FREE eBook

LEARNING D Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#d

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: d-language

It is an unofficial and free D Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official D Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with D Language

Remarks

D is a systems programming language with C-like syntax and static typing. It combines efficiency, control and modeling power with safety and programmer productivity.

Versions

Version	Changelog	Release Date
D	http://www.digitalmars.com/d/1.0/changelog.html	2007-01-23
D2	https://dlang.org/changelog/2.000.html	2007-06-17

Examples

Installation or Setup

The D programming language's standard compiler DMD can run on all major platforms. To install DMD see here. To install by command line you may run the command (found on the D website):

```
curl -fsS https://dlang.org/install.sh | bash -s dmd
```

Package Managers

Arch Linux

```
pacman -S dlang
```

Chocolatey

```
choco install dmd
```

Gentoo

```
layman -f -a dlang
```

OSX Homebrew

```
brew install dmd
```

Debian/Ubuntu

Installation on Debian/Ubuntu distributions needs that the APT repository be added to the sources list.

```
wget http://master.dl.sourceforge.net/project/d-apt/files/d-apt.list -O
/etc/apt/sources.list.d/d-apt.list
wget -qO - https://dlang.org/d-keyring.gpg | sudo apt-key add -
apt-get update
apt-get install dmd-bin
```

Other compilers

LDC is a D compiler thats uses the oficial DMD compiler frontend and LLVM as its backend.

GDC is a D compiler that uses the GCC backend to generate code.

IDEs

In order to make life easier you may also want to install an IDE (Integrated Development Environment). The D-Language Wiki has a list of available IDEs and Plugins for all Platforms here.

Hello World

```
import std.stdio;

// Let's get going!
void main()
{
    writeln("Hello World!");
}
```

To compile and run, save this text as a file called `main.d`. From the command line run `dmd main.d` to compile the program. Finally, run `./main` to execute the program in a bash shell or you can click on the executable on windows.

Hello, World!

To create the classic "Hello, world" printing program, create a file called `hello.d` with a text editor containing the following code :

```
import std.stdio;

void main() {
```

```
    writeln("Hello, World!");    //writeln() automatically adds a newline (\n) to the output
}
```

Explanation :

```
import std.stdio
```

This line tells the compiler that functions defined in the Standard Library module `std.stdio` will be used. Any module may be imported, as long as the compiler knows where to look for them. Many functions are provided as part of D's massive Standard Library.

```
void main() {
```

This line declares the function `main`, returning `void`. Note that unlike C and C++, D allows main to be of type `void`. The function `main` is special as it is the entry point of the program, i.e., this is where the execution of the program begins. A few notes about functions in general :

- A function's name can be anything that starts with a letter and is composed of letters, digits and underscores.
- Expected parameters will be a comma-separated list of variable names and their data types.
- The value that the function is expected to return can be any existing data type, and it must match the type of expression used in the return statement within the function.

The curly braces `{ … }` are used in pairs to indicate where a block of code begins and ends. They can be used in a lot of ways, but in this case they indicate where the function begins and ends.

```
writeln("Hello, World!");
```

`writeln` is a function declared in `std.stdio` that writes its agruments to `stdout`. In this case, its argument is `"Hello, World"`, which will be written to the console. Various format characters, similar to the ones used by C's `printf` may be used, like `\n`, `\r`, etc.

Every statement needs to be terminated by a semi-colon.

Comments are used to indicate something to the person reading the code and are treated like a blank by the compiler. In the code above, this is a comment:

```
//writeln() automatically adds a newline (\n) to the output
```

These are pieces of code that are ignored by the compiler. There are three different ways to comment in D :

1. `//` - Comment all text in the same line, after the `//`
2. `/* comment text */` - These are useful for multiline comments
3. `/+ comment text +` - These are also multiline comments

They are very useful to convey what a function / piece of code is doing to a fellow developer.

Compiling and Running the Program

To run this program, the code must fist be compiled into an executable. This can be done with the help of the compiler.

To compile using DMD, the reference D compiler, open a terminal, navigate to the the location of the file `hello.d` that you created and then run :

```
dmd hello.d
```

If no errors are found, the compiler will output an executable named after your source file. This can now be run by typing

```
./hello
```

Upon execution, the program will print out `Hello, World!`, followed by a newline.

Read values from a string

```
import std.format;

void main() {
    string s = "Name Surname 18";
    string name, surname;
    int age;
    formattedRead(s, "%s %s %s", &name, &surname, &age);
    // %s selects a format based on the corresponding argument's type
}
```

Official documentation for the format strings can be found at:
https://dlang.org/phobos/std_format.html#std.format

Read Getting started with D Language online: https://riptutorial.com/d/topic/1036/getting-started-with-d-language

Chapter 2: Associative Arrays

Examples

Standard use

```
int[string] wordCount(string[] wordList) {
    int[string] words;
    foreach (word; wordList) {
        words[word]++;
    }
    return words;
}

void main() {
    int[string] count = wordCount(["hello", "world", "I", "say", "hello"]);
    foreach (key; count.keys) {
        writefln("%s: %s", key, count[key]);
    }
    // hello: 2
    // world: 1
    // I: 1
    // say: 1

    // note: the order in the foreach is unspecified
}
```

Literals

```
int[string] aa0 = ["x": 5, "y": 6];  //int values, string keys
auto aa1 = ["x": 5.0, "y": 6.0];  // double values, string keys
string[int] aa2 = [10: "A", 11: "B"];  //string values, int keys
```

Add key-value pairs

```
int[string] aa = ["x": 5, "y": 6];
// The value can be set by its key:
aa["x"] = 7;
assert(aa["x"] == 7);
// if the key does not exist will be added
aa["z"] = 8;
assert(aa["z"] == 8);
```

Remove key-value pairs

Let's assume an associative array `aa`:

```
int[string] aa = ["x": 5, "y": 6];
```

Items can be removed by using `.remove()`, if key exits will be removed and `remove` returns `true`:

```
assert(aa.remove("x"));
```

if the given key does not exist `remove` does nothing and returns `false`:

```
assert(!aa.remove("z"));
```

Check if a key exist

```
int[string] numbers = ["a" : 10, "b" : 20];

assert("a" in numbers);
assert("b" in numbers);
assert("c" in numbers);
```

Read Associative Arrays online: https://riptutorial.com/d/topic/3159/associative-arrays

Chapter 3: Classes

Syntax

- class Foo { } // inherits from Object
- class Bar: Foo { } // Bar is a Foo too
- Foo f = new Foo(); // instantiate new objects on the heap

Remarks

See the specification, browse a book chapter on classes, inheritance and play interactively.

Examples

Inheritance

```
class Animal
{
    abstract int maxSize(); // must be implemented by sub-class
    final float maxSizeInMeters() // can't be overridden by base class
    {
        return maxSize() / 100.0;
    }
}

class Lion: Animal
{
    override int maxSize() { return 350; }
}

void main()
{
    import std.stdio : writeln;
    auto l = new Lion();
    assert(l.maxSizeInMeters() == 3.5);

    writeln(l.maxSizeInMeters()); // 3.5
}
```

Instantiation

```
class Lion
{
    private double weight; // only accessible with-in class

    this(double weight)
    {
        this.weight = weight;
    }
```

```
    double weightInPounds() const @property // const guarantees no modifications
    // @property functions are treated as fields
    {
        return weight * 2.204;
    }
}

void main()
{
    import std.stdio : writeln;
    auto l = new Lion(100);
    assert(l.weightInPounds == 220.4);

    writeln(l.weightInPounds); // 220.4
}
```

Read Classes online: https://riptutorial.com/d/topic/4695/classes

Chapter 4: Compile Time Function Evaluation (CTFE)

Remarks

CTFE is a mechanism which allows the compiler to execute functions at compile time. There is no special set of the D language necessary to use this feature - whenever a function just depends on compile time known values the D compiler might decide to interpret it during compilation.

You can also play interactively with CTFE.

Examples

Evaluate a function at compile-time

```
long fib(long n)
{
    return n < 2 ? n : fib(n - 1) + fib(n - 2);
}

struct FibStruct(int n) { // Remarks: n is a template
    ubyte[fib(n)] data;
}

void main()
{
    import std.stdio : writeln;
    enum f10 = fib(10); // execute the function at compile-time
    pragma(msg, f10); // will print 55 during compile-time
    writeln(f10); // print 55 during runtime
    pragma(msg, FibStruct!11.sizeof); // The size of the struct is 89
}
```

Read Compile Time Function Evaluation (CTFE) online:
https://riptutorial.com/d/topic/4694/compile-time-function-evaluation--ctfe-

Chapter 5: Contracts

Remarks

The assertions will be optimized away in an release build.

Examples

Function contracts

Function contracts allow the programer to check for inconsistencies. Inconsistencies include invalid parameters, checks for the correct return value or an invalid state of the object.

The checks can happen before and after the body of the function or method is executed.

```
void printNotGreaterThan42(uint number)
in {
    assert(number < 42);
}
body {
    import std.stdio : writeln;
    writeln(number);
}
```

The assertions will be optimized away in an release build.

Function contracts

For example if an method is invoked the state of the object may not allow that a method is called with specific parameters or not at all.

```
class OlderThanEighteen {
    uint age;

    final void driveCar()
    in {
         assert(age >= 18); // variable must be in range
    }
    body {
         // step on the gas
    }
}
```

Read Contracts online: https://riptutorial.com/d/topic/6830/contracts

Chapter 6: Dynamic Arrays & Slices

Syntax

- <type>[] <name>;

Remarks

Slices generate a new view on existing memory. They don't create a new copy. If no slice holds a reference to that memory anymore - or a sliced part of it - it will be freed by the garbage collector.

Using slices it's possible to write very efficient code for e.g. parsers that just operate on one memory block and just slice the parts they really need to work on - no need allocating new memory blocks.

Examples

Declaration and initialization

```
import std.stdio;

void main() {
    int[] arr = [1, 2, 3, 4];

    writeln(arr.length); // 4
    writeln(arr[2]); // 3

    // type inference still works
    auto arr2 = [1, 2, 3, 4];
    writeln(typeof(arr2).stringof); // int[]
}
```

Array operations

```
import std.stdio;

void main() {
    int[] arr = [1, 2, 3];

    // concatenate
    arr ~= 4;
    writeln(arr); // [1, 2, 3, 4]

    // per element operations
    arr[] += 10
    writeln(arr); // [11, 12, 13, 14]
}
```

Slices

```
import std.stdio;

void main() {
    int[] arr = [1, 2, 3, 4, 5];

    auto arr2 = arr[1..$ - 1]; // .. is the slice syntax, $ represents the length of the array
    writeln(arr2); // [2, 3, 4]

    arr2[0] = 42;
    writeln(arr[1]); // 42
}
```

Read Dynamic Arrays & Slices online: https://riptutorial.com/d/topic/2445/dynamic-arrays---slices

Chapter 7: Imports and modules

Syntax

- module my.package;
- import my.package;
- import my.package : function;
- import fancyName = mypackage;
- import my.package : fancyFunctionName = function;

Remarks

Modules automatically provide a namespace scope for their contents. Modules superficially resemble classes, but differ in that:

- There's only one instance of each module, and it is statically allocated.
- There is no virtual table.
- Modules do not inherit, they have no super modules, etc.
- Only one module per file.
- Module symbols can be imported.
- Modules are always compiled at global scope, and are unaffected by surrounding attributes or other modifiers.
- Modules can be grouped together in hierarchies called packages.

Modules offer several guarantees:

- The order in which modules are imported does not affect the semantics.
- The semantics of a module are not affected by what imports it.
- If a module C imports modules A and B, any modifications to B will not silently change code in C that is dependent on A.

Examples

Global imports

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

Multiple imports can either be specified in the same line, separated with a `comma` or in a new line.

```
import std.stdio, std.math;
import std.datetime;
void main()
```

```
{
    writeln("2^4: ", pow(2, 4));
    writeln("Current time: ", Clock.currTime());
}
```

Selective imports

Selective imports can help to cleanup the namespace and speed-up the compile-time even more, because the compiler only needs to parse the specific, selected functions.

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

Local imports

You can also import symbols in any scope, the import will only be looked up when the scope is needed (i.e. compiled) and the imported names will only be exposed in the imported scope. Most commonly the scope for local imports are functions, structs and classes.

```
void main()
{
    import std.stdio: writeln;
    writeln("Hello world");
}
// writeln isn't defined here
```

Public imports

Modules can be exposed to other modules with `public imports`.

```
public import std.math;
// only exports the symbol 'pow'
public import std.math : pow;
```

Renamed imports

A local name for an import can be given, through which all references to the module's symbols must be qualified with:

```
import io = std.stdio;
void main()
{
    io.writeln("Hello world");
    std.stdio.writeln("hello!"); // error, std is undefined
    writeln("hello!");           // error, writeln is undefined
}
```

Renamed imports are handy when dealing with very long import names.

Renamed and selective imports

Selective imports may also be renamed.

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

Module declaration

Modules have a one-to-one correspondence with source files. The module name is, by default, the file name with the path and extension stripped off, and can be set explicitly with the module declaration. The `ModuleDeclaration` sets the name of the module and what package it belongs to. If absent, the module name is taken to be the same name (stripped of path and extension) of the source file name.

```
module my.fancy.module;
```

Read Imports and modules online: https://riptutorial.com/d/topic/4344/imports-and-modules

Chapter 8: Loops

Syntax

- for (<initializer>; <loop condition>; <loop statement>) { <statements> }
- while (<condition>) { <statements> }
- do { <statements> } while (<condition>);
- foreach (<el>, <collection>)
- foreach_reverse (<el>, <collection>)

Remarks

- `for` loop in *Programming in D*, specification
- `while` loop in *Programming in D*, specification
- `do while` loop in *Programming in D*, specification
- `foreach` in *Programming in D*, opApply, specification

You can play with loops and foreach online.

Examples

For loop

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    for (int i = 0; i < arr.length; i++)
    {
        arr[i] *= 2;
    }
    writeln(arr); // [2, 6, 8]
}
```

While loop

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    int i = 0;
    while (i < arr.length)
    {
        arr[i++] *= 2;
    }
    writeln(arr); // [2, 6, 8]
}
```

do-while

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    int i = 0;
    assert(arr.length > 0, "Array must contain at least one element");
    do
    {
        arr[i++] *= 2;
    } while (i < arr.length);
    writeln(arr); // [2, 6, 8]
}
```

Foreach

Foreach allows a less error-prone and better readable way to iterate collections. The attribute `ref` can be used if we want to directly modify the iterated element.

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    foreach (ref el; arr)
    {
        el *= 2;
    }
    writeln(arr); // [2, 6, 8]
}
```

The index of the iteration can be accessed too:

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    foreach (i, el; arr)
    {
        arr[i] = el * 2;
    }
    writeln(arr); // [2, 6, 8]
}
```

Iteration in reverse order is possible too:

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    int i = 0;
    foreach_reverse (ref el; arr)
    {
        el += i++; // 4 is incremented by 0, 3 by 1, and 1 by 2
    }
```

```
    writeln(arr); // [3, 4, 4]
}
```

Break, continue & labels

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4, 5];
    foreach (i, el; arr)
    {
        if (i == 0)
            continue; // continue with the next iteration
        arr[i] *= 2;
        if (i == 2)
            break; // stop the loop iteration
    }
    writeln(arr); // [1, 6, 8, 5]
}
```

Labels can also be used to `break` or `continue` within nested loops.

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    outer: foreach (j; 0..10) // iterates with j=0 and j=1
        foreach (i, el; arr)
        {
            arr[i] *= 2;
            if (j == 1)
                break outer; // stop the loop iteration
    }
    writeln(arr); // [4, 6, 8] (only 1 reaches the second iteration)
}
```

Read Loops online: https://riptutorial.com/d/topic/4696/loops

Chapter 9: Memory & Pointers

Syntax

- &<variable> - access by reference (=gets the pointer to the data of the variable)
- *<variable> - deference operator (=gets the data object from a pointer)
- <type>* - data type that points to <type> (e.g. `int*)

Examples

Pointers

D is a system programming language and thus allows you to manually manage and mess up your memory. Nevertheless, D uses a garbage collector per default to free unused memory.

D provides pointer types T* like in C:

```
void main()
{
    int a;
    int* b = &a; // b contains address of a
    auto c = &a; // c is int* and contains address of a

    import std.stdio : writeln;
    writeln("a ", a);
    writeln("b ", b);
    writeln("c ", c);
}
```

Allocating on the heap

A new memory block on the heap is allocated using the `new` expression, which returns a pointer to the managed memory:

```
void main()
{
    int* a = new int;
    *a = 42; // dereferencing
    import std.stdio : writeln;
    writeln("a: ", *a);
}
```

@safe D

As soon as the memory referenced by a isn't referenced anymore through any variable in the program, the garbage collector will free its memory.

D also allows pointer arithmetic, except in code that is marked as `@safe`.

```
void safeFun() @safe
{
    writeln("Hello World");
    // allocating memory with the GC is safe too
    int* p = new int;
}

void unsafeFun()
{
    int* p = new int;
    int* fiddling = p + 5;
}

void main()
{
    safeFun();
    unsafeFun();
}
```

For more information about SafeD see the article from the D design team.

Read Memory & Pointers online: https://riptutorial.com/d/topic/6374/memory---pointers

Chapter 10: Ranges

Remarks

If a foreach is encountered by the compiler

```
foreach (element; range) {
```

it's internally rewritten similar to the following:

```
for (auto it = range; !it.empty; it.popFront()) {
    auto element = it.front;
    ...
}
```

Any object which fulfills the above interface is called an input range and is thus a type that can be iterated over:

```
struct InputRange {
    @property bool empty();
    @property T front();
    void popFront();
}
```

Examples

Strings and arrays are ranges

```
import std.stdio;

void main() {
    auto s = "hello world";
    auto a = [1, 2, 3, 4];

    foreach (c; s) {
        write(c, "!"); // h!e!l!l!o! !w!o!r!l!d!
    }
    writeln();

    foreach (x; a) {
        write(x * x, ", "); // 1, 4, 9, 16,
    }
}
```

Making a new Input Range type

The `InputRange` concept has three functions, example:

```
struct InputRange(T) {
```

```
    @property bool empty();
    @property T front();
    void popFront();
}
```

In short, a way to

1. check if the range is empty
2. get the current element
3. move to the next element

To make our own type a `InputRange`, we must implement these three functions. Let's take a look at the infinite sequence of squares.

```
struct SquaresRange {
    int cur = 1;

    @property bool empty() {
        return false;
    }

    @property int front() {
        return cur^^2;
    }

    void popFront() {
        cur++;
    }
}
```

See the D tour for an example with Fibonacci.

Read Ranges online: https://riptutorial.com/d/topic/3106/ranges

Chapter 11: Scope guards

Syntax

- scope(exit) - statements are called no matter how the current block was exited
- scope(success) - statements are called when the current block was exited normally
- scope(failure) - statements are called when the current block was exited through exception throwing

Remarks

Using scope guards makes code much cleaner and allows to place resource allocation and clean up code next to each other. These little helpers also improve safety because they make sure certain cleanup code is always called independent of which paths are actually taken at runtime.

The D scope feature effectively replaces the RAII idiom used in C++ which often leads to special scope guards objects for special resources.

Scope guards are called in the reverse order they are defined.

Play with scope guards or see an extensive tutorial.

Examples

Place allocation and cleanup code next to each other

Scope guards allow executing statements at certain conditions if the current block is left.

```
import core.stdc.stdlib;

void main() {
    int* p = cast(int*)malloc(int.sizeof);
    scope(exit) free(p);
}
```

Multiple, nested scopes

```
import std.stdio;

void main() {
    writeln("<html>");
    scope(exit) writeln("</html>");
    {
        writeln("\t<head>");
        scope(exit) writeln("\t</head>");
        "\t\t<title>%s</title>".writefln("Hello");
    } // the scope(exit) on the previous line is executed here
```

```
    writeln("\t<body>");
    scope(exit) writeln("\t</body>");

    writeln("\t\t<h1>Hello World!</h1>");
}
```

prints

```
<html>
    <head>
        <title>Hello</title>
    </head>
    <body>
        <h1>Hello World!</h1>
    </body>
</html>
```

Read Scope guards online: https://riptutorial.com/d/topic/4343/scope-guards

Chapter 12: Strings

Remarks

- Strings in D are immutable; use `.dup` to make a mutable `char` array if you want to edit in-place.

Examples

Reversing a string

`string` is defined as `alias string = immutable(char)[];`: so need to use `dup` to make a mutable char array, before it can be reversed:

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

int main() {

    string x = "Hello world!";
    char[] x_rev = x.dup.reverse;

    writeln(x_rev); // !dlrow olleH

    return 0;

}
```

Test for an empty or null string

Empty string

Empty string is not null but has zero length:

```
string emptyString = "";
// an empty string is not null...
assert(emptyString !is null);

// ... but it has zero lenght
assert(emptyString.length == 0);
```

Null string

```
string nullString = null;
```

a null string is null (De Lapalisse)

```
assert(nullString is null);
```

but, unlike C#, read the length of a null string does not generate error:

```
assert(nullString.length == 0);
assert(nullString.empty);
```

Test for empty or null

```
if (emptyOrNullString.length == 0) {
}

// or
if (emptyOrNullString.length) {
}

// or
import std.array;
if (emptyOrNullString.empty) {
}
```

Test for null

```
if (nullString is null) {
}
```

References

- What is the correct way to test for an empty string?
- Does D has C#'s string.Empty?

Convert string to ubyte[] and vice versa

String to immutable `ubyte[]`

```
string s = "unogatto";
immutable(ubyte[]) ustr = cast(immutable(ubyte)[])s;

assert(typeof(ustr).stringof == "immutable(ubyte[])");
assert(ustr.length == 8);
assert(ustr[0] == 0x75); //u
assert(ustr[1] == 0x6e); //n
```

```
assert(ustr[2] == 0x6f); //o
assert(ustr[3] == 0x67); //g
assert(ustr[7] == 0x6f); //o
```

String to `ubyte[]`

```
string s = "unogatto";
ubyte[] mustr = cast(ubyte[])s;

assert(typeof(mustr).stringof == "ubyte[]");

assert(mustr.length == 8);
assert(mustr[0] == 0x75);
assert(mustr[1] == 0x6e);
assert(mustr[2] == 0x6f);
assert(mustr[3] == 0x67);
assert(mustr[7] == 0x6f);
```

`ubyte[]` to string

```
ubyte[] stream = [ 0x75, 0x6e, 0x6f, 0x67];
string us  = cast(string)stream;
assert(us == "unog");
```

References

- DLang forum

Read Strings online: https://riptutorial.com/d/topic/5760/strings

Chapter 13: Structs

Examples

Defining a new Struct

To define the struct called Person with an integer type variable age, integer type variable height and float type variable ageXHeight:

```
struct Person {
    int age;
    int height;
    float ageXHeight;
}
```

Generally:

```
struct structName {
    /+ values go here +/
}
```

Struct Constructors

In D we can use constructors to initialize structs just like a class. To define a construct for the struct declared in the previous example we can type:

```
struct Person {
    this(int age, int height) {
        this.age = age;
        this.height = height;
        this.ageXHeight = cast(float)age * height;
    }
}

auto person = Person(18, 180);
```

Read Structs online: https://riptutorial.com/d/topic/4075/structs

Chapter 14: Templates

Syntax

- template identifier (TemplateParameterList) { ... }
- struct identifier (TemplateParameterList) { ... }
- class identifier (TemplateParameterList) { ... }
- ReturnType identifier (TemplateParameterList)(ParameterList) { ... }
- identifier!(TemplateInvocationList)

Examples

Function with one template

```
import std.stdio;

T min(T)(in T arg1, in T arg2) {
    return arg1 < arg2 ? arg1 : arg2;
}

void main() {
    //Automatic type inference
    writeln(min(1, 2));

    //Explicit type
    writeln(min!(ubyte)(1, 2));

    //With single type, the parenthesis might be ommited
    writeln(min!ubyte(1, 2));
}
```

template

An template can be introduced with `template`. It can contain functions and classes and other constructs.

```
template StaticArray(Type, size_t Length) {
     class StaticArray {
          Type content[Length];

          size_t myLength() {
            return getLength(this);
          }
     }

     private size_t getLength(StaticArray arr) {
          return Length;
     }
}

void main() {
```

```
    StaticArray!(int, 5) arr5 = new StaticArray!(int, 5);
    import std.stdio;
    writeln(arr5.myLength());
}
```

Read Templates online: https://riptutorial.com/d/topic/3892/templates

Chapter 15: Traits

Syntax

- __traits (TraitsKeyword, TraitsArguments...)

Examples

Iterating over the members of a struct

```
import std.stdio;

struct A {
    int b;
    void c();
    string d;
};

void main() {
    // The following foreach is unrolled in compile time
    foreach(name; __traits(allMembers, A)) {
        pragma(msg, name);
    }
}
```

The `allMembers` traits returns a tuple of string containing the names of the members of the given type. These strings are known at compile time.

Iterating over members of a struct/class without their inherited members

```
module main;

auto getMemberNames(T)() @safe pure {
    string[] members;

    foreach (derived; __traits(derivedMembers, T)) {
        members ~= derived;
    }

    return members;
}

class Foo {
    int a;
    int b;
}

class Bar : Foo {
    int c;
    int d;
    int e;
}
```

```
void main() {
    import std.stdio;

    foreach (member; getMemberNames!Bar) {
        writeln(member);
    }
}
```

derivedMembers returns a tuple of string literals, where each string is the member name.

The example outputs:

```
c
d
e
```

Read Traits online: https://riptutorial.com/d/topic/3416/traits

Chapter 16: UFCS - Uniform Function Call Syntax

Syntax

- aThirdFun(anotherFun(myFun(), 42); // common notation (also valid)
- myFun().anotherFun(42).aThirdFun(); // UFCS
- myFun.anotherFun(42).aThirdFun; // empty braces can be removed

Remarks

In a call `a.b(args...)`, if the type `a` does not have a method named `b`, then the compiler will try to rewrite the call as `b(a, args...)`.

Examples

Checking if a Number is Prime

```
import std.stdio;

bool isPrime(int number) {
    foreach(i; 2..number) {
        if (number % i == 0) {
            return false;
        }
    }

    return true;
}

void main() {
    writeln(2.isPrime);
    writeln(3.isPrime);
    writeln(4.isPrime);
    5.isPrime.writeln;
}
```

UFCS with ranges

```
void main() {
    import std.algorithm : group;
    import std.range;
    [1, 2].chain([3, 4]).retro; // [4, 3, 2, 1]
    [1, 1, 2, 2, 2].group.dropOne.front; //  tuple(2, 3u)
}
```

UFCS with Durations from std.datetime

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

void some_operation() {
    // Sleep for two sixtieths (2/60) of a second.
    Thread.sleep(2.seconds / 60);
    // Sleep for 100 microseconds.
    Thread.sleep(100.usecs);
}

void main() {
    MonoTime t0 = MonoTime.currTime();
    some_operation();
    MonoTime t1 = MonoTime.currTime();
    Duration time_taken = t1 - t0;

    writeln("You can do some_operation() this many times per second: ",
            1.seconds / time_taken);
}
```

Read UFCS - Uniform Function Call Syntax online: https://riptutorial.com/d/topic/4155/ufcs---uniform-function-call-syntax

Chapter 17: Unittesting

Syntax

- unittest { ... } - a block that is only run in "unittesting" mode
- assert(<expression that evaluates to a boolean>, <optional error message>)

Examples

Unittest blocks

Tests are an excellent way to ensure stable, bug-free applications. They serve as an interactive documentation and allow to modify code without fear to break functionality. D provides a convenient and native syntax for `unittest` block as part of the D language. Anywhere in a D module `unittest` blocks can be used to test functionality of the source code.

```
/**
Yields the sign of a number.
Params:
    n = number which should be used to check the sign
Returns:
    1 for positive n, -1 for negative and 0 for 0.
*/
T sgn(T)(T n)
{
    if (n == 0)
        return 0;
    return (n > 0) ? 1 : -1;
}

// this block will only be executed with -unittest
// it will be removed from the executable otherwise
unittest
{
    // go ahead and make assumptions about your function
    assert(sgn(10)  ==  1);
    assert(sgn(1)   ==  1);
    assert(sgn(-1)  == -1);
    assert(sgn(-10) == -1);
}
```

Executing unittest

If `-unittest` flag is passed to the D compiler, it will run all unittest blocks. Often it is useful to let the compiler generate a stubbed `main` function. Using the compile & run wrapper `rdmd`, testing your D program gets as easy as:

```
rdmd -main -unittest yourcode.d
```

Of course you can also split this process into two steps if you want:

```
dmd -main -unittest yourcode.d
./yourcode
```

For `dub` projects compiling all files and executing their unittest blocks can be done conveniently with

```
dub test
```

Pro tip: define `tdmd` as shell alias to save tipping.

```
alias tdmd="rdmd -main -unittest"
```

and then test your files with:

```
tdmd yourcode.d
```

Annotated unittest

For templated code it is often useful to verify that for function attributes (e.g. `@nogc` are inferred correctly. To ensure this for a specific test and thus type the entire unittest can be annotated

```
@safe @nogc pure nothrow unittest
{
    import std.math;
    assert(exp(0)  ==  1);
    assert(log(1)  ==  0);
}
```

Note that of course in D every block can be annotated with attributes and the compilers, of course, verifies that they are correct. So for example the following would be similar to the example above:

```
unittest
{
    import std.math;
    @safe {
        assert(exp(0)  ==  1);
        assert(log(1)  ==  0);
    }
}
```

Read Unittesting online: https://riptutorial.com/d/topic/6201/unittesting

Credits

S. No	Chapters	Contributors
1	Getting started with D Language	Alessio Sacco, André Puel, Bauss, Bennet Leff, Cauterite, Community, EsmaeelE, Gassa, Sirsireesh Kodali, Some coder, T.Furholzer, TuxCopter
2	Associative Arrays	Bauss, greenify, o3o, Shriken
3	Classes	greenify, o3o
4	Compile Time Function Evaluation (CTFE)	André Puel, greenify
5	Contracts	Quonux
6	Dynamic Arrays & Slices	André Puel, Bauss, greenify, Harry, Shriken
7	Imports and modules	greenify
8	Loops	greenify, o3o
9	Memory & Pointers	greenify
10	Ranges	André Puel, Cauterite, Shriken
11	Scope guards	André Puel, greenify
12	Strings	Harry, o3o, RamenChef
13	Structs	Bennet Leff
14	Templates	André Puel, Bauss, Quonux
15	Traits	André Puel, Bauss
16	UFCS - Uniform Function Call Syntax	André Puel, greenify, tre0n
17	Unittesting	greenify