных, называемую **массивом**, которая хранит последовательную коллекцию элементов одинакового типа. Массив используется для хранения коллекции данных. Часто бывает полезно думать о массиве, как о коллекции переменных одинакового типа.

Вместо объявления отдельных переменных, таких как number0, number1, ... и number99, вы объявляете одну переменную массива, такую как numbers, и используете нотацию numbers[0], numbers[1], и ..., numbers[99], которая предоставляет отдельные переменные. К конкретному элементу массива обращение производится по индексу.

Все массивы состоят из смежных областей памяти. Нижний адрес соответствует первому элементу, а высший адрес – последнему элементу.

Объявление массивов

Чтобы на языке D объявить массив, программист указывает тип элементов и их количество, требуемое для массива, следующим образом:

```
тип имя_массива [ размер_массива ];
```

Это называется одномерным массивом. *размер_массива* должен быть целочисленной константой больше нуля, а *тип* может быть любым допустимым типом данных языка D. Например, чтобы объявить 10-элементный массив типа double с именем *balance*, используйте такой оператор:

```
double balance[10];
```

Инициализация массивов

В языке D вы можете инициализировать элементы массива по одному, или сразу все с помощью одного оператора следующим образом:

```
double balance[5] = [1000.0, 2.0, 3.4, 7.0, 50.0];
```

Количество значений между квадратными скобками [ ] с правой стороны не может быть больше числа элементов, которое вы объявляете для размера массива между квадратными скобками [ ] слева.

Если вы опускаете размер массива, будет создан массив, достаточно большой для хранения инициализирующих значений. Поэтому, если вы пишете

```
double balance[] = [1000.0, 2.0, 3.4, 7.0, 50.0];
```

то вы создадите точно такой же массив, как в предыдущем примере.

```
balance[4] = 50.0;
```

Вышеприведенный оператор присваивает элементу массива номер 5 значение 50.0. Массив с индексом 4 будет пятым, т.е. последним элементом, потому что все массивы имеют 0 в качестве индекса своего первого элемента, который также называется базовым индексом. На следующем изображении показан тот же массив, о котором мы говорили выше:

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Доступ к элементам массива

Доступ к элементу осуществляется через индексацию имени массива. Это делается путём размещения индекса элемента в квадратных скобках после имени массива. Например:

```
double salary = balance[9];
```

Вышеприведенный оператор принимает 10-й элемент из массива и присваивает значение переменной *salary*. Следующий пример реализует объявление, присваивание и доступ к массиву:

```
import std.stdio;
void main() {
   int n[ 10 ]; // n - это массив из 10 целых чисел

   // инициализация элементов массива
   for ( int i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; // Установить элемент, расположенный по индексу i, в значение i
+ 100
   }

   writeln("Элемент \t Значение");

   // вывод каждого из значений элементов массива
   for ( int j = 0; j < 10; j++ ) {
      writeln(j, " \t ", n[j]);
   }
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Элемент  Значение
0        100
1        101
2        102
3        103
4        104
5        105
6        106
7        107
8        108
9        109
```

Статические массивы и динамические массивы

Если длина массива задана во время написания программы, такой массив является статическим массивом. Когда длина может меняться во время выполнения программы, такой массив является динамическим массивом.

Определение динамических массивов проще, чем определение массивов фиксированной длины, поскольку при отсутствии длины создаётся динамический массив:

```
int[] dynamicArray;
```

Свойства массивов

Вот список свойств массивов:

№	Свойство и Описание
1	**.init** Статический массив возвращает литерал массива, у которого каждый элемент установлен в значение литерала, являющегося свойством .init типа элемента массива.
2	**.sizeof** Статический массив возвращает длину массива, умноженную на количество байтов в элементе массива, в то время как динамические массивы возвращают размер ссылки на динамический массив, который равен 8 в 32-битных сборках и 16 на 64-битных сборках.
3	**.length** Статический массив возвращает количество элементов в массиве, тогда как в динамических массивах свойство используется для получения/установки количества элементов в массиве. Длина имеет тип *size_t*.
4	**.ptr** Возвращает указатель на первый элемент массива.
5	**.dup** Создаёт динамический массив такого же размера и копирует в него содержимое исходного массива.
6	**.idup** Создаёт динамический массив такого же размера и копирует в него содержимое исходного массива. Тип копии является immutable (неизменяемый).
7	**.reverse** Обращает на месте порядок элементов в массиве. Возвращает массив.
8	**.sort** Сортирует на месте элементы в массиве. Возвращает массив.

Пример

В следующем примере разъясняются различные свойства массива:

```
import std.stdio;

void main() {
   int n[ 5 ]; // n - это массив из 5 целых чисел

   // инициализация элементов массива n
   for ( int i = 0; i < 5; i++ ) {
```

```
      n[ i ] = i + 100; // Установить элемент, расположенный по индексу i, в значение i
+ 100
   }

   writeln("Инициализированное значение:", n.init);

   writeln("Длина: ", n.length);
   writeln("Размер в байтах: ", n.sizeof);
   writeln("Указатель:", n.ptr);

   writeln("Дубликат массива: ", n.dup);
   writeln("Неизменяемый дубликат массива: ", n.idup);

   n = n.reverse.dup;
   writeln("Перевёрнутый массив: ", n);

   writeln("Сортированный массив: ", n.sort);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит результат, похожий на это:

```
Инициализированное значение:[0, 0, 0, 0, 0]
Длина: 5
Размер в байтах: 20
Указатель:7FFF632BDA38
Дубликат массива: [100, 101, 102, 103, 104]
Неизменяемый дубликат массива: [100, 101, 102, 103, 104]
Перевёрнутый массив: [104, 103, 102, 101, 100]
Сортированный массив: [100, 101, 102, 103, 104]
```

Многомерные массивы в D

В D допустимы многомерные массивы. Вот общая форма объявления многомерного массива:

```
тип имя[размер1][размер2]...[размерN];
```

Пример

В следующем объявлении создаётся трехмерный массив целых чисел размером 5x10x4:

```
int threedim[5][10][4];
```

Двумерные массивы в D

Простейшей формой многомерного массива является двумерный массив. Двумерным массивом является, по существу, список одномерных массивов. Чтобы объявить двумерный массив с размером [x, y], вы должны описать его следующим образом:

```
тип имя_массива [ x ][ y ];
```

Где *тип* может быть любым допустимым типом данных языка D, а *имя_массива* должно быть действительным идентификатором языка D.

Двумерный массив можно рассматривать как таблицу, в которой есть x строк и y столбцов. Двумерный массив **a**, содержащий три строки и четыре столбца, показан ниже:

	Столбец 1	Столбец 2	Столбец 3	Столбец 4
Строка 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Строка 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Строка 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Таким образом, каждый элемент в массиве **a** идентифицируется как **a[i][j]**, где **a** – это имя массива, а **i** и **j** – это индексы, которые однозначно идентифицируют каждый элемент в **a**.

Инициализация двумерных массивов

Многомерные массивы можно инициализировать с помощью задания значений в скобках для каждой строки. Следующий массив состоит из 3 строк, и в каждой строке 4 столбца.

```
int a[3][4] = [
   [0, 1, 2, 3] ,   /*  инициализация строки с индексом 0 */
   [4, 5, 6, 7] ,   /*  инициализация строки с индексом 1 */
   [8, 9, 10, 11]   /*  инициализация строки с индексом 2 */
];
```

Вложенные фигурные скобки, которые указывают на предполагаемую строку, являются необязательными. Следующая инициализация эквивалентна предыдущему примеру:

```
int a[3][4] = [0,1,2,3,4,5,6,7,8,9,10,11];
```

Доступ к элементам двумерного массива

Доступ к элементу в двумерном массиве осуществляется через индексы, по индексу строки и индексу столбца массива. Например:

```
int val = a[2][3];
```

Этот оператор принимает четвёртый элемент из третьей строки массива. Вы можете проверить это в приведенной выше таблице.

```
import std.stdio;

void main () {
   // массив с 5-ю строками и 2-мя колонками.
   int a[5][2] = [ [0,0], [1,2], [2,4], [3,6],[4,8]];

   // Выводит значение каждого элемента массива
   for ( int i = 0; i < 5; i++ ) for ( int j = 0; j < 2; j++ ) {
      writeln( "a[" , i , "][" , j , "]: ", a[i][j]);
   }
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
```

```
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

Общие операции с массивами в D

Вот различные операции, выполняемые на массивах:

Срезы массивов

Мы часто используем часть массива, и выделение среза массива бывает весьма полезно. Ниже приведен простой пример выделения среза массива.

```
import std.stdio;

void main () {
   // массив из 5 элементов.
   double a[5] = [1000.0, 2.0, 3.4, 17.0, 50.0];
   double[] b;

   b = a[1..3];
   writeln(b);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
[2, 3.4]
```

*Элемент массива, расположенный по индексу – верхней границе среза, в срез не включается (в отличие от нижней границы). Поэтому срез **a[1..3]** вернул массив из двух элементов. – прим. пер.*

Копирование массива

Мы также применяем копирование массива. Ниже приведен простой пример для копирования массивов.

```
import std.stdio;

void main () {
   // массив из 5 элементов.
   double a[5] = [1000.0, 2.0, 3.4, 17.0, 50.0];
   double b[5];
   writeln("Массив a:",a);
   writeln("Массив b:",b);

   b[] = a;      // 5 элементов из a[5] копируются в b[5]
   writeln("Массив b:",b);

   b[] = a[];   // 5 элементов из a[5] копируются в b[5]
   writeln("Массив b:",b);

   b[1..2] = a[0..1]; // тоже, что b[1] = a[0]
   writeln("Массив b:",b);

   b[0..2] = a[1..3]; // тоже, что b[0] = a[1], b[1] = a[2]
   writeln("Массив b:",b);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Массив a:[1000, 2, 3.4, 17, 50]
Массив b:[nan, nan, nan, nan, nan]
Массив b:[1000, 2, 3.4, 17, 50]
Массив b:[1000, 2, 3.4, 17, 50]
Массив b:[1000, 1000, 3.4, 17, 50]
Массив b:[2, 3.4, 3.4, 17, 50]
```

Присвоение значений массиву

Ниже приведен простой пример для присвоения значения в массиве.

```
import std.stdio;

void main () {
   // массив из 5 элементов.
   double a[5];
   a[] = 5;
   writeln("Массив a:",a);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Массив a:[5, 5, 5, 5, 5]
```

Соединение массивов (конкатенация)

Ниже приведен простой пример соединения двух массивов.

```
import std.stdio;

void main () {
   // массив из 5 элементов.
   double a[5] = 5;
   double b[5] = 10;
   double [] c;
   c = a~b;
   writeln("Массив c: ",c);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Массив c: [5, 5, 5, 5, 5, 10, 10, 10, 10, 10]
```

Результат этой операции лучше присваивать динамическому массиву (как в этом примере), иначе легко можно нарваться на ошибку выхода за границы массива – прим. пер.

Ассоциативные массивы

Ассоциативные массивы имеют индекс, который не обязательно является целым числом, и он может быть разреженным. Индекс для ассоциативного массива называется **ключом** (Key), а его тип называется **KeyType**.

Ассоциативные массивы объявляются путем размещения типа ключа KeyType внутри скобок [] объявления массива. Ниже приведен простой пример для ассоциативного массива.

```
import std.stdio;

void main () {
   int[string] e;     // ассоциативный массив b из целых чисел с ключом строкового типа

   e["test"] = 3;
   writeln(e["test"]);

   string[string] f;  // ассоциативный массив f из строк с ключом строкового типа

   f["test"] = "Tuts";
   writeln(f["test"]);

   writeln(f);

   f.remove("test");
   writeln(f);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
3
Tuts
["test":"Tuts"]
[]
```

Инициализация ассоциативного массива

Ниже показана простая инициализация ассоциативного массива.

```
import std.stdio;

void main () {
   int[string] days =
      [ "Понедельник"   : 0, "Вторник" : 1, "Среда" : 2,
        "Четверг" : 3, "Пятница"  : 4, "Суббота"  : 5,
        "Воскресение"   : 6 ];
   writeln(days["Вторник"]);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
1
```

Свойства ассоциативного массива

Вот свойства ассоциативного массива:

№	Свойство и Описание
1	**.sizeof** Возвращает размер ссылки на ассоциативный массив; Это 4 в 32-битных сборках и 8 в 64-битных сборках.
2	**.length** Возвращает количество значений в ассоциативном массиве. В отличие от динамических массивов, это свойство доступно только для чтения.
3	**.dup** Создаёт новый ассоциативный массив того же размера и копирует в него содержимое ассоциативного массива.
4	**.keys** Возвращает динамический массив, элементами которого являются ключи ассоциативного массива.
5	**.values** Возвращает динамический массив, элементами которого являются значения ассоциативного массива.
6	**.rehash** Реорганизует ассоциативный массив на месте, чтобы поиск в нём был более эффективным. rehash эффективен, когда, например, при выполнении программы загружается таблица символов, и теперь в ней требуется быстрый поиск. Возвращает ссылку на реорганизованный массив.
7	**.byKey()** Возвращает делегат, подходящий для использования в качестве агрегата для оператора foreach, который будет перебирать ключи из ассоциативного массива.
8	**.byValue()** Возвращает делегат, подходящий для использования в качестве агрегата для оператора foreach, который будет перебирать значения из ассоциативного массива.
9	**.get(Key key, lazy Value defVal)** Ищет ключ; Если он существует, возвращает соответствующее значение, в противном случае вычисляет и возвращает defVal.
10	**.remove(Key key)** Удаляет объект по ключу.

Пример

Ниже приведён пример использования указанных свойств.

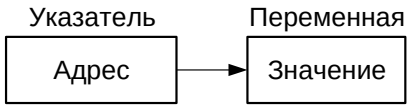
```
import std.stdio;

void main () {
   int[string] array1;

   array1["тест"] = 3;
   array1["тест2"] = 20;

   writeln("Размер в байтах: ", array1.sizeof);
   writeln("длина: ", array1.length);
   writeln("дубликат: ", array1.dup);
   array1.rehash;

   writeln("rehashed: ", array1);
   writeln("ключи: ", array1.keys);
   writeln("значения: ", array1.values);

   foreach (key; array1.byKey) {
      writeln("по ключу: ", key);
   }

   foreach (value; array1.byValue) {
      writeln("по значению ", value);
   }

   writeln("получить значение для ключа тест: ", array1.get("тест", 10));
   writeln("получить значение для ключа тест3: ", array1.get("тест3", 10));
   array1.remove("тест");
   writeln(array1);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Размер в байтах: 8
длина: 2
дубликат: ["тест":3, "тест2":20]
rehashed: ["тест2":20, "тест":3]
ключи: ["тест2", "тест"]
значения: [20, 3]
по ключу: тест2
по ключу: тест
по значению 20
по значению 3
получить значение для ключа тест: 3
получить значение для ключа тест3: 10
["тест2":20]
```

Указатели

В языке D указатели изучать легко и интересно. Некоторые задачи программирования выполняются проще с помощью указателей, а другие задачи, такие как распределение динамической памяти, не могут выполняться без них. Ниже показан простой указатель.



Вместо прямого указания на переменную, указатель указывает на адрес переменной. Как вы знаете, каждая переменная является местом в памяти, и каждая ячейка памяти имеет свой адрес, который можно получить, используя оператор ampersand (&), который возвращает адрес в памяти. Рассмотрим следующую программу, которая печатает адрес объявленных переменных:

```
import std.stdio;

void main () {
   int var1;
   writeln("Адрес переменной var1: ",&var1);

   char var2[10];
   writeln("Адрес переменной var2: ",&var2);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит результат, похожий на это:

```
Адрес переменной var1: 7FFE686BBB28
Адрес переменной var2: 7FFE686BBB30
```

Что такое указатели?

Указатель (pointer) – это переменная, значением которой является адрес другой переменной. Как и любую другую переменную или константу, вы должны объявить указатель, прежде чем сможете с ним работать. Общая форма объявления переменной указателя:

```
тип *имя_указателя;
```

Здесь **тип** – это базовый тип, на который будет указывать указатель; Он должен быть допустимым типом программирования, а **имя_указателя** – это имя переменной указателя. Звездочкой, которую вы использовали для объявления указателя, является та же самая звездочка, которую вы используете для умножения. Однако, в этом выражении звездочка используется для обозначения переменной как указателя. Ниже приведены допустимые объявления указателей:

```
int    *ip;    // указатель на integer
double *dp;    // указатель на double
float  *fp;    // указатель на float
char   *ch     // указатель на символ
```

Фактический тип данных у всех указателей, указывают ли они на integer, float, char или на что-то другое – один и тот же, длинное шестнадцатеричное число, которое представляет

собой адрес в памяти. Единственное различие между указателями разных типов данных – это тип данных переменной или константы, на которую указывает указатель.

Использование указателей в программировании на D

Существует несколько важных операций, которые мы часто используем с указателями.

- Определить переменные указателя
- Назначить адрес переменной указателю
- Наконец, получить доступ к значению по адресу, доступному из переменной указателя.

Это делается с помощью унарного оператора *****, который возвращает значение переменной, расположенной по адресу, указанному его операндом. В следующем примере используются эти операции:

```
import std.stdio;

void main () {
   int var = 20;   // фактическое объявление переменной.
   int *ip;        // переменная указателя
   ip = &var;   // сохранить адрес переменной var в переменной указателя

   writeln("Значение переменной var: ", var);

   writeln("Адрес, сохранённый в переменной ip: ", ip);

   writeln("Значение переменной *ip: ", *ip);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит результат, похожий на это:

```
Значение переменной var: 20
Адрес, сохранённый в переменной ip: 7FFEBEA77270
Значение переменной *ip: 20
```

Нулевой указатель

Всегда полезно назначить указатель **null** переменной указателя, если у вас нет точного адреса, который нужно назначить. Это делается во время объявления переменной. Указатель, которому присвоен **null**, называется нулевым указателем.

Указатель null – это неявно определённая константа с нулевым значением. Рассмотрим следующую программу:

```
import std.stdio;

void main () {
   int  *ptr = null;
   writeln("Значение ptr равно " , ptr) ;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Значение ptr равно null
```

В большинстве операционных систем программам не разрешается обращаться к памяти по адресу 0, поскольку эта память зарезервирована операционной системой. Однако, нулевой адрес памяти имеет особое значение; он означает, что указателю не назначена никакая доступная ячейка памяти.

По соглашению, если указатель содержит значение null (ноль), предполагается, что он ни на что не указывает. Чтобы проверить указатель, содержится ли в нём null, вы можете использовать оператор if следующим образом:

```
if(ptr)     // успешно, если p не равен null
if(!ptr)    // успешно, если p равен null
```

Таким образом, если для всех неиспользуемых указателей задано значение null, и вы избегаете использования нулевого указателя, вы можете избежать случайного ошибочного применения неинициализированного указателя. Довольно часто неинициализированные переменные могут содержать какие-то мусорные значения, и это затрудняет отладку программы.

Арифметика указателей

Существует четыре арифметических оператора, которые могут применяться для указателей: ++, --, + и -

Чтобы понять арифметику указателей, рассмотрим указатель на целое с именем **ptr**, который указывает на адрес 1000. Предполагая, что целые числа являются 32-битными, давайте выполним следующую арифметическую операцию над указателем:

```
ptr++
```

теперь **ptr** будет указывать на адрес 1004, потому что каждый раз, когда ptr увеличивается, он указывает на следующее целое число. Эта операция перемещает указатель к следующей ячейке памяти, не влияя на фактическое значение в ячейке памяти.

Если **ptr** указывает на символ (тип char), адрес которого равен 1000, то приведённая выше операция перенаправит его к адресу 1001, потому что следующий символ будет доступен в ячейке 1001.

Увеличение указателя

Мы предпочитаем использовать указатель в нашей программе вместо массива, потому что переменную указателя можно увеличить, в отличие от имени массива, которое увеличить невозможно, поскольку оно является постоянным указателем. Следующая программа увеличивает указатель переменной для доступа к каждому последующему элементу массива:

```
import std.stdio;

const int MAX = 3;

void main () {
   int var[MAX] = [10, 100, 200];
   int *ptr = &var[0];

   for (int i = 0; i < MAX; i++, ptr++) {
      writeln("Адрес var[" , i , "] = ", ptr);
      writeln("Значение var[" , i , "] = ", *ptr);
```

```
   }
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит результат, похожий на это:

```
Адрес var[0] = 7FFD2870CCD0
Значение var[0] = 10
Адрес var[1] = 7FFD2870CCD4
Значение var[1] = 100
Адрес var[2] = 7FFD2870CCD8
Значение var[2] = 200
```

Указатели и массивы

Указатели и массивы сильно связаны. Однако они не являются полностью взаимозаменяемыми. Например, рассмотрим следующую программу:

```
import std.stdio;

const int MAX = 3;

void main () {
   int var[MAX] = [10, 100, 200];
   int *ptr = &var[0];
   var.ptr[2]  = 290;
   ptr[0] = 220;

   for (int i = 0; i < MAX; i++, ptr++) {
      writeln("Адрес var[" , i , "] = ", ptr);
      writeln("Значение var[" , i , "] = ", *ptr);
   }
}
```

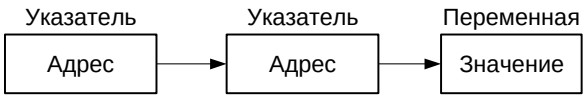
В приведенной выше программе вы можете увидеть выражение var.ptr[2], используемое для присвоения значения второму элементу, и ptr[0], которое используется для присвоения нулевому элементу. Оператор приращения может использоваться с ptr, но не с var.

Когда вы скомпилируете и выполните эту программу, она возвратит результат, похожий на это:

```
Адрес var[0] = 7FFC03A8AEA0
Значение var[0] = 220
Адрес var[1] = 7FFC03A8AEA4
Значение var[1] = 100
Адрес var[2] = 7FFC03A8AEA8
Значение var[2] = 290
```

Указатель на указатель

Указатель на указатель представляет собой форму многократной косвенности или цепочки указателей. Обычно указатель содержит адрес переменной. Когда мы определяем указатель на указатель, первый указатель содержит адрес второго указателя, который, в свою очередь, указывает на местоположение, содержащее фактическое значение, как показано ниже.



Переменная, являющаяся указателем на указатель, должна быть объявлена соответствующим образом. Это делается путём размещения дополнительной звездочки перед её именем. Например, вот так выглядит синтаксис объявления указателя на указатель на тип int:

```
int **var;
```

Когда к целевому значению применяется косвенный доступ через указатель на указатель, тогда для доступа к этому значению требуется, чтобы оператор звёздочки применялся дважды, как показано ниже в примере:

```
import std.stdio;

const int MAX = 3;

void main () {
   int var = 3000;
   writeln("Значение переменной var : ", var);

   int *ptr = &var;
   writeln("Значение, доступное как *ptr : ", *ptr);

   int **pptr = &ptr;
   writeln("Значение, доступное как **pptr : ", **pptr);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Значение переменной var : 3000
Значение, доступное как *ptr : 3000
Значение, доступное как **pptr : 3000
```

Передача указателя в функцию

D позволяет вам передать указатель в функцию. Для этого просто объявите параметр функции с типом указателя.

В следующем простом примере указатель передаётся в функцию.

```
import std.stdio;

void main () {
   // массив из пяти элементов целого типа.
   int balance[5] = [1000, 2, 3, 17, 50];
   double avg;

   avg = getAverage( &balance[0], 5 ) ;
   writeln("Среднее арифметическое равно : " , avg);
}

double getAverage(int *arr, int size) {
   int    i;
   double avg, sum = 0;

   for (i = 0; i < size; ++i) {
      sum += arr[i];
   }

   avg = sum/size;
   return avg;
```

```
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Среднее арифметическое равно : 214.4
```

Возврат указателя из функции

Рассмотрим следующую функцию, которая возвращает 10 чисел с помощью указателя, через который передаётся адрес первого элемента массива.

```
import std.stdio;

void main () {
   int *p = getNumber();

   for ( int i = 0; i < 10; i++ ) {
      writeln("*(p + " , i , ") : ", *(p + i));
   }
}

int * getNumber( ) {
   static int r [10];

   for (int i = 0; i < 10; ++i) {
      r[i] = i;
   }

   return &r[0];
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
*(p + 0) : 0
*(p + 1) : 1
*(p + 2) : 2
*(p + 3) : 3
*(p + 4) : 4
*(p + 5) : 5
*(p + 6) : 6
*(p + 7) : 7
*(p + 8) : 8
*(p + 9) : 9
```

*Я тут опять со своими нудными замечаниями. В этом примере важно "волшебное" ключевое слово **static** в объявлении массива внутри функции. Если его пропустить, то компилятор откажется выводить наружу ссылку через указатель на внутреннюю переменную **r**, потому что при выходе из функции она перестанет существовать. Существует несколько способов обойти эту проблему, и использование **static** – самый простой из них, но он имеет побочный эффект – массив **r** становится чем-то вроде глобальной переменной – прим. пер.*

Указатель на массив

Имя массива является постоянным указателем на первый элемент массива. Поэтому в объявлении:

```
double balance[50];
```

balance – это указатель на &balance[0], т.е. на адрес первого элемента массива balance. Таким образом, следующий фрагмент программы присваивает указателю **p** адрес первого элемента массива **balance**:

```
double *p;
double balance[10];

p = balance;
```

Легально использовать имена массивов в качестве постоянных указателей и наоборот. Поэтому *(balance + 4) является законным способом доступа к значению balance[4].

Когда вы храните адрес первого элемента в p, вы можете обращаться к элементам массива, используя *p, *(p+1), *(p+2) и так далее. В следующем примере показаны все концепции, рассмотренные выше:

```
import std.stdio;

void main () {
   // массив из 5 элементов.
   double balance[5] = [1000.0, 2.0, 3.4, 17.0, 50.0];
   double *p;

   p = &balance[0];

   // Выводит значение каждого элемента массива
   writeln("Значения элементов массива с использованием указателя " );

   for ( int i = 0; i < 5; i++ ) {
      writeln( "*(p + ", i, ") : ", *(p + i));
   }
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Значения элементов массива с использованием указателя
*(p + 0) : 1000
*(p + 1) : 2
*(p + 2) : 3.4
*(p + 3) : 17
*(p + 4) : 50
```

При таком использовании указателей необходимо вручную следить за индексами и не выйти за границы массива. Для эксперимента я увеличил верхнюю границу цикла for в последнем примере, никаких ошибок (ни от компилятора, ни времени выполнения) не последовало, и программа послушно выдала мне "мусорные" данные, расположенные в памяти после массива. При использовании стандартной нотации массивов, например, balance[6], система будет сама отслеживать недопустимые индексы – прим.пер.

Кортежи

Кортежи используются для объединения нескольких значений в виде одного объекта. Кортежи содержат последовательность элементов. Элементы могут быть типами, выражениями или псевдонимами. Элементы кортежа и их количество фиксируются во время компиляции, и их нельзя изменить во время выполнения.

Кортежи имеют свойства, похожие на свойства как структур, так и массивов. Элементы кортежа могут быть разных типов, так же, как в структуре. Доступ к элементам можно получить посредством индексирования, как в массиве. Они реализованы в стандартной библиотеке с помощью шаблона Tuple из модуля std.typecons. Tuple использует TypeTuple из модуля std.typetuple для некоторых своих операций. (*Сейчас модуль std.typetuple практически пуст, только публично импортирует модуль std.meta, тип TypeTuple в нём есть, но сейчас является псевдонимом типа AliasSeq, вроде бы их переименовали для уменьшения путаницы – прим. пер.*)

Создание кортежа с помощью tuple()

Кортежи можно создавать с помощью функции tuple(). Обращение к элементам кортежа происходит по значению индекса. Пример показан ниже.

Пример

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

void main() {
   auto myTuple = tuple(1, "Tuts");
   writeln(myTuple);
   writeln(myTuple[0]);
   writeln(myTuple[1]);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Tuple!(int, string)(1, "Tuts")
1
Tuts
```

Создание кортежа с помощью шаблона Tuple

Кортеж также можно создавать непосредственно через шаблон Tuple вместо функции tuple(). Тип и имя каждого элемента указываются в виде двух последовательных параметров шаблона. При создании кортежа с помощью шаблона можно получить доступ к элементам через свойства.

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

void main() {
   auto myTuple = Tuple!(int, "id",string, "value")(1, "Tuts");
   writeln(myTuple);
```

```
   writeln("по индексу 0 : ", myTuple[0]);
   writeln("через свойство .id : ", myTuple.id);

   writeln("по индексу 1 : ", myTuple[1]);
   writeln("через свойство .value ", myTuple.value);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Tuple!(int, "id", string, "value")(1, "Tuts")
по индексу 0 : 1
через свойство .id : 1
по индексу 1 : Tuts
через свойство .value Tuts
```

Свойство .expand и параметры функций

Члены кортежа можно раскрыть либо с помощью свойства .expand, либо через операцию среза. Это раскрытое/нарезанное значение можно передавать в качестве списка аргументов функции. Пример показан ниже.

Пример

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

void method1(int a, string b, float c, char d) {
   writeln("method1 ",a,"\t",b,"\t",c,"\t",d);
}

void method2(int a, float b, char c) {
   writeln("method2 ",a,"\t",b,"\t",c);
}

void main() {
   auto myTuple = tuple(5, "моя строка", 3.3, 'r');

   writeln("method1 вызов 1");
   method1(myTuple[]);

   writeln("method1 вызов 2");
   method1(myTuple.expand);

   writeln("method2 вызов 1");
   method2(myTuple[0], myTuple[$-2..$]);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
method1 вызов 1
method1 5       моя строка      3.3     r
method1 вызов 2
method1 5       моя строка      3.3     r
method2 вызов 1
method2 5       3.3     r
```

TypeTuple

Кортеж TypeTuple определяется в модуле std.typetuple. Список значений и типов, разделенных запятыми. Ниже приведен простой пример использования TypeTuple. TypeTuple

используется для создания списка аргументов, шаблонного списка, и списка литералов массива.

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

alias TypeTuple!(int, long) TL;

void method1(int a, string b, float c, char d) {
   writeln("method1 ",a,"\t",b,"\t",c,"\t",d);
}

void method2(TL tl) {
   writeln(tl[0],"\t", tl[1] );
}

void main() {
   auto arguments = TypeTuple!(5, "моя строка", 3.3,'r');
   method1(arguments);
   method2(5, 6L);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
method1 5       моя строка      3.3     r
5       6
```

Структуры

Структура представляет собой ещё один пользовательский тип данных, доступный в программировании на языке D, который позволяет комбинировать элементы данных разных типов.

Структуры используются для представления записей. Предположим, вы хотите отслеживать свои книги в библиотеке. Вы можете отслеживать следующие атрибуты каждой книги:

- Заголовок – Title
- Автор – Author
- Тема – Subject
- Идентификатор книги – Book ID

Определение структуры

Чтобы определить структуру, вы должны использовать оператор struct. Оператор struct определяет для вашей программы новый тип данных с более чем одним членом. Формат выражения struct такой:

```
struct [тег структуры] {
   объявление члена;
   объявление члена;
   ...
   объявление члена;
} [Одна или несколько переменных типа структуры];
```

Тег структуры является необязательным, а объявление каждого члена является обычным объявлением переменной, таким как int i; или float f;, или любое другое допустимое определение переменной. В конце определения структуры до точки с запятой вы можете указать одну или несколько переменных типа этой структуры, которые являются необязательными. Вот как вы бы объявили структуру Books:

```
struct Books {
   char [] title;
   char [] author;
   char [] subject;
   int   book_id;
};
```

Доступ к членам структуры

Чтобы получить доступ к любому члену структуры, вы используете **оператор доступа к члену (.)**. Оператор доступа в коде выглядит как точка между именем переменной структуры и членом структуры, к которому мы хотим получить доступ. Вы можете использовать ключевое слово **struct** для определения переменных типа структуры. В следующем примере объясняется использование структуры:

```
import std.stdio;

struct Books {
```

```
   char [] title;
   char [] author;
   char [] subject;
   int   book_id;
};

void main( ) {
   Books Book1;        /* Объявление Book1 типа Book */
   Books Book2;        /* Объявление Book2 типа Book */

   /* спецификация книги 1 */
   Book1.title = "D Programming".dup;
   Book1.author = "Raj".dup;
   Book1.subject = "D Programming Tutorial".dup;
   Book1.book_id = 6495407;

   /* спецификация книги 2 */
   Book2.title = "D Programming".dup;
   Book2.author = "Raj".dup;
   Book2.subject = "D Programming Tutorial".dup;
   Book2.book_id = 6495700;

   /* вывести информацию о Book1 */
   writeln( "Заголовок Book1 : ", Book1.title);
   writeln( "Автор Book1 : ", Book1.author);
   writeln( "Тема Book1 : ", Book1.subject);
   writeln( "Идентификатор Book1 : ", Book1.book_id);

   /* вывести информацию о Book2 */
   writeln( "Заголовок Book2 : ", Book2.title);
   writeln( "Автор Book2 : ", Book2.author);
   writeln( "Тема Book2 : ", Book2.subject);
   writeln( "Идентификатор Book2 : ", Book2.book_id);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Заголовок Book1 : D Programming
Автор Book1 : Raj
Тема Book1 : D Programming Tutorial
Идентификатор Book1 : 6495407
Заголовок Book2 : D Programming
Автор Book2 : Raj
Тема Book2 : D Programming Tutorial
Идентификатор Book2 : 6495700
```

Структуры как аргументы функции

Вы можете передавать структуру как аргумент функции практически так же, как вы передаете любую другую переменную или указатель. Вы можете получать доступ к переменным структуры таким же образом, как вы получали доступ в приведённом выше примере:

```
import std.stdio;

struct Books {
   char [] title;
   char [] author;
   char [] subject;
   int   book_id;
};
```

```
void main( ) {
   Books Book1;        /* Объявление Book1 типа Book */
   Books Book2;        /* Объявление Book2 типа Book */

   /* спецификация книги 1 */
   Book1.title = "D Programming".dup;
   Book1.author = "Raj".dup;
   Book1.subject = "D Programming Tutorial".dup;
   Book1.book_id = 6495407;

   /* спецификация книги 2 */
   Book2.title = "D Programming".dup;
   Book2.author = "Raj".dup;
   Book2.subject = "D Programming Tutorial".dup;
   Book2.book_id = 6495700;

   /* вывести информацию о Book1 */
   printBook( Book1 );

   /* вывести информацию о Book2 */
   printBook( Book2 );
}

void printBook( Books book ) {
   writeln( "Заголовок книги : ", book.title);
   writeln( "Автор книги : ", book.author);
   writeln( "Тема книги : ", book.subject);
   writeln( "Идентификатор книги : ", book.book_id);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Заголовок книги : D Programming
Автор книги : Raj
Тема книги : D Programming Tutorial
Идентификатор книги : 6495407
Заголовок книги : D Programming
Автор книги : Raj
Тема книги : D Programming Tutorial
Идентификатор книги : 6495700
```

Инициализация структуры

Структуры можно инициализировать в двух формах: в одной используется конструктор, а другая с использованием формата {}. Пример показан ниже.

Пример

```
import std.stdio;

struct Books {
   char [] title;
   char [] subject = "Empty".dup;
   int   book_id = -1;
   char [] author = "Raj".dup;
};

void main( ) {
   Books Book1 = Books("D Programming".dup, "D Programming Tutorial".dup, 6495407 );
   printBook( Book1 );
```

```
   Books Book2 = Books("D Programming".dup,
      "D Programming Tutorial".dup, 6495407,"Raj".dup );
   printBook( Book2 );

   Books Book3 =  {title:"Obj C programming".dup, book_id : 1001};
   printBook( Book3 );
}

void printBook( Books book ) {
   writeln( "Название книги : ", book.title);
   writeln( "Автор книги : ", book.author);
   writeln( "Тема книги : ", book.subject);
   writeln( "Идентификатор книги : ", book.book_id);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Название книги : D Programming
Автор книги : Raj
Тема книги : D Programming Tutorial
Идентификатор книги : 6495407
Название книги : D Programming
Автор книги : Raj
Тема книги : D Programming Tutorial
Идентификатор книги : 6495407
Название книги : Obj C programming
Автор книги : Raj
Тема книги : Empty
Идентификатор книги : 1001
```

Статические члены

Статические переменные инициализируются только один раз. Например, чтобы иметь уникальные идентификаторы для книг, мы можем сделать id статическим, увеличивать эту переменную и присваивать её в поле book_id. Пример показан ниже.

Пример

```
import std.stdio;

struct Books {
   char [] title;
   char [] subject = "Empty".dup;
   int   book_id;
   char [] author = "Raj".dup;
   static int id = 1000;
};

void main( ) {
   Books Book1 = Books("D Programming".dup, "D Programming Tutorial".dup, ++Books.id );
   printBook( Book1 );

   Books Book2 = Books("D Programming".dup, "D Programming Tutorial".dup, ++Books.id);
   printBook( Book2 );

   Books Book3 =  {title: "Obj C programming".dup, book_id: ++Books.id};
   printBook( Book3 );
}

void printBook( Books book ) {
```

```
   writeln( "Название книги : ", book.title);
   writeln( "Автор книги : ", book.author);
   writeln( "Тема книги : ", book.subject);
   writeln( "Идентификатор книги : ", book.book_id);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Название книги : D Programming
Автор книги : Raj
Тема книги : D Programming Tutorial
Идентификатор книги : 1001
Название книги : D Programming
Автор книги : Raj
Тема книги : D Programming Tutorial
Идентификатор книги : 1002
Название книги : Obj C programming
Автор книги : Raj
Тема книги : Empty
Идентификатор книги : 1003
```

К сожалению, авторы этого учебника не упомянули о возможности объявлять методы (функции-члены) в структуре. Таким образом, про них рассказано гораздо меньше, чем можно было бы (например, о конструкторах, о перегрузке операций и т.д.). Фактически структуры в языке D имеют большинство свойств класса (см. раздел ООП), за исключением следующих отличий:

- *Объекты структур обычно создаются в стеке, как обычные переменные, хотя создавать их в куче через оператор **new** тоже допустимо.*
- *В отличие от классов, структуры не поддерживают наследование.*

Возможно, я ещё чего-то не знаю, и моё описание тоже не полное – прим. пер.

Объединения

Объединение – это специальный тип данных, доступный в D, который позволяет хранить различные типы данных в одном и том же месте памяти. Вы можете определить объединение со многими членами, но только один из них может содержать значение в определённый момент времени. Объединения предоставляют эффективный способ использования одного и того же места памяти для нескольких целей.

Определение объединения в D

Чтобы определить объединение, вы должны использовать оператор **union** таким же образом, как и при определении структуры. Оператор union определяет для вашей программы новый тип данных с более чем одним членом. Формат оператора union выглядит следующим образом:

```
union [тег объединения] {
   объявление члена;
   объявление члена;
   ...
   объявление члена;
} [Одна или несколько переменных типа union];
```

Тег объединения является необязательным, а объявление каждого члена является обычным объявлением переменной, таким как int i; или float f;, или любое другое допустимое определение переменной. В конце определения объединения до точки с запятой вы можете указать одну или несколько переменных типа этого объединения, которые являются необязательными. Вот как можно определить тип объединения с именем Data, который имеет три члена **i**, **f** и **str**:

```
union Data {
   int i;
   float f;
   char str[20];
} data;
```

Переменная типа **Data** может хранить целое число, число с плавающей точкой или массив символов. Это означает, что для хранения нескольких типов данных может использоваться одна и та же ячейка памяти. Вы можете использовать любые встроенные или определённые пользователем типы данных внутри объединения по своим потребностям.

Память, занимаемая объединением, будет достаточно большой, чтобы хранить наибольший член объединения. В приведенном выше примере тип данных будет занимать 20 байтов пространства памяти, поскольку это максимальное пространство, которое может быть занято массивом символов. В следующем примере отображается суммарный объём памяти, занимаемый указанным объединением:

```
import std.stdio;

union Data {
   int i;
   float f;
   char str[20];
};
```

```
int main( ) {
   Data data;

   writeln( "Объём памяти, занимаемый данными : ", data.sizeof);

   return 0;
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Объём памяти, занимаемый данными : 20
```

Доступ к членам объединения

Чтобы получить доступ к любому члену объединения, вы используете **оператор доступа к члену (.)**. Оператор доступа в коде выглядит как точка между именем переменной объединения и членом объединения, к которому мы хотим получить доступ.

Пример

В следующем примере разъясняется использование объединения:

```
import std.stdio;

union Data {
   int i;
   float f;
   char str[13];
};

void main( ) {
   Data data;

   data.i = 10;
   data.f = 220.5;

   data.str = "D Programming".dup;
   writeln( "размер data : ", data.sizeof);
   writeln( "data.i : ", data.i);
   writeln( "data.f : ", data.f);
   writeln( "data.str : ", data.str);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
размер data : 16
data.i : 1917853764
data.f : 4.12236e+30
data.str : D Programming
```

Здесь вы можете увидеть, что значения членов объединения **i** и **f** были повреждены, потому что место в памяти заняло последнее присвоенное переменной значение, и именно по этой причине значение члена **str** отобразилось хорошо.

Теперь давайте снова рассмотрим тот же пример, в котором мы теперь будем использовать одну переменную за один раз, что является основной целью использования объединений:

ВИДОИЗМЕНЕННЫЙ Пример

```
import std.stdio;

union Data {
   int i;
   float f;
   char str[13];
};
void main( ) {
   Data data;
   writeln( "размер data : ", data.sizeof);

   data.i = 10;
   writeln( "data.i : ", data.i);

   data.f = 220.5;
   writeln( "data.f : ", data.f);

   data.str = "D Programming".dup;
   writeln( "data.str : ", data.str);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
размер data : 16
data.i : 10
data.f : 220.5
data.str : D Programming
```

Здесь значения всех членов отобразились хорошо, потому что одновременно используется только один из них.

Диапазоны

Диапазоны – это абстракция доступа к элементу. Эта абстракция позволяет использовать большое количество алгоритмов для большого количества типов контейнеров. Диапазоны подчеркивают то, как происходит доступ к элементам контейнера, в отличие от того, как реализуются контейнеры. Диапазоны – это очень простая концепция, основанная на том, определяет ли данный тип некоторый набор методов (функций-членов).

Диапазоны являются неотъемлемой частью языка D. Срезы в D являются реализациями самого мощного диапазона – диапазона с произвольным доступом, и множество возможностей в Phobos связано с диапазонами. Многие алгоритмы в Phobos возвращают объекты с временным диапазоном. Например, функция filter(), выбирающая элементы, отвечающие некоторому критерию (предикату), фактически возвращает объект диапазона, а не массив.

Диапазоны чисел

Диапазоны чисел используются довольно часто, это диапазоны чисел типа int. Несколько примеров для диапазонов чисел показаны ниже:

```
// Пример 1
foreach (value; 3..7)

// Пример 2
int[] slice = array[5..10];
```

Диапазоны в Phobos

Диапазоны, относящиеся к структурам и интерфейсам классов, представляют диапазоны в Phobos. Phobos – официальная среда исполнения и стандартная библиотека, которая поставляется вместе с компилятором языка D.

Существуют различные типы диапазонов, которые включают в себя следующие:

- Входной диапазон (InputRange)
- Лидирующий диапазон (ForwardRange)
- Двунаправленный диапазон (BidirectionalRange)
- Диапазон с произвольным доступом (RandomAccessRange)
- Выходной диапазон (OutputRange)

Входной диапазон

Самый простой диапазон – это входной диапазон. Другие диапазоны предъявляют больше требований к той реализации, на которой они основаны. Вот три функции, требуемые входным диапазоном:

- **empty** – Она информирует, пуст ли диапазон; она должна возвращать значение true, когда диапазон считается пустым; false в противном случае.

•**front** – Она предоставляет доступ к элементу в начале диапазона.

•**popFront()** – Он сокращает диапазон от начала, удаляя первый элемент.

Пример

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

struct Student {
   string name;
   int number;

   string toString() const {
      return format("%s(%s)", name, number);
   }
}

struct School {
   Student[] students;
}
struct StudentRange {
   Student[] students;

   this(School school) {
      this.students = school.students;
   }
   @property bool empty() const {
      return students.length == 0;
   }
   @property ref Student front() {
      return students[0];
   }
   void popFront() {
      students = students[1 .. $];
   }
}

void main() {
   auto school = School([ Student("Иванов", 1), Student("Петров", 2),
Student("Сидоров", 3)]);
   auto range = StudentRange(school);
   writeln(range);

   writeln(school.students.length);

   writeln(range.front);

   range.popFront;

   writeln(range.empty);
   writeln(range);
}
```

Думаю, этот кусок кода требует некоторого пояснения.

•*Атрибут* ***@property*** *у метода позволяет вызывать этот метод без скобок, таким образом методы front() и empty() становятся "псевдочленами", и их можно вызывать без скобок: range.front; range.empty;*

– прим. пер.

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
[Иванов(1), Петров(2), Сидоров(3)]
3
Иванов(1)
false
[Петров(2), Сидоров(3)]
```

Лидирующий диапазон

Лидирующий диапазон требует наличия метода **save** в дополнение к остальным трём функциям Входного диапазона, и при вызове этот метод должен возвращать копию диапазона.

```
import std.array;
import std.stdio;
import std.string;
import std.range;

struct FibonacciSeries {
   int first = 0;
   int second = 1;
   enum empty = false;   //  бесконечный диапазон

   @property int front() const {
      return first;
   }
   void popFront() {
      int third = first + second;
      first = second;
      second = third;
   }
   @property FibonacciSeries save() const {
      return this;
   }
}

void report(T)(const dchar[] title, const ref T range) {
   writefln("%s: %s", title, range.take(5));
}

void main() {
   auto range = FibonacciSeries();
   report("Исходный диапазон", range);

   range.popFrontN(2);
   report("После удаления двух элементов", range);

   auto theCopy = range.save;
   report("Копия", theCopy);

   range.popFrontN(3);
   report("После удаления еще трех элементов", range);
   report("Копия", theCopy);
}
```

Продолжу свою нудятину:

Многие функции, работающие с диапазонами и несколько готовых диапазонов определены в модуле стандартной библиотеки ***std.range****.*

•*Функция **take** определена в модуле **std.range**, она принимает в качестве параметров диапазон и целое число, и выдаёт диапазон с соответствующим количеством элементов исходного диапазона.*

•*Функция **popFrontN** определена в модуле **std.range.primitives**, она принимает в качестве параметров диапазон и целое число, и вызывает у диапазона соответствующее количество раз метод popFront(). В диапазонной терминологии это называется "продвинуться" (advances) по диапазону.*

•*Вызов обеих этих функций происходит с помощью так называемой "нотации псевдочленов" языка D (в английских источниках это называется UFCS, Uniform Function Call Syntax)), т.е. если компилятор встречает запись **a.fun(b, c)** , а у объекта **a** нет метода **fun**, то компилятор попробует применить функцию fun с первым аргументом a: **fun(a, b, c)**.*

– прим. пер.

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Исходный диапазон: [0, 1, 1, 2, 3]
После удаления двух элементов: [1, 2, 3, 5, 8]
Копия: [1, 2, 3, 5, 8]
После удаления еще трех элементов: [5, 8, 13, 21, 34]
Копия: [1, 2, 3, 5, 8]
```

Двунаправленный диапазон

Двунаправленный диапазон предоставляет два метода дополнительно к методам Лидирующего диапазона. Функция **back**, аналогичная front, обеспечивает доступ к последнему элементу диапазона. Функция **popBack** аналогична функции popFront и удаляет последний элемент из диапазона.

Пример

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

struct Reversed {
   int[] range;

   this(int[] range) {
      this.range = range;
   }
   @property bool empty() const {
      return range.empty;
   }
   @property int front() const {
      return range.back;  //  наоборот
   }
   @property int back() const {
      return range.front; // наоборот
   }
   void popFront() {
      range.popBack();
   }
```

```
    void popBack() {
        range.popFront();
    }
}

void main() {
    writeln(Reversed([ 1, 2, 3]));
}
```

И тут, похоже, нужны пояснения:

- *Модуль стандартной библиотеки **std.range.primitives** объявляет функции двунаправленного диапазона (save, empty, popFront, popBack, front, back) для встроенных массивов. Модуль **std.array** их публично импортирует. Поэтому импортирования любого из этих модулей достаточно, чтобы массивы стали полноценными двунаправленными диапазонами.*

– прим. пер.

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
[3, 2, 1]
```

Бесконечный диапазон с произвольным доступом

Дополнительно к методам Лидирующего диапазона требуется метод **opIndex()**. Кроме того, во время компиляции должно быть известно, что значение функции empty всегда false. Простой пример, показывающий диапазон квадратов чисел показан ниже.

```
import std.array;
import std.stdio;
import std.string;
import std.range;
import std.algorithm;

class SquaresRange {
    int first;
    this(int first = 0) {
        this.first = first;
    }
    enum empty = false;
    @property int front() const {
        return opIndex(0);
    }
    void popFront() {
        ++first;
    }
    @property SquaresRange save() const {
        return new SquaresRange(first);
    }
    int opIndex(size_t index) const {
        /* Эта функция отрабатывает за постоянное время */
        immutable integerValue = first + cast(int)index;
        return integerValue * integerValue;
    }
}

bool are_lastTwoDigitsSame(int value) { // Функция, возвращающая true, если последние
две цифры аргумента одинаковы
```

```
    /* Должен иметь как минимум две цифры */
    if (value < 10) {
        return false;
    }

    /* Число из последних двух цифр должно делиться на 11 */
    immutable lastTwoDigits = value % 100;
    return (lastTwoDigits % 11) == 0;
}

void main() {
    auto squares = new SquaresRange();

    writeln(squares[5]);

    writeln(squares[10]);

    squares.popFrontN(5);
    writeln(squares[0]);

    writeln(squares.take(50).filter!are_lastTwoDigitsSame);
}
```

Придётся продолжать:

- *Функция **filter** объявлена в модуле стандартной библиотеки **std.algorithm.iteration** и публично импортируется модулем **std.algorithm**. Она принимает диапазон и функцию-предикат (в нашем случае функцию are_lastTwoDigitsSame), и выдаёт на выходе новый диапазон с теми элементами исходного диапазона, для которых предикат возвращает true.*

– прим. пер.

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
25
100
25
[100, 144, 400, 900, 1444, 1600, 2500]
```

Конечный диапазон с произвольным доступом

Здесь дополнительно к функциям двунаправленного диапазона требуются методы **opIndex()** и **length** . Это объясняется с помощью подробного примера, в котором используется последовательность Фибоначчи и пример с диапазоном квадратов, использованные ранее. Этот пример хорошо работает на обычном D-компиляторе, но не работает в онлайн-компиляторе.

Пример

```
import std.array;
import std.stdio;
import std.string;
import std.range;
import std.algorithm;

struct FibonacciSeries {
    int first = 0;
```

```
    int second = 1;
    enum empty = false;   //  бесконечный диапазон

    @property int front() const {
        return first;
    }
    void popFront() {
        int third = first + second;
        first = second;
        second = third;
    }
    @property FibonacciSeries save() const {
        return this;
    }
}

void report(T)(const dchar[] title, const ref T range) {
    writefln("%40s: %s", title, range.take(5));
}

class SquaresRange {
    int first;
    this(int first = 0) {
        this.first = first;
    }
    enum empty = false;
    @property int front() const {
        return opIndex(0);
    }
    void popFront() {
        ++first;
    }
    @property SquaresRange save() const {
        return new SquaresRange(first);
    }
    int opIndex(size_t index) const {
        /* Эта функция отрабатывает за постоянное время */
        immutable integerValue = first + cast(int)index;
        return integerValue * integerValue;
    }
}


struct Together {
    const(int)[][] slices;
    this(const(int)[][] slices ...) {
        this.slices = slices.dup;
        clearFront();
        clearBack();
    }
    private void clearFront() {
        while (!slices.empty && slices.front.empty) {
            slices.popFront();
        }
    }
    private void clearBack() {
        while (!slices.empty && slices.back.empty) {
            slices.popBack();
        }
    }
    @property bool empty() const {
        return slices.empty;
```

```
    }
    @property int front() const {
        return slices.front.front;
    }
    void popFront() {
        slices.front.popFront();
        clearFront();
    }
    @property Together save() const {
        return Together(slices.dup);
    }
    @property int back() const {
        return slices.back.back;
    }
    void popBack() {
        slices.back.popBack();
        clearBack();
    }
    @property size_t length() const {
        return reduce!((a, b) => a + b.length)(size_t.init, slices);
    }
    int opIndex(size_t index) const {
        /* Сохранить индекс для сообщения об ошибке */
        immutable originalIndex = index;

        foreach (slice; slices) {
            if (slice.length > index) {
                return slice[index];
            } else {
                index -= slice.length;
            }
        }
        throw new Exception(
            format("Invalid index: %s (length: %s)", originalIndex, this.length));
    }
}
void main() {
    auto range = Together(FibonacciSeries().take(10).array, [ 777, 888 ],
        (new SquaresRange()).take(5).array);
    writeln(range.save);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 777, 888, 0, 1, 4, 9, 16]
```

Выходной диапазон

Выходной диапазон представляет выходной поток элементов, аналогичный отправке символов в stdout. OutputRange требует поддержки операции put(range, element). **put()** – это функция, определённая в модуле std.range. Она определяет возможности диапазона и элемента во время компиляции и использует наиболее подходящий метод для вывода элементов. Ниже приведен простой пример.

```
import std.algorithm;
import std.stdio;

struct MultiFile {
    string delimiter;
    File[] files;
```

```
   this(string delimiter, string[] fileNames ...) {
      this.delimiter = delimiter;

      /* stdout всегда включен */
      this.files ~= stdout;

      /* Файловый объект для каждого имени файла */
      foreach (fileName; fileNames) {
         this.files ~= File(fileName, "w");
      }
   }
   void put(T)(T element) {
      foreach (file; files) {
         file.write(element, delimiter);
      }
   }
}
void main() {
   auto output = MultiFile("\n", "output_0", "output_1");
   copy([ 1, 2, 3], output);
   copy([ "красный", "синий", "зелёный" ], output);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
[1, 2, 3]
["красный", "синий", "зелёный"]
```

Псевдонимы

Псевдоним, как об этом говорит его название, является альтернативным именем для существующих имён. Синтаксис для псевдонима показан ниже.

```
alias новое_имя = существующее_имя;
```

Ниже приведен устаревший синтаксис, на всякий случай, если вы будете изучать какие-либо примеры в устаревшем формате. Настоятельно рекомендуется не использовать его.

```
alias существующее_имя новое_имя;
```

Существует также другой синтаксис, который используется с выражением, он приведён ниже, в нём мы можем напрямую использовать имя псевдонима вместо выражения.

```
alias выражение имя_псевдонима;
```

Как вы, возможно, знаете, typedef даёт возможность создавать новые типы. Псевдоним может выполнять работу typedef и даже больше. Ниже приведен простой пример использования псевдонима, который использует шаблон to из модуля std.conv, предоставляющий возможность преобразования типов.

```
import std.stdio;
import std.conv:to;

alias to!(string) toString;

void main() {
   int a = 10;
   string s = "Проверка "~toString(a);
   writeln(s);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Проверка 10
```

В приведенном выше примере вместо использования выражения to!string(a), мы назначили его псевдониму toString, что сделало его более удобным и понятным.

Псевдоним для кортежа

Давайте посмотрим на другой пример, где мы можем установить имя псевдонима для кортежа типов.

```
import std.stdio;
import std.typetuple;

alias TL = TypeTuple!(int, long);

void method1(TL tl) {
   writeln(tl[0],"\t", tl[1] );
}

void main() {
   method1(5, 6L);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
5       6
```

В приведённом выше примере кортеж типов присваивается переменной псевдонима и упрощает определение метода и доступ к переменным. Такой вид доступ ещё более полезен, если нам понадобится повторно использовать такие кортежи типов.

Псевдоним для типов данных

Существует множество случаев, когда нам нужно переопределить обычные типы данных, которые необходимо использовать в приложении. Когда несколько программистов пишут приложение, может случиться так, что один человек использует int, другой double и т.д. Чтобы избежать таких конфликтов, мы часто используем собственные типы для типов данных. Ниже приведен простой пример.

Пример

```
import std.stdio;

alias myAppNumber = int;
alias myAppString = string;

void main() {
   myAppNumber i = 10;
   myAppString s = "ТестоваяСтрока";

   writeln(i,s);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
10ТестоваяСтрока
```

Псевдонимы для переменных класса

Часто возникает потребность, когда нам нужно получить доступ к переменным-членам суперкласса в подклассе, это может быть сделано с помощью псевдонима, возможно, под другим именем.

Если вы не знакомы с концепцией классов и наследования, перед изучением этого раздела ознакомьтесь с руководством по классам и наследованию.

Пример

Ниже приведен простой пример.

```
import std.stdio;

class Shape {
   int area;
}

class Square : Shape {
   string name() const @property {
      return "Square";
   }
   alias squareArea = Shape.area;
}
```

```
void main() {
   auto square = new Square;
   square.squareArea = 42;
   writeln(square.name);
   writeln(square.squareArea);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Square
42
```

Псевдоним this

Псевдоним предоставляет возможность автоматического преобразования пользовательских типов. Синтаксис показан ниже, где ключевые слова alias и this записываются по обе стороны от переменной-члена или функции-члена.

```
alias переменная_член_или_метод this;
```

Пример

Ниже приведен пример, показывающий силу alias this.

```
import std.stdio;

struct Rectangle {
   long length;
   long breadth;

   double value() const @property {
      return cast(double) length * breadth;
   }
   alias value this;
}
double volume(double rectangle, double height) {
   return rectangle * height;
}

void main() {
   auto rectangle = Rectangle(2, 3);
   writeln(volume(rectangle, 5));
}
```

В приведенном выше примере вы можете видеть, что структура rectangle преобразуется в значение типа double с помощью псевдонима this на метод.

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
30
```

Mixins

Слово "mixins" переводится как "примеси", но в тексте я его, пожалуй, оставлю без перевода – прим. пер.

Mixins – это программные конструкции, которые позволяют подмешивать сгенерированный во время компиляции код внутрь основного исходного кода. Mixins могут быть следующих типов:

- Строковые Mixins
- Шаблоны Mixins
- Mixins пространств имён

Строковые Mixins

В D имеется возможность вставлять код в виде строки, если эта строка известна во время компиляции. Синтаксис строковых mixins показан ниже:

```
mixin (генерируемая_во_время_компиляции_строка)
```

Пример

Ниже приведен простой пример для строковых mixins.

```
import std.stdio;

void main() {
   mixin(`writeln("Привет, мир!");`);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Привет, мир!
```

Вот еще один пример, в котором мы передаём строку во время компиляции, чтобы mixins могли использовать функции для повторного использования кода.

```
import std.stdio;

string print(string s) {
   return `writeln("` ~ s ~ `");`;
}

void main() {
   mixin (print("str1"));
   mixin (print("str2"));
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
str1
str2
```

Шаблоны Mixins

Шаблоны в D определяют шаблоны обобщённого кода, чтобы компилятор мог генерировать из них фактические экземпляры. Шаблоны могут генерировать функции, структуры, объединения, классы, интерфейсы и любой другой допустимый код на D. Синтаксис шаблонов mixins показан ниже.

```
mixin имя_шаблона!(параметры_шаблона)
```

Ниже приведен простой пример для шаблонных mixins, где мы создаем шаблон с классом Department и mixin, создающий экземпляр шаблона, и, следовательно, делаем функции setName и printNames доступными для структуры college.

Пример

```
import std.stdio;

template Department(T, size_t count) {
   T[count] names;
   void setName(size_t index, T name) {
      names[index] = name;
   }

   void printNames() {
      writeln("Названия");

      foreach (i, name; names) {
         writeln(i," : ", name);
      }
   }
}

struct College {
   mixin Department!(string, 2);
}

void main() {
   auto college = College();
   college.setName(0, "название1");
   college.setName(1, "название2");
   college.printNames();
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Названия
0 : название1
1 : название2
```

Mixins пространств имён

Mixins пространств имён используются, чтобы избежать неоднозначностей в шаблонных mixins. Например, могут существовать две переменные, одна из которых явно определена в коде основной программы, а другая – в примешанном. Когда примешанное имя совпадает с именем, находящимся в окружающей области видимости, тогда основным является имя, находящееся в окружающей области. Доступ к примешанному имени осуществляется через имя mixin'a. Этот пример показан ниже.

Пример

```
import std.stdio;

template Person() {
   string name;

   void print() {
      writeln(name);
   }
}

void main() {
   string name;

   mixin Person a;
   name = "имя 1";
   writeln(name);

   a.name = "имя 2";
   print();
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
имя 1
имя 2
```

Модули

Модули являются строительными блоками языка D. Они основаны на простой концепции. Каждый исходный файл является модулем. Соответственно, единственными файлами, в которых мы пишем программы, являются отдельные модули. По умолчанию имя модуля совпадает с именем файла без расширения .d.

При явном указании имя модуля определяется ключевым словом module, которое должно находиться на первой строке, не являющейся комментарием в исходном файле. Например, предположим, что имя исходного файла – «employee.d». Тогда имя модуля указывается ключевым словом *module*, за которым следует *employee*. Это показано ниже.

Слово «employee» означает «сотрудник», оно используется во всех примерах этого раздела – прим. пер.

```
module employee;

class Employee {
   // Определение класса находится здесь.
}
```

Строка объявления модуля не обязательна. Если она не указана, имя модуля такое же, как и имя файла без расширения .d.

Имена файлов и модулей

D поддерживает Unicode в исходном тексте и именах модулей. Однако поддержка Unicode файловыми системами различна. Например, хотя большинство файловых систем в Linux поддерживают Unicode, файловые системы Windows могут не различать буквы нижнего и верхнего регистра для имён файлов. Кроме того, большинство файловых систем ограничивают набор символов, которые могут использоваться для имён файлов и каталогов. Из соображений переносимости я рекомендую использовать для файлов только буквы ASCII в нижнем регистре. Например, «employee.d» будет подходящим именем файла для класса с именем employee.

Соответственно, имя модуля также будет состоять из букв ASCII:

```
module employee;  // Имя модуля, состоящее из букв ASCII

class eëmployёë { }
```

Пакеты D

Комбинация связанных между собой модулей называется пакетом. В D пакеты – это простая концепция: исходные файлы, которые находятся внутри одного и того же каталога, считаются принадлежащими к одному и тому же пакету. Имя каталога становится именем пакета, которое также должно быть указано в виде первой части имен модулей.

Например, если файлы «employee.d» и «office.d» находятся внутри каталога «company», то указание имени каталога вместе с именем модуля делает их частью одного и того же пакета:

```
module company.employee;

class Employee { }
```

Аналогично для модуля office:

```
module company.office;

class Office { }
```

Поскольку имена пакетов соответствуют именам каталогов, имена пакетов модулей, которые находятся глубже одного уровня каталогов, должны отражать эту иерархию. Например, если в каталоге «company» включен каталог «branch», имя модуля внутри этого каталога также должно включать branch.

```
module company.branch.employee;
```

Использование модулей в программах

Ключевое слово import, которое мы использовали почти в каждой программе, предназначено для введения другого модуля в текущий модуль:

```
import std.stdio;
```

Имя модуля также может содержать имя пакета. Например, часть std. в этом утверждении указывает, что stdio является модулем, который является частью пакета std.

Расположение модулей

Компилятор ищет файлы модулей, прямо преобразуя имена пакетов и модулей в имена каталогов и файлов.

Например, модули company.employee и company.office будут расположены соответственно в файлах «company/employee.d» и «company/office.d» (или в «company\employee.d» и «company\office.d», в зависимости от файловой системы).

Длинные и короткие имена модулей

Имена, которые используются в программе, можно указывать с именами модулей и пакетов, как показано ниже.

```
import company.employee;
auto employee0 = Employee();
auto employee1 = company.employee.Employee();
```

Длинные имена обычно не нужны, но иногда возникают конфликты имен. Например, при обращении к имени, которое появляется в нескольких модулях, компилятор не может решить, какой из них имеется в виду. В следующей программе используются длинные имена, чтобы различать два отдельных класса *employee*, объявленные в двух различных модулях: *company* и *college*.

Первый модуль employee в каталоге company выглядит следующим образом.

```
module company.employee;

import std.stdio;

class Employee {
   public:
      string str;
```

```
   void print() {
      writeln("Сотрудник компании: ", str);
   }
}
```

Второй модуль employee в каталоге college выглядит следующим образом.

```
module college.employee;

import std.stdio;

class Employee {
   public:
      string str;

   void print() {
      writeln("Сотрудник колледжа: ", str);
   }
}
```

Основной модуль в файле hello.d должен быть сохранён в каталоге, содержащем каталоги college и company. Он выглядит следующим образом.

```
import company.employee;
import college.employee;

import std.stdio;

void main() {
   auto myemployee1 = new company.employee.Employee();
   myemployee1.str = "сотрудник1";
   myemployee1.print();

   auto myemployee2 = new college.employee.Employee();
   myemployee2.str = "сотрудник2";
   myemployee2.print();
}
```

Ключевого слова import недостаточно для того, чтобы модули стали частью программы. Оно просто предоставляет сущности, объявленные в импортируемом модуле, внутри текущего модуля. Это необходимо только для компиляции кода.

Для вышеприведенной программы необходимо также указать «company/employee.d» и «college/employee.d» в командной строке компиляции.

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
$ dmd hello.d company/employee.d college/employee.d -ofhello
$ ./hello
Сотрудник компании: сотрудник1
Сотрудник колледжа: сотрудник2
```

Шаблоны

Шаблоны являются основой обобщённого программирования, которое включает в себя написание кода таким образом, чтобы он не зависел от какого-либо конкретного типа.

Шаблон представляет собой схему или формулу для создания обобщённого класса или функции.

Шаблоны – это возможность, которая позволяет описывать код как образец, по которому компилятор автоматически сгенерирует программный код. Части исходного кода могут быть оставлены незаполненными для компилятора, пока эта часть не будет использована в программе. Компилятор заполняет недостающие части.

Укажу здесь, что я переводил Учебное пособие по шаблонам языка D. Для глубокого изучения этой не самой простой темы и некоторых смежных тем оно лучше подходит – прим. пер.

Шаблоны функций

Определение функции в качестве шаблона оставляет один или несколько используемых типов как неопределённые, впоследствии они будут выведены компилятором. Типы, которые остаются неопределёнными, задаются в списке параметров шаблона, который находится между именем функции и списком параметров функции. По этой причине шаблоны функций имеют два списка параметров:

- список параметров шаблона
- список параметров функции

```
import std.stdio;

void print(T)(T value) {
   writefln("%s", value);
}

void main() {
   print(42);

   print(1.2);

   print("test");
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
42
1.2
test
```

Шаблон функции с несколькими параметрами шаблона

Может присутствовать несколько параметров-типов. Это показано в следующем примере.

```
import std.stdio;

void print(T1, T2)(T1 value1, T2 value2) {
   writefln(" %s %s", value1, value2);
}

void main() {
   print(42, "Test");

   print(1.2, 33);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
 42 Test
 1.2 33
```

Шаблоны класса

Так же, как мы можем определить шаблоны функций, также мы можем определить шаблоны классов. Следующий пример определяет класс Stack и реализует обобщённые методы push и pop для элементов из стека.

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

class Stack(T) {
   private:
      T[] elements;
   public:
      void push(T element) {
         elements ~= element;
      }
      void pop() {
         --elements.length;
      }
      T top() const @property {
         return elements[$ - 1];
      }
      size_t length() const @property {
         return elements.length;
      }
}

void main() {
   auto stack = new Stack!string;

   stack.push("Test1");
   stack.push("Test2");

   writeln(stack.top);
   writeln(stack.length);

   stack.pop;
   writeln(stack.top);
   writeln(stack.length);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Test2
2
Test1
1
```

Неизменяемость

Мы часто используем переменные, значение которых можно изменять, но бывает множество случаев, когда изменяемость не требуется. В таких случаях можно использовать const (константные) или immutable (неизменяемые) переменные. Ниже приводится несколько примеров, где можно использовать неизменяемую переменную.

- В случае математических констант, таких как *pi*, которые никогда не меняются.
- В случае массивов, в которых мы хотим хранить значения, которые не требуется изменять.

Свойство неизменяемости позволяет точно знать, являются ли переменные изменяемыми или нет, гарантируя, что какие бы то ни было операции не изменят значение неизменяемых переменных. Это также снижает риск некоторых типов программных ошибок.

Концепция неизменяемости в D представлена ключевыми словами **const** и **immutable**. Хотя эти два слова близки по смыслу, их обязанности в программах различны, и иногда они являются несовместимыми.

Типы неизменяемых переменных в D

Существует три типа определения переменных, которые никогда нельзя будет изменять.

- константы **enum**
- переменные с квалификатором **immutable**
- переменные с квалификатором **const**

Константы *enum* в D

Константы, объявленные как перечисления (enum), позволяют связывать константные значения со значимыми именами во время компиляции. Ниже приведен простой пример.

Пример

```
import std.stdio;

enum Day{
   Sunday = 1,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday
}

void main() {
   Day day;
   day = Day.Sunday;

   if (day == Day.Sunday) {
```

```
      writeln("Этот день - воскресенье");
   }
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Этот день - воскресенье
```

Переменные с квалификатором *immutable* в D

Неизменяемые переменные можно определить во время выполнения программы. Квалификатор immutable просто указывает компилятору, что после инициализации переменная становится неизменной. Ниже приведен простой пример.

Пример

```
import std.stdio;
import std.random;

void main() {
   int min = 1;
   int max = 10;

   immutable number = uniform(min, max + 1);
   // невозможно изменить переменную number
   // number = 34; // приведёт к ошибке компиляции
   typeof(number) value = 100;

   writeln(typeof(number).stringof, number);
   writeln(typeof(value).stringof, value);
}
```

Когда вы скомпилируете и выполните эту программу, её результат будет похож на следующий:

```
immutable(int)4
immutable(int)100
```

В приведенном выше примере вы можете видеть, как можно переносить тип данных на другую переменную и использовать stringof при выводе на экран.

Переменные с квалификатором *const* в D

Константные переменные нельзя изменять аналогично неизменяемым. Неизменяемые переменные можно передавать в функции в качестве их неизменяемых параметров, и поэтому рекомендуется использовать immutable вместо const. Тот же пример, что был использован ранее, ниже модифицирован для const.

Пример

```
import std.stdio;
import std.random;

void main() {
   int min = 1;
   int max = 10;

   const number = uniform(min, max + 1);
   // невозможно изменить переменную number
   // number = 34; // приведёт к ошибке компиляции
```

```
   typeof(number) value = 100;

   writeln(typeof(number).stringof, number);
   writeln(typeof(value).stringof, value);
}
```

Когда вы скомпилируете и выполните эту программу, её результат будет похож на следующий:

```
const(int)7
const(int)100
```

Неизменяемые параметры в D

const стирает информацию о том, является ли исходная переменная изменяемой или неизменяемой, а использование immutable выполняет передачу её другим функциям с сохранением первоначального квалификатора. Ниже приведен простой пример.

Пример

```
import std.stdio;

void print(immutable int[] array) {
   foreach (i, element; array) {
      writefln("%s: %s", i, element);
   }
}

void main() {
   immutable int[] array = [ 1, 2 ];
   print(array);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
0: 1
1: 2
```

Файловый ввод/вывод

Файлы представлены структурой *File* модуля std.stdio. Файл представляет собой последовательность байтов вне зависимости от того, является ли он текстовым или двоичным файлом.

Язык D предоставляет доступ как к высокоуровневым, так и к низкоуровневым (уровень OS) функциям для работы с файлами на ваших устройствах хранения.

Открытие файлов в D

Стандартные потоки ввода-вывода stdin и stdout уже открыты при запуске программ. Они готовы к использованию. С другой стороны, файлы необходимо сначала открыть, указав имя файла и необходимые права доступа.

```
File file = File(путь_к_файлу, "режим");
```

Здесь **путь_к_файлу** является строковым литералом с именем файла, а **режим** доступа может иметь одно из следующих значений:

№	Режим и описание
1	**r** Открывает существующий текстовый файл для чтения.
2	**w** Открывает текстовый файл для записи, если он не существует, создаётся новый файл. В этот момент ваша программа начнёт писать контент с самого начала файла.
3	**a** Открывает текстовый файл для записи в режиме добавления, если он не существует, создается новый файл. В этот момент ваша программа начнёт добавлять содержимое к существующему файлу.
4	**r+** Открывает текстовый файл сразу для чтения и записи.
5	**w+** Открывает текстовый файл сразу для чтения и записи. Если файл существует, то сначала он обрезается до нулевой длины, в противном случае файл создаётся.
6	**a+** Открывает текстовый файл сразу для чтения и записи. Если файл не существует, он создаётся. Чтение начнётся с самого начала, но запись будет вестись только в режиме добавления.

Закрытие файлов в D

Чтобы закрыть файл, используйте функцию file.close(), где file содержит ссылку на файл.

```
file.close();
```

Любой файл, открытый программой, должен быть закрыт, когда программа закончит использовать этот файл. В большинстве случаев файлы не нужно закрывать явно; они автоматически закрываются при уничтожении объектов File.

Запись в файл

file.writeln используется для записи в открытый файл.

```
file.writeln("привет");
```

```
import std.stdio;
import std.file;

void main() {
   File file = File("test.txt", "w");
   file.writeln("привет");
   file.close();
}
```

Когда этот код будет скомпилирован и выполнен, он создаст новый файл **test.txt** в каталоге, в котором он был запущен (в рабочем каталоге программы).

Чтение файла

Следующий метод читает одну строку из файла:

```
string s = file.readln();
```

Ниже приведен полный пример с чтением и записью.

```
import std.stdio;
import std.file;

void main() {
   File file = File("test.txt", "w");
   file.writeln("привет");
   file.close();
   file = File("test.txt", "r");

   string s = file.readln();
   writeln(s);

   file.close();
}
```

Когда вы скомпилируете и выполните эту программу, она запишет и прочитает файл, а затем выведет следующий результат:

```
привет
```

Вот ещё один пример, в котором файл читается до конца.

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

void main() {
```

```
   File file = File("test.txt", "w");
   file.writeln("привет");
   file.writeln("мир");
   file.close();
   file = File("test.txt", "r");

   while (!file.eof()) {
      string line = chomp(file.readln());
      writeln("строка - ", line);
   }
}
```

Когда вы скомпилируете и выполните эту программу, она запишет файл, затем его прочитает и выведет следующий результат:

```
строка - привет
строка - мир
строка -
```

В приведенном выше примере вы можете увидеть пустую третью строку, так как writeln создаёт следующую строку после своего выполнения.

Параллельное выполнение

Параллелизм делает возможным запуск программы по нескольким потокам одновременно. Примером параллельной программы является веб-сервер, одновременно реагирующий на множество клиентов. Параллельное выполнение осуществляется достаточно просто при передаче сообщений, но может быть очень сложным, если оно основано на совместном использовании данных.

Работу с параллельным выполнением обеспечивает модуль стандартной библиотеки std.concurrency – прим.пер.

Данные, передаваемые между потоками, называются сообщениями. Сообщения могут состоять из переменных любого типа и любого количества. Каждый поток имеет идентификатор, который используется для указания получателей сообщений. Любой поток, который запускает другой поток, называется владельцем нового потока.

Инициирование потоков в D

Функция spawn() принимает указатель в качестве параметра и запускает новый поток, стартующий с этой функции. Любые операции, выполняемые этой функцией, включая другие функции, которые она может вызвать, будут выполняться в новом потоке. Owner (*владелец*) и worker (*работник*) начинают выполняться отдельно, как если бы они были независимыми программами.

Пример

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

void worker(int a) {
   foreach (i; 0 .. 4) {
      Thread.sleep(dur!"seconds"(1));
      writeln("Рабочий поток ",a + i);
   }
}

void main() {
   foreach (i; 1 .. 4) {
      Thread.sleep(dur!"seconds"(2));
      writeln("Главный поток ",i);
      spawn(&worker, i * 5);
   }

   writeln("Функция main завершена.");
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит результат, похожий на это:

```
Главный поток 1
Рабочий поток 5
Главный поток 2
```

```
Рабочий поток 6
Рабочий поток 7
Рабочий поток 10
Главный поток 3
Функция main завершена.
Рабочий поток 8
Рабочий поток 11
Рабочий поток 15
Рабочий поток 12
Рабочий поток 16
Рабочий поток 13
Рабочий поток 17
Рабочий поток 18
```

Идентификаторы потоков в D

Переменная *thisTid*, доступная глобально на уровне модуля, всегда является идентификатором текущего потока. Также вы можете получить threadId, когда вызывается spawn. Пример показан ниже.

Пример

```
import std.stdio;
import std.concurrency;

void printTid(string tag) {
   writefln("%s: %s, адрес: %s", tag, thisTid, &thisTid);
}

void worker() {
   printTid("Рабочий поток ");
}

void main() {
   Tid myWorker = spawn(&worker);

   printTid("Главный поток ");

   writeln(myWorker);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит результат, похожий на это:

```
Главный поток : Tid(6b0100), адрес: 40BB30
Рабочий поток : Tid(6b0080), адрес: 40BB30
Tid(6b0080)
```

Передача сообщений в D

Функция send() отправляет сообщения, а функция receiveOnly() ожидает сообщения определенного типа. Существуют и другие подобные функции: prioritySend(), receive() и receiveTimeout().

Главный поток в следующей программе отправляет своему рабочему потоку сообщение типа int и ожидает от него сообщения типа double. Потоки продолжают отправлять сообщения туда и обратно, пока главный поток не отправит отрицательное целое число. Пример показан ниже.

Пример

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

void workerFunc(Tid tid) {
   int value = 0;
   while (value >= 0) {
      value = receiveOnly!int();
      auto result = to!double(value) * 5; tid.send(result);
   }
}

void main() {
   Tid worker = spawn(&workerFunc,thisTid);

   foreach (value; 5 .. 10) {
      worker.send(value);
      auto result = receiveOnly!double();
      writefln("отослано: %s, принято: %s", value, result);
   }

   worker.send(-1);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
отослано: 5, принято: 25
отослано: 6, принято: 30
отослано: 7, принято: 35
отослано: 8, принято: 40
отослано: 9, принято: 45
```

Передача сообщения с ожиданием в D

Простой пример с передачей сообщений и ожиданием показан ниже.

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

void workerFunc(Tid tid) {
   Thread.sleep(dur!("msecs")( 500 ),);
   tid.send("привет");
}

void main() {
   spawn(&workerFunc,thisTid);
   writeln("Ожидание сообщения");
   bool received = false;

   while (!received) {
      received = receiveTimeout(dur!("msecs")( 100 ), (string message) {
         writeln("принято: ", message);
      });

      if (!received) {
         writeln("... пока нет сообщения");
      }
```

```
    }
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Ожидание сообщения
... пока нет сообщения
... пока нет сообщения
... пока нет сообщения
... пока нет сообщения
... пока нет сообщения
принято: привет
```

Обработка исключений

Исключение – это проблема, которая может возникнуть во время выполнения программы. Это ответ D на исключительные ситуации, которые возникают во время выполнения программы, например, при попытке деления на ноль.

Исключения предоставляют способ передачи контроля из одной части программы в другую. В D обработка исключений производится с помощью трех ключевых слов: **try**, **catch** и **throw**.

- **throw** − При возникновении проблемы программа бросает исключение. Это делается с использованием ключевого слова **throw**.
- **catch** − Программа получает исключение через обработчик исключений в том месте в программе, где вы хотите справиться с этой проблемой. Ключевое слово **catch** указывает на вылавливание исключений.
- **try** − Блок **try** определяет блок кода, в котором могут активироваться определённые исключения. За ним следует один или несколько блоков catch.

Если предполагается, что блок может вызвать исключение, его ловят с использованием комбинации ключевых слов **try** и **catch**. Блок try/catch помещается вокруг кода, который может сгенерировать исключение. Код внутри блока try/catch называется защищённым кодом, а синтаксис использования try/catch выглядит следующим образом:

```
try {
   // защищённый код
}
catch( ExceptionName e1 ) {
   // блок catch
}
catch( ExceptionName e2 ) {
   // блок catch
}
catch( ExceptionName eN ) {
   // блок catch
}
```

Вы можете перечислить несколько операторов **catch**, чтобы ловить разные типы исключений, если ваш блок **try** может вызвать более одного типа исключений в различных ситуациях.

Выбрасывание исключений в D

Исключения можно бросить в любом месте блока кода с использованием команды throw. В оператор **throw** передаётся любое выражение, тип результата этого выражения должен определять тип созданного исключения.

В следующем примере выбрасывается исключение в случае попытки деления на ноль:

Пример

```
double division(int a, int b) {
   if( b == 0 ) {
```

```
      throw new Exception("Ситуация деления на ноль!");
   }

   return (a/b);
}
```

Вылавливание исключений в D

Блок **catch** после блока **try** вылавливает любое исключение. Вы можете указать, какой тип исключения вы хотите поймать, и он будет определяться объявлением об исключении, которое должно находиться в круглых скобках, следующих за ключевым словом catch.

```
try {
   // защищённый код
}

catch( ExceptionName e ) {
   // код для обработки исключения ExceptionName
}
```

Код, показанный выше, позволяет обработать исключения типа **ExceptionName**. Если вы хотите указать, что блок catch должен обрабатывать любые исключения, которые могут быть брошены в блоке try, вы должны поместить многоточие ... между круглыми скобками, заключающими объявление об исключении, следующим образом:

```
try {
   // защищённый код
}

catch(...) {
   // код для обработки любого исключения
}
```

В следующем примере генерируется исключение деления на ноль. Оно ловится в блоке catch.

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

string division(int a, int b) {
   string result = "";

   try {
      if( b == 0 ) {
         throw new Exception("Нельзя делить на ноль!");
      } else {
         result = format("%s",a/b);
      }
   } catch (Exception e) {
      result = e.msg;
   }

   return result;
}

void main () {
   int x = 50;
   int y = 0;

   writeln(division(x, y));

   y = 10;
```

```
   writeln(division(x, y));
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Нельзя делить на ноль!
5
```

Контрактное программирование

Контрактное программирование в языке D сосредоточено на предоставлении простых и понятных средств обработки ошибок. Контрактное программирование в D реализуется тремя типами блоков кода:

- блок body
- блок in
- блок out

Блок body в D

Блок body содержит фактический исполняемый код. Блоки in и out являются необязательными, а блок body является обязательным. Ниже приведён простой синтаксис.

```
возвращаемый_тип имя_функции(параметры_функции)
in {
   // блок in
}

out (result) {
   // блок out
}

body {
   // фактический блок функции
}
```

Блок in для предусловий в D

Блок in используется для простых предварительных условий, в которых проверяется, допустимы ли входные параметры и находятся ли они в пределах того диапазона, который может обрабатываться кодом. Преимущество блока in в том, что все входные условия могут храниться вместе и отделяются от фактического тела функции. Ниже приведено простое предусловие для проверки пароля на минимальную длину.

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

bool isValid(string password)
in {
   assert(password.length>=5);
}

body {
   // другие условия
   return true;
}

void main() {
   writeln(isValid("пароль"));
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
true
```

Блоки out для постусловий в D

Блок out заботится о возвращаемых значениях из функции. Он подтверждает, что возвращаемое значение находится в ожидаемом диапазоне. Ниже показан простой пример, содержащий как in, так и out; здесь пара месяц, год преобразуется в возраст в комбинированной десятичной форме.

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

double getAge(double months,double years)
in {
   assert(months >= 0);
   assert(months <= 12);
}

out (result) {
   assert(result>=years);
}

body {
   return years + months/12;
}

void main () {
   writeln(getAge(10,12));
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
12.8333
```

Условная компиляция

Условная компиляция – это процесс выбора, какой код компилировать, а какой не компилировать, подобно #if / #else / #endif в C и C++. Любой оператор, который ещё не скомпилирован, должен быть синтаксически правильным.

Условная компиляция включает в себя проверки условий, которые можно вычислить во время компиляции. Условные выражения времени выполнения, такие как if, for, while не являются выражениями условной компиляции. Для неё предназначены следующие операторы D:

- debug
- version
- static if

Выражение debug в D

Оператор *debug* (отладка) полезен при разработке программы. Выражения и инструкции, помеченные как debug, компилируются в программе только при включённом переключателе компилятора -debug.

```
debug условно_компилируемое_выражение;

debug {
   // ... условно компилируемый код ...
} else {
   // ... код, который компилируется в противном случае ...
}
```

Предложение *else* является необязательным. И одиночное выражение, и блок кода, показанные выше, компилируются только в том случае, если у компилятора включен параметр -debug.

Вместо того, чтобы вообще удалять строки программы, их можно помечать оператором debug.

```
debug writefln("%s оператор, действующий только в режиме debug", value);
```

Такие строки включаются в программу только при наличии параметра компилятора -debug.

```
dmd test.d -oftest -w -debug
```

Выражение debug(tag) в D

Операторам debug можно присваивать имена (теги), таким образом они будут включаться в программу выборочно.

```
debug(mytag) writefln("%s не найдено", value);
```

Такие строки включаются в программу только при наличии параметра компилятора -debug.

```
dmd test.d -oftest -w -debug = mytag
```

Операторы debug с блоком кода также могут иметь теги.

```
debug(mytag) {
   //
}
```

Допустимо одновременно включать несколько тегов отладки.

```
dmd test.d -oftest -w -debug = mytag1 -debug = mytag2
```

Выражение debug(level) в D

Иногда полезнее связывать отладочные выражения с числовыми уровнями. Повышение уровня может предоставить более подробную информацию.

```
import std.stdio;

void myFunction() {
   debug(1) writeln("debug1");
   debug(2) writeln("debug2");
}

void main() {
   myFunction();
}
```

Будут скомпилированы отладочные выражения и блоки, которые меньше или равны указанному уровню,

```
$ dmd test.d -oftest -w -debug = 1
$ ./test
debug1
```

Выражения version(tag) and version(level) в D

Оператор *version* (версия) похож на debug и используется таким же образом. Предложение else необязательно. Хотя version работает практически так же, как и debug, наличие отдельных ключевых слов помогает различать области их применения. Как и при debug, можно включить больше одной версии.

```
import std.stdio;

void myFunction() {
   version(1) writeln("version1");
   version(2) writeln("version2");
}

void main() {
   myFunction();
}
```

Будут скомпилированы версионные выражения и блоки, которые меньше или равны указанному уровню,

```
$ dmd test.d -oftest -w -version = 1
$ ./test
version1
```

Static if

Static if – это эквивалент инструкции if времени компиляции. Как и оператор if, static if принимает логическое выражение и вычисляет его. В отличие от оператора if, static if не

относится к потоку выполнения; вместо этого он определяет, нужно ли включать фрагмент кода в программу или нет.

Выражение if не связано с оператором *is*, который мы видели ранее, как синтаксически, так и семантически. Он вычисляется во время компиляции. Его результатом является значение типа int, равное 0 или 1; в зависимости от выражения, указанного в круглых скобках. Хотя принимаемое выражение не является логическим выражением, само оно используется как логическое выражение времени компиляции. Это особенно полезно для условий static if и ограничений шаблонов.

```
import std.stdio;

enum Days {
   sun,
   mon,
   tue,
   wed,
   thu,
   fri,
   sat
};

void myFunction(T)(T mytemplate) {
   static if (is (T == class)) {
      writeln("Этот тип – класс");
   } else static if (is (T == enum)) {
      writeln("Этот тип – перечисление");
   }
}

void main() {
   Days day;
   myFunction(day);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Этот тип – перечисление
```

Объектно-ориентированное программирование

Классы и объекты

Классы являются центральной возможностью в языке D, поддерживающей объектно-ориентированное программирование, и они часто называются пользовательскими типами.

Класс используется для указания формы объекта и сочетает в себе представление данных и методы для управления этими данными в один аккуратный пакет. Данные и функции внутри класса называются членами класса.

Определения класса в D

Когда вы определяете класс, вы определяете схему для типа данных. В нём нет фактического определения каких-либо данных, но есть определение того, что означает имя класса, то есть то, что будет представлять собой объект класса и какие операции можно выполнять с таким объектом.

Определение класса начинается с ключевого слова **class**, за которым следует имя класса; затем идёт тело класса, заключённое в пару фигурных скобок. Определение класса не должно завершаться ни точкой с запятой, ни списком объявлений. Например, мы определили тип данных Box (коробка) с использованием ключевого слова **class** следующим образом:

```
class Box {
   public:
      double length;   // Длина коробки
      double breadth;  // Ширина коробки
      double height;   // Высота коробки
}
```

Ключевое слово **public** определяет атрибут доступа для членов класса, которые следуют за ним. Доступ к публичному члену можно получить извне класса в пределах области видимости объекта класса. Вы также можете установить для членов класса атрибуты **private** или **protected**, которые мы обсудим в подразделе Инкапсуляция.

Объявление объектов в D

Класс предоставляет собой прототип для объектов, поэтому по-существу объект создается из класса. Вы объявляете объекты класса точно таким же образом, как вы объявляете переменные базовых типов. Следующие утверждения объявляют два объекта класса Box:

```
Box box1;          // Объявить box1 типа Box
Box box2;          // Объявить box2 типа Box
```

Оба объекта box1 и box2 имеют собственные копии данных-членов.

К сожалению, здесь авторы учебника не совсем правы. Такое объявление подходит для объектов структур, а для объектов классов память не будет распределена, и будет вызвана ошибка времени выполнения при попытке доступа к членам класса. Объекты классов надо создавать из кучи с помощью оператора ***new****:*

```
Box box1 = new Box();           // Объявить и создать box1 типа Box
Box box2 = new Box();           // Объявить и создать box2 типа Box
```

– прим. пер.

Доступ к членам данных

Доступ к публичным членам данных объекта класса осуществляется с помощью оператора прямого доступа (.). Давайте попробуем следующий пример, чтобы всё было ясно:

```
import std.stdio;

class Box {
   public:
      double length;   // Длина коробки
      double breadth;  // Ширина коробки
      double height;   // Высота коробки
}

void main() {
   Box box1 = new Box();    // Объявить и создать box1 типа Box
   Box box2 = new Box();    // Объявить и создать box2 типа Box
   double volume = 0.0;     // Здесь сохраняем объём коробки.

   // спецификация коробки 1
   box1.height = 5.0;
   box1.length = 6.0;
   box1.breadth = 7.0;

   // спецификация коробки 1
   box2.height = 10.0;
   box2.length = 12.0;
   box2.breadth = 13.0;

   // объём коробки 1
   volume = box1.height * box1.length * box1.breadth;
   writeln("Объём коробки 1 : ",volume);

   // объём коробки 2
   volume = box2.height * box2.length * box2.breadth;
   writeln("Объём коробки 2 : ", volume);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Объём коробки 1 : 210
Объём коробки 2 : 1560
```

Важно отметить, что доступ к закрытым и защищённым членам невозможен напрямую, с использованием оператора прямого доступа к члену (.). Вскоре вы узнаете, как можно получить доступ к закрытым и защищённым членам.

Классы и объекты в D

К настоящему моменту у вас есть базовое представление о классах и объектах в D. Существуют и другие интересные понятия, связанные с классами и объектами, которые мы обсудим в следующих подразделах, перечисленных ниже:

№ п/п	Понятие и описание
1	**Функции-члены класса** Функция-член класса – это функция, у которой определение её или её прототипа находится в определении класса, как и для любой другой переменной.
2	**Модификаторы доступа класса** Член класса может быть определён как public (открытый), private (закрытый) или protected (защищенный). По умолчанию члены будут считаться закрытыми.
3	**Конструкторы и деструкторы** Конструктор класса является специальной функцией в классе, которая вызывается при создании нового объекта класса. Деструктор также является специальной функцией, которая вызывается при удалении созданного объекта.
4	**Указатель this в D** Каждый объект имеет специальный указатель с именем **this**, который указывает на сам этот объект.
5	**Указатели на классы в D** Указатель на класс делается точно так же, как указатель на структуру.
6	**Статические члены класса** Как данные-члены, так и функции-члены класса могут быть объявлены как статические.

Функции-члены класса

Функция-член (или "метод") – это функция, специфичная для класса. Она работает в любом объекте класса, членом которого она является, и имеет доступ ко всем членам класса этого объекта.

Функция-член вызывается с помощью применения оператора точки (.) к объекту, где она манипулирует данными, связанными с этим объектом.

Давайте применим эти концепции, чтобы присвоить значение различным членам класса и получить эти значения:

```
import std.stdio;

class Box {
   public:
      double length;         // Длина коробки
      double breadth;        // Ширина коробки
      double height;         // Высота коробки
```

```
   double getVolume() {
      return length * breadth * height;
   }
   void setLength( double len ) {
      length = len;
   }
   void setBreadth( double bre ) {
      breadth = bre;
   }
   void setHeight( double hei ) {
      height = hei;
   }
}

void main( ) {
   Box box1 = new Box();    // Объявить и создать box1 типа Box
   Box box2 = new Box();    // Объявить и создать box2 типа Box
   double volume = 0.0;     // Здесь сохраняем объём коробки.

   // спецификация коробки 1
   box1.setLength(6.0);
   box1.setBreadth(7.0);
   box1.setHeight(5.0);

   // спецификация коробки 2
   box2.setLength(12.0);
   box2.setBreadth(13.0);
   box2.setHeight(10.0);

   // объём коробки 1
   volume = box1.getVolume();
   writeln("Объём коробки 1 : ",volume);

   // объём коробки 2
   volume = box2.getVolume();
   writeln("Объём коробки 2 : ", volume);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Объём коробки 1 : 210
Объём коробки 2 : 1560
```

Модификаторы доступа класса

Скрытие данных – одна из важных особенностей объектно-ориентированного программирования, которая позволяет запретить функциям программы напрямую обращаться к внутреннему представлению класса. Ограничение доступа к членам класса определяется разделами внутри тела класса, помеченными словами **public**, **private**, и **protected**. Ключевые слова public (открытый), private (закрытый) и protected (защищённый) называются спецификаторами доступа.

Класс может иметь по нескольку разделов, помеченных как public, protected, или private. Каждый раздел остаётся в силе до тех пор, пока не появится метка другого раздела или правая скобка, закрывающая тело класса. По умолчанию доступ для членов классов является открытым (public).

```
class Base {

   public:

  // открытые члены класса расположены здесь

   protected:

  // защищённые члены класса расположены здесь

   private:

  // закрытые члены класса расположены здесь

};
```

Открытые члены в D

Член класса со спецификатором **public** доступен из любого места вне класса, но в рамках программы. Вы можете присваивать и получать значение открытых переменных без использования какой-либо функции-члена, как показано в следующем примере:

Пример

```
import std.stdio;

class Line {
   public:
      double length;

      double getLength() {
         return length ;
      }

      void setLength( double len ) {
         length = len;
      }
}

void main( ) {
   Line line = new Line();

   // установить длину линии
   line.setLength(6.0);
   writeln("Длина линии : ", line.getLength());

   // установить длину линии без использования фунции-члена
   line.length = 10.0; // Допустимо, т.к. length объявлена как public
   writeln("Длина линии : ", line.length);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Длина линии : 6
Длина линии : 10
```

Закрытые члены

К члену класса со спецификатором **private** нельзя получить доступ или даже просмотреть его извне класса. Только члены класса могут иметь доступ к закрытым членам.

Практически вам нужно определить данные в разделе private, а связанные с ними функции в разделе public, чтобы их можно было вызывать извне класса, как показано в следующей программе.

```
import std.stdio;

class Box {
   public:
      double length;

      // Определения функций-членов
      double getWidth() {
         return width ;
      }
      void setWidth( double wid ) {
         width = wid;
      }

   private:
      double width;
}

// Главная функция программы
void main( ) {
   Box box = new Box();

   box.length = 10.0;
   writeln("Длина коробки : ", box.length);

   box.setWidth(10.0);
   writeln("Ширина коробки : ", box.getWidth());
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Длина коробки : 10
Ширина коробки : 10
```

*Очередное нудное уточнение. Правила доступа к членам класса со спецификатором **private**, описанные в этом разделе, на самом деле действуют в языке C++. А в языке D доступ к таким членам открыт из любого места в модуле, в котором объявлен этот класс. Для функций из других модулей доступ действительно закрыт. Так что в приведённом выше примере вполне можно написать в последней строке:*

```
writeln("Ширина коробки : ", box.width);
```

– прим. пер.

Защищенные члены

Переменная или функция-член со спецификатором **protected** очень похожа на закрытый член (private), но она предоставляет ещё одно преимущество, в соответствии с которым к ним имеется доступ в производных классах.

Вы изучите производные классы и наследование в одной из следующих глав. На данный момент вы можете проверить следующий пример, когда один производный класс **SmallBox** унаследован от родительского класса **Box**.

Следующий пример аналогичен приведенному выше примеру, и здесь член **width** доступен из любой функции-члена производного класса SmallBox.

```
import std.stdio;

class Box {
   protected:
      double width;
}

class SmallBox:Box  { // SmallBox - это производный класс.
   public:
      double getSmallWidth() {
         return width ;
      }

      void setSmallWidth( double wid ) {
         width = wid;
      }
}

void main( ) {
   SmallBox box = new SmallBox();

   // установить ширину коробки с использованием функции-члена
   box.setSmallWidth(5.0);
   writeln("Ширина коробки : ", box.getSmallWidth());
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Ширина коробки : 5
```

И ещё добавлю, что кроме трёх рассмотренных в этом разделе, в языке D также существуют ещё два спецификатора доступа: ***package*** *– доступ на уровне пакета, и* ***export*** *– доступ извне программы, используется для динамически связываемых библиотек. – прим. пер.*

Конструкторы и деструкторы

Конструктор класса

Конструктор класса – это специальная функция-член класса, которая выполняется всякий раз, когда мы создаём новые объекты этого класса.

Конструктор имеет имя **this** и у него не должно быть никакого возвращаемого типа, даже void. Конструкторы могут быть очень полезны для установки начальных значений для некоторых переменных-членов.

В следующем примере иллюстрируется концепция конструктора:

```
import std.stdio;

class Line {
   public:
      void setLength( double len ) {
         length = len;
      }
      double getLength() {
         return length;
      }
      this() {
         writeln("Объект создаётся");
      }

   private:
      double length;
}

void main( ) {
   Line line = new Line();

   // задать длину линии
   line.setLength(6.0);
   writeln("Длина линии : " , line.getLength());
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Объект создаётся
Длина линии : 6
```

Конструктор с параметрами

Конструктор по умолчанию не имеет никаких параметров, но если вам это нужно, конструктор может иметь параметры. Он поможет вам присвоить начальные значения объекту во время его создания, как показано в следующем примере:

Пример

```
import std.stdio;

class Line {
   public:
      void setLength( double len ) {
         length = len;
      }
      double getLength() {
         return length;
      }
      this( double len) {
         writeln("Объект создаётся, length = " , len );
         length = len;
      }

   private:
      double length;
}

// Главная функция программы
void main( ) {
   Line line = new Line(10.0);
```

```
   // получить изначально заданную длину.
   writeln("Длина линии : ",line.getLength());

   // переустановить длину линии
   line.setLength(6.0);
   writeln("Длина линии : ", line.getLength());
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Объект создаётся, length = 10
Длина линии : 10
Длина линии : 6
```

Деструктор класса

Деструктор – это особая функция-член класса, которая выполняется всякий раз, когда объект класса выходит из области видимости, или когда выражение удаления применяется к указателю на объект этого класса.

Деструктор имеет имя **~this**, т.е. такое же, как и у конструктора, но префиксом тильды (~). Он не может ни возвращать значение, ни принимать какие-либо параметры. Деструктор может быть очень полезен для освобождения ресурсов перед выходом из программы, например, закрыть файлы, освободить память и т. д.

В следующем примере иллюстрируется концепция деструктора:

```
import std.stdio;

class Line {
   public:
      this() {
         writeln("Объект создаётся");
      }

      ~this() {
         writeln("Объект удаляется");
      }

      void setLength( double len ) {
         length = len;
      }

      double getLength() {
         return length;
      }

   private:
      double length;
}

// Главная функция программы
void main( ) {
   Line line = new Line();

   // задать длину линии
   line.setLength(6.0);
   writeln("Длина линии : ", line.getLength());
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Объект создаётся
Длина линии : 6
Объект удаляется
```

Указатель *this* в D

Каждый объект в D имеет доступ к собственному адресу через важный указатель, с именем **this**. Этот указатель является неявным параметром для всех функций-членов. Поэтому внутри функции-члена this может использоваться как ссылка на вызывающий объект.

Попробуем следующий пример, чтобы понять идею указателя *this*:

```
import std.stdio;

class Box {
   public:
      // Объявление конструктора
      this(double l = 2.0, double b = 2.0, double h = 2.0) {
         writeln("Вызван конструктор.");
         length = l;
         breadth = b;
         height = h;
      }

      double Volume() {
         return length * breadth * height;
      }

      int compare(Box box) {
         return this.Volume() > box.Volume();
      }

   private:
      double length;     // Длина коробки
      double breadth;    // Ширина коробки
      double height;     // Высота коробки
}

void main() {
   Box box1 = new Box(3.3, 1.2, 1.5);    // Объявление box1
   Box box2 = new Box(8.5, 6.0, 2.0);    // Объявление box2

   if(box1.compare(box2)) {
      writeln("box2 меньше, чем box1");
   } else {
      writeln("box2 равна или больше, чем box1");
   }
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Вызван конструктор.
Вызван конструктор.
box2 равна или больше, чем box1
```

Указатели на классы в D

Указатель на класс в D работает точно так же, как указатель на структуру, и для доступа к элементам класса вы используете оператор доступа точка (.), так же, как и для указателей на структуры. Также как и со всеми указателями, вы должны его инициализировать перед использованием.

Попробуем следующий пример, чтобы понять концепцию указателя на класс:

```
import std.stdio;

class Box {
   public:
      // Объявление конструктора
      this(double l = 2.0, double b = 2.0, double h = 2.0) {
         writeln("Вызван конструктор.");
         length = l;
         breadth = b;
         height = h;
      }

      double Volume() {
         return length * breadth * height;
      }

   private:
      double length;     // Длина коробки
      double breadth;    // Ширина коробки
      double height;     // Высота коробки
}

void main() {
   Box box1 = new Box(3.3, 1.2, 1.5);    // Объявление box1
   Box box2 = new Box(8.5, 6.0, 2.0);    // Объявление box2
   Box *ptrBox;                // Объявление указателя на класс.

   // Сохранить адрес первого объекта
   ptrBox = &box1;

   // Теперь пробуем обратиться к члену, используя оператор доступа
   writeln("Объём box1 : ", ptrBox.Volume());

   // Сохранить адрес второго объекта
   ptrBox = &box2;

   // Теперь пробуем обратиться к члену, используя оператор доступа
   writeln("Объём box2: ", ptrBox.Volume());
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Вызван конструктор.
Вызван конструктор.
Объём box1 : 5.94
Объём box2: 102
```

Статические члены класса

Мы можем определить статические элементы класса, используя ключевое слово **static**. Когда мы объявляем элемент класса статическим, это означает, что независимо от того, сколько объектов класса создано, существует только одна копия статического члена.

Статический член разделяется всеми объектами класса. Все статические данные инициализируются значением по-умолчанию для своего типа данных при загрузке модуля, если нет никакой другой инициализации.

Попробуем следующий пример, чтобы понять концепцию статических данных:

```
import std.stdio;

class Box {
   public:
      static int objectCount = 0;

      // Объявление конструктора
      this(double l = 2.0, double b = 2.0, double h = 2.0) {
         writeln("Вызван конструктор.");
         length = l;
         breadth = b;
         height = h;

         // Увеличение каждый раз при создании объекта
         objectCount++;
      }

      double Volume() {
         return length * breadth * height;
      }

   private:
      double length;     // Длина коробки
      double breadth;    // Ширина коробки
      double height;     // Высота коробки
};

void main() {
   Box box1 = new Box(3.3, 1.2, 1.5);    // Объявление box1
   Box box2 = new Box(8.5, 6.0, 2.0);    // Объявление box2

   // Вывод общего количества объектов.
   writeln("Всего объектов: ", Box.objectCount);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Вызван конструктор.
Вызван конструктор.
Всего объектов: 2
```

Статические функции-члены

Объявляя функцию-член как статическую, вы делаете её независимой от какого-либо конкретного объекта класса. Статическая функция-член может быть вызвана, даже если

объекты класса не существуют, и обращение к статическим функциям можно выполнять с использованием только имени класса через оператор доступа точка.

Статическая функция-член может получить доступ только к статической переменной-члену, другим статическим функциям-членам и любым другим функциям вне класса.

У статических функций-членов есть область видимости класса, и они не имеют доступа к указателю **this** класса. Вы можете использовать статическую функцию-член, чтобы определить, были ли созданы какие-либо объекты класса или нет.

Попробуем следующий пример, чтобы понять идею статических функций-членов:

```
import std.stdio;

class Box {
   public:
      static int objectCount = 0;

      // Объявление конструктора
      this(double l = 2.0, double b = 2.0, double h = 2.0) {
         writeln("Вызван конструктор.");
         length = l;
         breadth = b;
         height = h;

         // Увеличение каждый раз при создании объекта
         objectCount++;
      }

      double Volume() {
         return length * breadth * height;
      }

      static int getCount() {
         return objectCount;
      }

   private:
      double length;     // Длина коробки
      double breadth;    // Ширина коробки
      double height;     // Высота коробки
};

void main() {
   // Вывод общего количества объектов перед созданием объекта.
   writeln("Количество в начале: ", Box.getCount());

   Box box1 = new Box(3.3, 1.2, 1.5);    // Объявление box1
   Box box2 = new Box(8.5, 6.0, 2.0);    // Объявление box2

   // Вывод общего количества объектов после создания объектов.
   writeln("Количество в конце: ", Box.getCount());
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Количество в начале: 0
Вызван конструктор.
Вызван конструктор.
Количество в конце: 2
```

Наследование

Одним из наиболее важных понятий в объектно-ориентированном программировании является наследование. Наследование позволяет определить класс в терминах другого класса, что упрощает создание и поддержку приложения. Это также даёт возможность повторно использовать функциональность кода и ускорять реализацию.

При создании класса вместо написания совершенно новых данных-членов и функций-членов программист может обозначить, что новый класс должен наследовать члены от существующего класса. Этот существующий класс называется **базовым** классом, а новый класс называется **производным**.

Идея наследования реализует отношение "B является A". Например, млекопитающее ЯВЛЯЕТСЯ животным, собака ЯВЛЯЕТСЯ млекопитающим, следовательно, собака ЯВЛЯЕТСЯ животным и так далее.

Базовые классы и производные классы в D

Класс может наследоваться только от одного класса и от нескольких интерфейсов (будут рассмотрены в одном из последующих разделов). Для определения наследуемого класса мы используем список наследования, который может включать имя одного базового класса и, вслед за ним, через запятую, список имён интерфейсов. Список наследования отделяется от имени наследуемого класса двоеточием:

```
class наследуемый_класс: базовый_класс, интерфейс1, интерфейс2
```

Рассмотрим базовый класс **Shape** и его производный класс **Rectangle**:

```
import std.stdio;

// Базовый класс
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
}

// Производный класс
class Rectangle: Shape {
   public:
      int getArea() {
         return (width * height);
      }
}

void main() {
```

```
   Rectangle Rect = new Rectangle();

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Вывод площади объекта.
   writeln("Общая площадь: ", Rect.getArea());
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Общая площадь: 35
```

Контроль доступа и наследование

Производный класс может получить доступ ко всем незакрытым членам своего базового класса. Таким образом, члены базового класса, которые не должны быть доступны для функций-членов производных классов, должны быть объявлены закрытыми (private) в базовом классе.

Производный класс наследует все методы базового класса со следующими исключениями:

- Конструкторы, деструкторы и конструкторы копирования базового класса.
- Перегруженные операторы базового класса.

*Замечание по поводу наследования перегруженных операторов (методов opUnary, opBinary и пр.). Я попробовал использовать класс **Rectangle**, унаследованный от класса **Shape** (см. пример выше) с перегруженным оператором "-" (унарный минус). Сам оператор работал, т.е. формально его наследование работает. Но результатом таких операторов обычно должен быть объект того же типа, как и тот, к которому применяется оператор. Просто это сделать не получается – если функция opUnary объявлена в классе Shape, то она и возвращает объект класса Shape, в то время как если мы применяем унарный минус к объекту Rectangle, то нам нужно возвращать объект класса Rectangle. Возможно, как-то эта проблема решается шаблонной магией, но без неё можно считать, что перегруженные операторы не наследуются – прим. пер.*

Многоуровневое наследование

Наследование может быть с большим количеством уровней, как показано в следующем примере.

```
import std.stdio;

// Базовый класс
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }
```

```
   protected:
      int width;
      int height;
}

// Производный класс
class Rectangle: Shape {
   public:
      int getArea() {
         return (width * height);
      }
}

class Square: Rectangle {
   this(int side) {
      this.setWidth(side);
      this.setHeight(side);
   }
}

void main() {
   Square square = new Square(13);

   // Вывод площади объекта.
   writeln("Общая площадь: ", square.getArea());
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Общая площадь: 169
```

Перегрузка

D позволяет указать более одного определения для имени **функции** или **оператора** в одной и той же области видимости, это называется перегрузкой функции и перегрузкой оператора соответственно.

Перегруженное объявление – это объявление, которое было объявлено с тем же именем, что и предыдущее объявление в той же области видимости, за исключением того, что оба объявления имеют разные аргументы и, очевидно, различные определения (реализацию).

Когда вы вызываете перегруженную **функцию** или **оператор**, компилятор определяет наиболее подходящее определение для использования, сравнивая используемые для вызова функции или оператора типы аргументов, с типами параметров, указанными в определениях. Процесс выбора наиболее подходящей перегруженной функции или оператора называется **разрешением перегрузки**.

Перегрузка функций

Вы можете иметь несколько определений для одного и того же имени функции в одной области видимости. Определения функций должны отличаться друг от друга по типам и/или количеству аргументов в списке аргументов. Вы не можете перегружать объявления функций, которые отличаются только возвращаемым типом.

Пример

В следующем примере используется одна и та же функция **print()** для отображения разных типов данных:

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

class printData {
   public:
      void print(int i) {
         writeln("Вывод int: ",i);
      }

      void print(double f) {
         writeln("Вывод float: ",f );
      }

      void print(string s) {
         writeln("Вывод string: ",s);
      }
};

void main() {
   printData pd = new printData();

   // Вызов print для вывода целого числа
   pd.print(5);

   // Вызов print для вывода числа с плавающей точкой
```

```
   pd.print(500.263);

   // Вызов print для вывода строки
   pd.print("Привет, D");
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Вывод int: 5
Вывод float: 500.263
Вывод string: Привет, D
```

Перегрузка операторов

Вы можете переопределить или перегрузить большинство встроенных операторов, доступных в D. Таким образом, программист может использовать операторы с определёнными пользователем типами также, как со встроенными типами.

Операторы могут быть перегружены с помощью функции с именем, состоящим из строки op, за которой следуют Add, Sub и т.д., исходя из того, какой оператор должен быть перегружен. Мы можем перегрузить оператор +, чтобы складывать две коробки, как показано ниже.

```
Box opAdd(Box b) {
   Box box = new Box();
   box.length = this.length + b.length;
   box.breadth = this.breadth + b.breadth;
   box.height = this.height + b.height;
   return box;
}
```

Замечание.

*Такие функции для перегрузки операторов, как opAdd, opSub и т.д. являются механизмом из первой версии языка D. В D2 в настоящее время они допустимы, но не рекомендуются для использования, т.к. их поддержка не гарантируется в будущем. Вместо них нужно использовать унифицированные шаблонные функции opUnary, opBinary, opCmp и т.д. (см. подраздел **Типы перегрузки операторов**) – прим. пер.*

В следующем примере показана концепция перегрузки оператора с использованием функции-члена. Здесь объект передается в виде аргумента, его свойства доступны через этот объект. Объект, в котором вызван оператор, можно получить с помощью оператора **this**, как показано ниже:

```
import std.stdio;

class Box {
   public:
      double getVolume() {
         return length * breadth * height;
      }

      void setLength( double len ) {
         length = len;
      }

      void setBreadth( double bre ) {
```

```
         breadth = bre;
      }

      void setHeight( double hei ) {
         height = hei;
      }

      Box opAdd(Box b) {
         Box box = new Box();
         box.length = this.length + b.length;
         box.breadth = this.breadth + b.breadth;
         box.height = this.height + b.height;
         return box;
      }

   private:
      double length;       // Длина коробки
      double breadth;      // Ширина коробки
      double height;       // Высота коробки
};

// Основная функция программы
void main( ) {
   Box box1 = new Box();    // Объявление коробки box1 типа Box
   Box box2 = new Box();    // Объявление коробки box2 типа Box
   Box box3 = new Box();    // Объявление коробки box3 типа Box
   double volume = 0.0;     // Здесь хранится объём коробки.

   // спецификация коробки 1
   box1.setLength(6.0);
   box1.setBreadth(7.0);
   box1.setHeight(5.0);

   //спецификация коробки 2
   box2.setLength(12.0);
   box2.setBreadth(13.0);
   box2.setHeight(10.0);

   // объём коробки 1
   volume = box1.getVolume();
   writeln("Объём коробки 1 : ", volume);

   // объём коробки 2
   volume = box2.getVolume();
   writeln("Объём коробки 2 : ", volume);

   // Сложить два объекта следующим образом:
   box3 = box1 + box2;

   // объём коробки 3
   volume = box3.getVolume();
   writeln("Объём коробки 3 : ", volume);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Объём коробки 1 : 210
Объём коробки 2 : 1560
Объём коробки 3 : 5400
```

Типы перегрузки операторов

В основном, существует три типа перегрузки операторов, как указано ниже.

№ п/п	Типы перегрузки
1	**Перегрузка унарных операторов**
2	**Перегрузка бинарных операторов**
3	**Перегрузка операторов сравнения**

Существует гораздо больше методов, ответственных за перегрузку операторов, попробую их перечислить:

- ***opCast*** *– перегрузка оператора приведения типов* ***cast***
- ***opEquals*** *– перегрузка оператора проверки на равенство == (и, косвенно, проверки на неравенство* ***!=****)*
- ***opCall*** *– создание оператора вызова функции, т.е. при наличии метода с таким именем у объекта* ***a****, этот объект можно вызывать как функцию* ***a()***
- ***opAssign****,* ***opIndexAssign*** *– перегрузка операторов присваивания = и присваивания по индексу* ***a[i]=***
- ***opOpAssign****,* ***opIndexOpAssign*** *– перегрузка бинарных операций "на месте", или, другими словами, операций, совмещённых с присваиванием, например +=, -= и т.д., и, соответственно, таких же операций по индексу*
- ***opIndex****,* ***opSlice****,* ***opDollar*** *– перегрузка операторов, связанных с массивами, соответственно, индексации* ***[i]****, выделения среза* ***[i..j]*** *и размера массива* ***$***
- ***opDispatch*** *– создание оператора перенаправления (диспетчера), в который будут перенаправляться все вызовы несуществующих у объекта методов*

– прим. пер.

Перегрузка унарных операторов

В следующей таблице показан список унарных операторов и их назначение.

Имя функции	Оператор	Назначение
opUnary	-	Отрицание (числовое дополнение)
opUnary	+	Такое же значение (или, копия)
opUnary	~	Побитовое отрицание
opUnary	*	Доступ к содержимому указателя
opUnary	++	Увеличение на единицу
opUnary	--	Уменьшение на единицу

Ниже приведен пример, в котором показана перегрузка унарного оператора.

```
import std.stdio;

class Box {
   public:
      double getVolume() {
         return length * breadth * height;
      }

      void setLength( double len ) {
         length = len;
      }

      void setBreadth( double bre ) {
         breadth = bre;
      }

      void setHeight( double hei ) {
         height = hei;
      }

      Box opUnary(string op)() {
         if(op == "++") {
            Box box = new Box();
            box.length = this.length + 1;
            box.breadth = this.breadth + 1 ;
            box.height = this.height + 1;
            return box;
         }
      }

   private:
      double length;       // Длина коробки
      double breadth;      // Ширина коробки
      double height;       // Высота коробки
};

// Основная функция программы
void main( ) {
   Box box1 = new Box();    // Объявление box1 типа Box
   Box box2 = new Box();    // Объявление box2 типа Box
   double volume = 0.0;     // Здесь хранится объём коробки

   // Спецификация box 1
   box1.setLength(6.0);
   box1.setBreadth(7.0);
   box1.setHeight(5.0);

   // объём box 1
   volume = box1.getVolume();
   writeln("Объём box1 : ", volume);

   // Увеличение объекта:
   box2 = ++box1;

   // объём box2
   volume = box2.getVolume();
   writeln("Объём box2 : ", volume);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Объём box1 : 210
Объём box2 : 336
```

Перегрузка бинарных операторов

В следующей таблице показан список бинарных операторов и их назначение.

Имя функции	Оператор	Назначение
opBinary	+	сложение
opBinary	-	вычитание
opBinary	*	умножение
opBinary	/	деление
opBinary	%	остаток от деления
opBinary	^^	степень
opBinary	&	побитовое И
opBinary	|	побитовое ИЛИ
opBinary	^	побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ
opBinary	<<	битовый сдвиг влево
opBinary	>>	битовый сдвиг вправо со знаком
opBinary	>>>	битовый сдвиг вправо без знака
opBinary	~	соединение
opBinary	in	нахождение в

Ниже приведен пример, в котором показана перегрузка бинарного оператора.

Пример

```
import std.stdio;

class Box {
   public:
      double getVolume() {
         return length * breadth * height;
      }

      void setLength( double len ) {
         length = len;
      }

      void setBreadth( double bre ) {
         breadth = bre;
      }

      void setHeight( double hei ) {
         height = hei;
      }

      Box opBinary(string op)(Box b) {
         if(op == "+") {
            Box box = new Box();
```

```
            box.length = this.length + b.length;
            box.breadth = this.breadth + b.breadth;
            box.height = this.height + b.height;
            return box;
         }
      }
   private:
      double length;      // Длина коробки
      double breadth;     // Ширина коробки
      double height;      // Высота коробки
};

// Основная функция программы
void main( ) {
   Box box1 = new Box();    // Объявление box1 типа Box
   Box box2 = new Box();    // Объявление box2 типа Box
   Box box3 = new Box();    // Объявление box3 типа Box
   double volume = 0.0;     // Здесь хранится объём коробки

   // Спецификация box 1
   box1.setLength(6.0);
   box1.setBreadth(7.0);
   box1.setHeight(5.0);

   // Спецификация box 2
   box2.setLength(12.0);
   box2.setBreadth(13.0);
   box2.setHeight(10.0);

   // объём box 1
   volume = box1.getVolume();
   writeln("Объём Box1 : ", volume);

   // объём box 2
   volume = box2.getVolume();
   writeln("Объём Box2 : ", volume);

   // Сложение двух объектов следующим образом:
   box3 = box1 + box2;

   // объём box 3
   volume = box3.getVolume();
   writeln("Объём Box3 : ", volume);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Объём Box1 : 210
Объём Box2 : 1560
Объём Box3 : 5400
```

Перегрузка операторов сравнения

В следующей таблице показан список операторов сравнения и их назначение.

Имя функции	Оператор	Назначение
opCmp	<	меньше
opCmp	<=	не больше
opCmp	>	больше
opCmp	>=	не меньше

Операторы сравнения используются при сортировке массивов. В следующем примере показано, как использовать операторы сравнения.

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

struct Box {
   int volume;
   int opCmp(const ref Box box) const {
      return (volume == box.volume ? box.volume - volume: volume - box.volume);
   }

   string toString() const {
      return format(" Объём:%s", volume);
   }
}

void main() {
   Box[] boxes;
   int j = 10;

   foreach (i; 0 .. 10) {
      boxes ~= Box(j*j*j);
      j = j-1;
   }

   writeln("Несортированный массив");
   writeln(boxes);
   boxes.sort;
   writeln("Сортированный массив");
   writeln(boxes);
   writeln(boxes[0]<boxes[1]);
   writeln(boxes[0]>boxes[1]);
   writeln(boxes[0]<=boxes[1]);
   writeln(boxes[0]>=boxes[1]);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Несортированный массив
[ Объём:1000,  Объём:729,  Объём:512,  Объём:343,  Объём:216,  Объём:125,  Объём:64,
Объём:27,  Объём:8,  Объём:1]
Сортированный массив
[ Объём:1,  Объём:8,  Объём:27,  Объём:64,  Объём:125,  Объём:216,  Объём:343,
Объём:512,  Объём:729,  Объём:1000]
true
false
true
false
```

Инкапсуляция

Все программы на языке D состоят из следующих двух основополагающих элементов:

- **Утверждения программы (код)** − Это часть программы, которая выполняет некоторые действия, и они называются функциями.
- **Данные программы** − Это информация в программе, которая влияет на функции программы.

Инкапсуляция − это концепция объектно-ориентированного программирования, которая связывает вместе данные и функции, которые управляют данными, и которая защищает и то, и другое от внешних помех и неправильного использования. Инкапсуляция данных привела к важной концепции ООП − **скрытию данных**.

Инкапсуляция данных − это механизм объединения данных, и использующих их функций, а **абстракция данных** − это механизм, при котором наружу выводятся только интерфейсы, а детали реализации скрываются от пользователя.

D поддерживает свойства инкапсуляции и скрытия данных посредством создания пользовательских типов, называемых **классами**. Мы уже изучили, что класс может содержать закрытые (**private**), защищённые (**protected**) и открытые (**public**) члены. По умолчанию все элементы, определённые в классе, являются открытыми. Например:

```
class Box {
   public:
      double getVolume() {
         return length * breadth * height;
      }
   private:
      double length;      // Длина коробки
      double breadth;     // Ширина коробки
      double height;      // Высота коробки
};
```

Переменные length, breadth и height являются закрытыми. Это означает, что к ним могут обращаться только другие члены класса Box, а не какая-либо другая часть вашей программы. Это один из способов инкапсуляции.

Чтобы сделать части класса открытыми (т.е. доступными для других частей вашей программы), вы должны объявить их после ключевого слова **public**. Все переменные или функции, определённые после спецификатора public, доступны из любых функций вашей программы.

В идеале нужно хранить как можно больше деталей каждого класса скрытыми от всех других классов.

Инкапсуляция данных в D

Любая программа на D, в которой вы реализуете класс с открытыми и закрытыми членами, является примером инкапсуляции данных и абстракции данных. Рассмотрим следующий пример:

Пример

```
import std.stdio;

class Adder {
   public:
      // конструктор
      this(int i = 0) {
         total = i;
      }

      // интерфейс во внешний мир
      void addNum(int number) {
         total += number;
      }

      //интерфейс во внешний мир
      int getTotal() {
         return total;
      };

   private:
      // данные, скрытые от внешнего мира
      int total;
}

void main( ) {
   Adder a = new Adder();

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);
   writeln("Всего ",a.getTotal());
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Всего 60
```

Вышеприведённый класс складывает числа вместе и возвращает сумму. Открытые члены **addNum** и **getTotal** являются интерфейсами для внешнего мира, и пользователь должен знать о них, чтобы использовать класс. Закрытый член **total** − это нечто, скрытое от внешнего мира, но необходимое для правильного функционирования класса.

Стратегия проектирования классов в D

Большинство из нас узнали через горький опыт, что члены класса нужно делать закрытыми по умолчанию, если нам действительно не нужно выводить их наружу. Это просто хорошая **инкапсуляция**.

Эта мудрость применяется чаще всего к данным-членам, но она одинаково применима ко всем членам, включая виртуальные функции.

Интерфейсы

Интерфейс − это способ заставить классы, которые наследуют от него, реализовать некоторые функции или переменные. Функции не должны реализовываться в интерфейсе, поскольку они всегда реализуются в классах, которые наследуются от интерфейса.

Я для себя понял (возможно, не совсем верно), что смысл интерфейсов в том, чтобы сделать некое приближение к запрещённому в языке D множественному наследованию (тоже самое есть в языке Java). Реализацию унаследовать не получится, а вот поведение – вполне. И полиморфизм с интерфейсами тоже действует. – прим. пер.

Интерфейс создается с использованием ключевого слова *interface*, вместо слова class, хотя они похожи во многих отношениях. Когда вы хотите унаследовать класс от интерфейса, и при этом класс уже наследуется от другого класса, вам нужно отделить имя класса от имени интерфейса запятой.

Давайте рассмотрим простой пример, в котором показано использование интерфейса.

Пример

```
import std.stdio;

// Базовый интерфейс
interface Shape {
   public:
      void setWidth(int w);
      void setHeight(int h);
}

// Унаследованный класс
class Rectangle: Shape {
   int width;
   int height;

   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }
      int getArea() {
         return (width * height);
      }
}

void main() {
   Rectangle Rect = new Rectangle();
   Rect.setWidth(5);
   Rect.setHeight(7);

   // Вывод площади объекта.
   writeln("Общая площадь: ", Rect.getArea());
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Общая площадь: 35
```

Интерфейс с функциями final и static

Интерфейс может содержать конечные (с ключевым словом final) и статические (со словом static) методы, реализация которых должна быть включена в сам интерфейс. Конечные функции не могут быть переопределены производным классом. Ниже приведен простой пример.

Пример

```
import std.stdio;

// Базовый интерфейс
interface Shape {
   public:
      void setWidth(int w);
      void setHeight(int h);

      static void myfunction1() {
         writeln("Это статический метод");
      }
      final void myfunction2() {
         writeln("Это конечный метод");
      }
}

// Унаследованный класс
class Rectangle: Shape {
   int width;
   int height;

   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }
      int getArea() {
         return (width * height);
      }
}

void main() {
   Rectangle rect = new Rectangle();

   rect.setWidth(5);
   rect.setHeight(7);

   // Вывод площади объекта.
   writeln("Общая площадь: ", rect.getArea());
   rect.myfunction1();
   rect.myfunction2();
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Общая площадь: 35
Это статический метод
Это конечный метод
```

Абстрактные классы

Абстракция относится к способности создавать абстрактные классы в ООП. Абстрактным классом является тот, который не может быть создан. Все остальные функциональные возможности класса всё еще существуют, и все его поля, методы и конструкторы доступны одинаковым образом. Просто вы не можете создать экземпляр абстрактного класса.

Если класс абстрактный, и его объект не может быть создан, класс не имеет большого смысла, если у него нет подклассов. Как правило, абстрактные классы возникают на этапе проектирования. Родительский класс содержит общую функциональность набора дочерних классов, но сам он является слишком абстрактным, чтобы его можно было использовать самостоятельно.

Использование абстрактных классов в D

Используйте ключевое слово **abstract** для объявления абстрактного класса. Это ключевое слово должно находиться в объявлении класса перед ключевым словом class. Ниже приведен пример того, как абстрактный класс можно унаследовать и использовать.

Пример

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

abstract class Person {
   int birthYear, birthDay, birthMonth;
   string name;

   int getAge() {
      SysTime sysTime = Clock.currTime();
      return sysTime.year - birthYear;
   }
}

class Employee : Person {
   int empID;
}

void main() {
   Employee emp = new Employee();
   emp.empID = 101;
   emp.birthYear = 1980;
   emp.birthDay = 10;
   emp.birthMonth = 10;
   emp.name = "Сотрудник1";

   writeln(emp.name);
   writeln(emp.getAge);
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Сотрудник1
37
```

Абстрактные функции

Подобно классам, функции также могут быть абстрактными. Реализация такой функции не предоставляется в своём классе, но должна предоставляться в классе, который наследует класс с абстрактной функцией. В вышеприведённый пример добавлена абстрактная функция.

Пример

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

abstract class Person {
   int birthYear, birthDay, birthMonth;
   string name;

   int getAge() {
      SysTime sysTime = Clock.currTime();
      return sysTime.year - birthYear;
   }
   abstract void print();
}
class Employee : Person {
   int empID;

   override void print() {
      writeln("Сведения о сотруднике указаны ниже:");
      writeln("ID сотрудника: ", this.empID);
      writeln("Имя сотрудника: ", this.name);
      writeln("Возраст: ",this.getAge);
   }
}

void main() {
   Employee emp = new Employee();
   emp.empID = 101;
   emp.birthYear = 1980;
   emp.birthDay = 10;
   emp.birthMonth = 10;
   emp.name = "Сотрудник1";
   emp.print();
}
```

Когда вы скомпилируете и выполните эту программу, она возвратит следующий результат:

```
Сведения о сотруднике указаны ниже:
ID сотрудника: 101
Имя сотрудника: Сотрудник1
Возраст: 37
```