Table of Contents

Introduction

What is D?

- D is a high level programming language for applications and systems programming.
- D is a compiled language like C/C++, no bytecode like in Java, no interpreter like in Perl.
- D is targetted to iron out several design flaws of C/C++. You can think of D as an upgraded version of C++, as the syntax is fairly similar - although it is probably more than that.

If you want to know more about the philosophy behind the D language, look at Walter Bright's explanations: http://www.digitalmars.com/d/overview.html

To do: Add more of the D philosophy here because it is really great :)

A simple tutorial

This is a step-by-step tutorial to write your first application in D.

If you are already experienced in writing programs for another language you should just read the "Requirements" page and then continue to read the next section.

Requirements

This tutorial is currently available for Windows with DMD compiler and SkyOS with GDC.

Downloading and installing the DMD compiler under Windows

To do: a full section with an installation guide

First you have to download the latest version of the D compiler from http://www.digitalmars.com//d/dcompiler.html#Win32 - download the dmd.zip and dmc.zip files. DMD uses the DMC linker and make tool for building.

Unpack the dmd.zip file to a temporary directory. It contains two folders: "dm" and "dmd". I don't really know why there are two directories - imo you can safely delete the dm directory because it only contains files that we're going to extract from dmc.zip.

Extract the files dm/bin/link.exe from the dmc.zip to dmd/bin/link.exe, extract the dm/lib/snn.lib to dmd/lib/snn.lib.
The dmd/bin/sc.ini file contains some settings for the DMD compiler, among others the path to the link.exe. Edit it with you favourite text editor from:

```
[Version]
version=7.51 Build 020

[Environment]
LIB="%@P%\..\lib";\dm\lib
DFLAGS="-I%@P%\..\src\phobos"
LINKCMD=%@P%\..\..\dm\bin\link.exe
```

to

```
[Version]
version=7.51 Build 020

[Environment]
LIB="%@P%\..\lib"
DFLAGS="-I%@P%\..\src\phobos"
LINKCMD=%@P%\link.exe
```

so the compiler knows where to find your extracted "link.exe".

Now you're done, you can move the dmd directory anywhere you want to, perhaps you would want to move it to your programs directory and rename it to "Digital Mars D compiler" or anything. If you want to easily access the dmd compiler, set the Windows path to the bin directory:

To Do: Verify this on an english Windows (98/ME/2000/XP and so on) because i only have german Windows XP
Right-click on "My computer", click on Properties, go to the advanced tab, click the "Enviroment Variables" button. Select your "Path" Variable in the list box, click on edit. Add a semicolon when it's not already there, then add the Path to the bin directory (for example: "...;C:\Program Files\DMD\bin"). Quit all the dialog boxed with a click on "OK".

Now you should be able to run dmd.exe in your command shell:

- Open the command shell: Start->Run, enter cmd (command on Windows 95/98/ME) and press enter
- Enter "dmd" and press enter
- -> You should see the following:

```
Digital Mars D Compiler v0.110
Copyright (c) 1999-2004 by Digital Mars written by Walter Bright
Documentation: www.digitalmars.com/d/index.html
Usage:
  dmd files.d ... { -switch }

  files.d        D source files
  -c             do not link
  -d             allow deprecated features
  -g             add symbolic debug info
  -gt            add trace profiling hooks
  -v             verbose
  -O             optimize
  -odobjdir      write object files to directory objdir
  -offilename    name output file to filename
  -op            do not strip paths from source file
  -Ipath         where to look for imports
  -Llinkerflag   pass linkerflag to link
  -debug         compile in debug code
  -debug=level   compile in debug code <= level
  -debug=ident   compile in debug code identified by ident
  -inline        do function inlining
  -release       compile release version
  -unittest      compile in unit tests
  -version=level compile in version code >= level
  -version=ident compile in version code identified by ident
```

Downloading and installing the GDC compiler under SkyOS

The first you have to do is downloading the GDC. It calls "Unstable SkyOS package" at this time. But works good!

The most difficult would be done:)
By default, Firefox save it under your home folder e.g. "boot/home/admin/".
Go to your home folder and simple click on the gdc.pkg. The Package file will do the rest.

Now you should be able to run gdc in your bash:
- Open your bash.
- Type cd /boot/programs/gdc/bin.
- Call ./gdc Usage.

Now your D compiler is ready!

A first program in D!

Fire up your favourite text editor! (There are a lots of text editors that support D or can be set up to handle D).
To do: A page with all the text editor support, link to it from here :)

Type (or copy-and-paste) the following code:

```d
int main(char[][] args)
{
  return 0;
}
```

Save this file as "test.d" wherever you want to save your programming projects :)

You have to compile the program with DMD before you can execute it. For such small programs it is totally sufficient to write a batch file that compiles your program. (For larger programs you

will use a build tool or Makefiles if you want to edit them a lot).
Just create a new text file named "test.bat" and type those two lines:

```
@dmd test.d
@pause >nul
```

The first line evecutes the d compiler, the second line will wait for you to press enter, so the window stays open and you can read the output of the compiler (The "@" p. Save the file to the same directory as "test.d" and execute it. This is what my computer says:

```
C:\D\bin\link.exe test,,,user32+kernel32/noi;
```

This means the code was compiled successfully. You should now have some new files:

- **test.exe** - your program in executable format
- **test.map** - an auxilary file (can be viewed with a text editor)
- **test.obj** - the compiled but not linked version of the program

You can safely delete any of these three files - the compiler will generate them again when you compile your program. Now you can run your test.exe - if you just double-click it a console window will be opened and almost immediately closed again because the program has already finished. (It's fast, isn't it? :)

If you want to see the output, just write another batch file "run-test".bat":

```
@test.exe
@pause>nul
```

(Almost the same thing as the test.bat)

When you double-click it you should see an empty window, when you press enter the window is closed.

Congratulations, you have written your first D program! It simply does nothing else than immediately exit after you start it, but it works - and it is fast :)

Understanding the program

Well the program does nothing. What are all the lines for?

Actually the program does a lot of things, but these things are not visible.
The program has to initialise before it jumps right into your code, so you won't have to bother with the garbage collector and other "complicated stuff".

When the program is done initializing whatever it needs to initialize it executes the "main" function - and that is what you have written in "test.d".

```
int main(char[][] args)
```

This line says: This is the main function, it returns an "int" when called, and when you want to call it you have to give it a "char[[" which will be named "args". This is actually an array of array of chars - or an array of strings if you want to call it so. These strings are the arguments you called you program with. The result "int" of the function is the exit code of your program, it is commonly used to signal success or failure of the program (0 means success, not 0 means there was some kind of error). This return value can be used e.g. in batch files or by other programs that execute you program.

```
{
  return 0;
}
```

This is the function body. The curly braces are there to group the commands together that belong to the function.
"return" is a keyword that means "exit the function here and return something" - and we want to return 0.
Each statement should be ended with a semicolon.

Hello world!

Now we're going to extend the program. Open the "test.d" again and edit it to:

```
import std.stdio;

int main(char[][] args)
{
  writef("Hello world!");
  return 0;
}
```

"import" tells the compiler that you are using another module from the standard library. The "std.stdio" module contains some really useful functions for input and ouput in your program. One of these functions is the writef function that is called in your main. It takes a string as an argument here: The string is "Hello world!". The function will print that string to the console.

Compile and execute the program with test.bat and run-test.bat. The output should be:

```
Hello world!
```

Switching to D

This section aims to help programmers of other languages with understanding D.

Some general notes on the D language:

- For a list of supported Editors, take a look at this page: http://www.prowiki.org/wiki4d/wiki.cgi?EditorSupport
- Debugging D is problematic. Most users seem to prefer adding some printf/writef to their code instead of trying to set up gdb, Visual Studio or WinDBG to debug D code
- D should be pretty platform-independant as long as you do not use system-specific features (e.g. inline assembler)
- There is linux support. There is even a gcc-Frontend for D.

From C/C++

You may already have noticed that the D syntax is similar to C syntax so this should be easy to get used to.
D is also just as powerful as C++, it just dropped some really problematic things and added some great new things.

Major additions to C++ include:

- Garbage collection. Don't worry about your objects :)
- Arrays with known length, array slicing, array concatenation
- Class syntax and semantics resemble more the way it's done in Java than C++
- Less (!) pointers. You could say almost no more pointers - but you can still use them if you want to!
- Strings in the language core, strings are NOT zero terminated!

Things that were dropped:

- No more header files. Everything that belongs together should be in one file (ie "module").
- No more preprocessor. You will perhaps miss it at the beginning.
- No forward declarations. (I don't really know if this is 100% true)
- The "->" operator was completely replaced by "."
- Implicit casts - you have to cast explicitly - it's better that way.

Things that were changed and that you have to be aware of:

- You can't define and declare structs and class in just one statement. So you don't need to type a semicolon after struct/class anymore. Example:

```
struct Foo
{
  int x, y;
}
Foo foo;
```

- The constructor and destructor are always called "this" and "~this". The base class' constructor can be called with "super":

```
class Foo
{
  this()
  {
    // Do something

    super();
    // Do something

  }
  ~this()
  {
    // Blah

  }
```

```
}
```

- Comparing class references to "null" is done with the "is" operator, because the "==" can be overloaded!

```
if (myfoo is null)
{
  //...

}
```

From Java

Switching from Java to D is not a big step. There are a few differences because D is compiled for native code. As both languages are somehow influenced by or inherited from C or C++ syntax, there won't be a big problem :-)

Let me show you some D code that works and is very similar to java code:

```
// a class called Foo

class Foo
{
  // a variable

  int myInteger;

  // this is the constructor!!

  this()
  {
    myInteger = 1;
  }

  // a function :-)

  int bar()
  {
    for (ubyte i=0; i<2; i++) {
      myInteger *= 2;
    }
    return myInteger;
  }

} // end of class Foo


void main()
{
  // creates a new class of Foo

  Foo myFoo   = new Foo();

  // calls the function bar in foo three times.

  writefln("%d",myFoo.bar());
  writefln("%d",myFoo.bar());
  writefln("%d",myFoo.bar());
}
```

Output:

```
4
16
64
```

Difference: In D, we don't have to use a class name that is same name as the file and we import **std.stdio** to write to the display:

```
// Import standard-in-out

import std.stdio;

// Main function

int main(char[][] args)
{
  char[] world() {
    return "world!";
  }
  writefln("Hello "~world());

  // We have to return with a int to exit the program (0 means "no error").

  return 0;
}
```

Difference: The **main** function must return an int to the system.

Difference: In D "+" and "~" are not the same! The first one (plus) is only for numbers (innt b=5+3, int a=2+b) and the concatenation ("~") is only for arrays (which strings are):

```
char[] str   = "Paradise";
char[] text  = "Welcome to "~str~".";
```

See also:
Language Reference, An advanced tutorial: File read and write, Java to D

From Pascal/Delphi

You will have to get used to the C syntax and the C way of doing things. The C/C++/D syntax looks a bit weird and full of special characters to the human eye :) but it is very efficient - your fingers will be pleased. (If you want to go to extremes, learn Perl *g*).

The "import" syntax is very similar to "uses" in Pascal, modules are like units and strings work almost the same way as in Delphi: you won't have to worry about string length and zero termination unless your interfacing to one of the "lesser" (*g*) languages.

To do: Add a lot more of comparison to Pascal and Delphi, keep on laughing at C/C++ people for not having binary-safe strings!

From PHP :)

I often use PHP. No, not only for websites, i write a lot of networking hacks in it, e.g. IRC-Bots and stuff. So i came up with the idea to "compare" PHP and D.

First and most important: D is a compiled language, PHP is interpreted.
Although PHP people may think PHP is a pretty fast interpreter because code is pre-compiled and cached and... - D is always faster. I even believe that D code could be faster when you add up compile and execute time.

D is strictly typed. PHP isn't. You wont be able to mulitply your string "45" with integer 3.
You also have to declare variables before using them, and they will only hold one certain type.

D has a lot more complex object orientation. Learn about that soon (Templates and stuff). You may even be able to compensate the type problem with that as you could write a class that can do anything a PHP variable can do.

PHP's got a really great standard library, thats why i have to write functions only for really

specialized stuff. D's standard library, Phobos, isn't that large, but there are tons of libs from other people around. Take a look at http://dsource.org/ for example.

An advanced tutorial: File read and write

This example is not as easy as "Hello World", but it gives a clue on how D programs look like.

Let's begin with a new project, called "fio.d", which checks the existance of a file and writes it to the output:

```
// Import some needed modules.

import std.stdio; // Module for input/output on the console

import std.file;  // Module for file i/o


// The "main" function: the program starts here.

int main(char[][] args)
{
  // A variable array of char (= string) with the name of the file.

  char[]  filename  = "text.txt";

  return 0;

}
```

The first function we need is called **exists**. It is located in the "std.file" module and can be used like this:

```
if (exists(filename)!=0) {
  /* File exists! */
} else {
  /* File does not exist! */
}
```

To read content from a file, we can use the function **read** and to get it's size, we use **getSize**, both are declared in "std.file" again.

To write to the console, we use the function **writefln**, which takes any number of arguments, including ones containing formatting codes.

```
// Import

import std.stdio;
import std.file;

// Main function

int main(char[][] args)
{
  char[]  filename  = "text.txt";
  if (exists(filename)!=0) {
    uint size       = getSize(filename);
    writefln("%s", cast(char[])read(filename));
    writefln("");

    writefln("Size of file = %d (bytes).",size);

  } else {

    writefln("File not found. Please try again later :-)");
    /* File does not exist! */
  }
  return 0;
}
```

That's all the trick about displaying the content. Well not quite... the line

```
writefln("%s", cast(char[])read(filename));
```

needs a bit more explanation. The **read** function reads in the content of the file into a dynamic array of bytes; a **byte[]** type. However, the **writefln** does not know how to display an array of arbitary bytes. It does know how to display character strings though and an array of ASCII bytes is the same as a character string. So if your file only contains ASCII characters, you can tell D that the **read** function is returning a character array (a.k.a. a string). You do this by the **cast** syntax.

```
cast( char[] )
```

Also note that the first argument to the **writefln** call is a string containing a single formatting code *%s*. This tells writefln that the next argument is to be displayed as a text string. If you didn't have this formatting code, and the text file contained formatting codes, **writefln** would crash as it would be expecting more arguments to match the formatting codes embedded in the text file.

Now we will add one thing: If there is no file, the program should create one. We do this by calling **write**:

```
write(filename,"Well done!\n"~
      "The file "~filename~" was written and read without problems.\n\n"~
      "If you did understand this tutorial, the time has come to\n"~
      "get  in  touch  with  the  'Language  Reference'  or  the\n"~
      "'standard runtime library', also called \"phobos\".");
```

Replace the comment in the code above ("/* File does not exist! */") with this call to the write function, compile and test it :-)

See also:
std.stdio, std.file

Language Reference

This is the reference for the core language of D.

General syntax

All files in D are called "modules".

A module holds one or more declarations for functions, structs, classes, templates, and variables.
You can't write a class that spans over more than one file, nor can you do that with functions or structs.

Here is an example for a D module (actually it's the main module of a tetris clone i wrote), just to get the look and feel of the language:

```
private import std.loader;

private import derelict.sdl.sdl;
private import derelict.sdl.image;
private import derelict.sdl.mixer;

private import msgstr;

private import surface;
```

```
private import videoinfo;
private import gameresources;

private import options;
private import menuscreen;
private import game;

Surface SetVideoMode(uint x, uint y, uint bpp, bool fullscreen)
{

  Uint32 flags = 0; //SDL_DOUBLEBUF;

  if (fullscreen)
    flags |= SDL_FULLSCREEN;

  if (fullscreen)
    msg_str("Setting full screen video mode");
  else
    msg_str("Opening windowed screen");

  Surface screen = surface.Surface.SetVideoMode(x, y, bpp, flags);
  msg_ok();

  VideoInfo vi = new VideoInfo();
  vi.PrintProperties();

  SDL_WM_SetCaption("Tetris - written 2004 by MD",null);
  return screen;
}


int main(char[][] args)
{
  std.loader.ExeModule_Init();

  msg_str("Loading Derelict SDL");
  try { DerelictSDL_Load(); } catch (Exception e) { msg_end("NO DLL"); return 1; }
  msg_ok();

  msg_str("Loading Derelict SDL_image");
  try { DerelictSDLImage_Load(); } catch (Exception e) { msg_end("NO DLL"); return 1; }
  msg_ok();

  msg_str("Loading Derelict SDL_mixer");
  try { DerelictSDLMixer_Load(); } catch (Exception e) { msg_end("NO DLL"); return 1; }
  msg_ok();

  msg_str("Init SDL Audio and Video");
  if (SDL_Init(SDL_INIT_AUDIO | SDL_INIT_VIDEO) == 0)
  {
    msg_ok();

    Options o = new Options("resource", "options.txt");

    Surface screen = SetVideoMode(o.VideoModeX, o.VideoModeY, o.VideoModeBPP,
o.Fullscreen);
    surface.Surface.PrintProperties(screen.sdl_surface);
    GameResources gr = new GameResources(o);

    while (1)
    {
      MenuScreen menuscreen = new MenuScreen(screen, gr);
      uint m = menuscreen.Run();
      delete menuscreen;

      if (m == 0) break;

      switch (m)
      {
        case 1:
          Game game = new Game(screen, gr);
          game.Run(1);
          delete game;
          break;

        case 2:
          Game game = new Game(screen, gr);
          game.Run(2);
          delete game;
          break;
      }
    }
```

```
    msg_str("Waiting for sound to finish playing");
    while (gr.sound.SoundPlaying)
    {
      // Do nothing, just wait :)

      SDL_Delay(50);
    }
    msg_ok();

    delete gr;
    delete screen;

    msg_str("Shutting down SDL");
    SDL_Quit();
    msg_ok();

  } else
    msg_err();

  return 0;
}
```

Statements

Statements are all kinds of thing that can appear in your code block, for example in a function body.

Expression Statements

An expression statement consists just of any expression. Basically, expressions are:

- Function calls ("foo()")
- Assignments ("=", "+=", ...), see operators
- Comparations ("==", "<", "is", ...), see operators
- "Mathematical", boolean and unary expressions ("a + b", "a - b", "a && b" ...), see operators
- Conditional expressions ("foo ? "a" : "b")
- Literal values (like 8 or the string "abc"), see literal values
- Variables, see variable types
- Some special keywords ("this" or "super()")

Examples:

```
x = y;
foo();
a = (b < c) ? b : c;
```

Special cases:
There are some expression statements that need some more explanation about what they do and - that`s the point - in which order they do this.

Examples:

```
i += ++i;
i += i++;
```

What`s the difference between those two statements?

- Well, the first one increments ***i*** and adds the result (arithmetically) to the *old* ***i***. (**pre**-increment)
- The second statement adds ***i*** itself and increments the result. (**post**-increment)

(In this case there is no difference in the result, but allways take a look at what you need!)

```
while ( (++i) < j )
  statement;

while ( (i++) < j )
  statement;
```

As before explained, there is a big difference between pre-incrementals and post-incrementals inside the condition of a while-loop.

- First example increments ***i*** **before** the comparison,
- second one increments ***i*** **after** the comparison.

With the first example you can avoid using while-statements like:

```
while( i < j )
  i++;
```

Special Keywords

See also:
Classes

- "*this*" is a pointer to current object (pointer gets dereferenced in statements afaik - so *x = this;* would copy the object) and the 'name' for constructors.

```
class foo
{
  this() { /* ... */ }

  foo bar() { return this; }
}
```

- "super()" calls the constructor of the base-class:

```
class Foo
{
  this() { /* ... */ }

  /* ... */
}

class Bar : Foo
{
  this() { super(); }   // calls Foo.this()

}
```

Conditional expressions
Conditional expressions are expressions that are evaluated on boolean purpose; wether something is true or false for example. They're used mostly in control structures, such as while loops or if constructs. For conditional expressions there exist special operators, like AND (**&&**) and OR (**||**).

```
if(conditional expression)
{
  // code to execute

}
```

You are not limitted to only using conditional operators in conditional expressions; in fact, sometimes it's useful to use a normal operator in a conditional expression. What is true though, is that for a true conditional expression, the eventually evaluated value will be 1 (true) or 0 (false). Any other value than 0 will be interpreted as being true.

== Equals, != Not equals

```
1 == 2; // evaluates into FALSE (0)

1 == 1; // evaluates into TRUE

1 != 2; // evaluates into TRUE
```

&& And, || Or

```
(1 == 1) && (2 == 2); // evaluates into TRUE

(1 == 1) && (1 == 2); // evaluates into FALSE

(1 == 1) || (1 == 2); // evaluates into TRUE

(1 == 2) || (3 == 4); // evaluates into FALSE
```

To do: Explain conditional expressions and special keywords here, maybe a bit of expression evaluation and special cases (i += ++i;) too.

If statement

```
if ( /* boolean expression*/ )
  statement;
// Optional:

else
  statement;
```

"If" executes the next statement only if the boolean expression evaluates to "true". If there is an "else" block, it gets executed if the expression evaluates to "false".
The statements that are executed are often block statements.

Example:

```
if (i == 4)
{
  writef("i is equal to 4");
} else {
  writef("i is not equal to 4");
}
```

It is also possible to make multiple constructs of if-else, they work as you expect them to work:

```
if(i == 4)
{
  // code for i == 4

}
else if(i == 5)
{
  // code for i == 5

}
else
{
  // code for any other i

}
```

While statement

```
while ( /* boolean expression */ )
  statement;
```

or:

```
while ( /* boolean expression */ )
{
  statement1;
  statement2;
  // ...

}
```

Executes the statement as long as the boolean expression is true. If the expression is false from the beginning, it will not execute any statements. If you want the loop to execute your statements at least once, use the Do-While Statement.

Example:

```
int i = 0;
while (i < 4)
{
  writefln("i = %d", i);
  i++;
}
```

Output:

```
i = 0
i = 1
i = 2
i = 3
```

Switch-Statement

The switch-statement provides a more structured way of catching multiple conditions in a single block.

```
switch( /* variable */ )
{
// if ( variable == 0)

  case 0:
    statement;
    break;

// if ( (variable == 1) || (variable == 2) || (variable == 3) || (variable == 4) )

  case 1,2,3,4:
    statement;
    break;

// if (! (std.string.strcmp(variable,"hello") || std.string.strcmp(variable,"world")) )

  case "hello", "world":
    statement;
    break;

// "else"

  default:
    statement;
    break;
}
```

The only 'disadvantage' of switch-statements is the need of constant values in the case-conditions. Apart from that you can use every (constant) type which is implicity convertible to the variable you want to evaluate.

Do-While Statement

The do-while-statement has a quite similar functionality to the normal while-statement.

```
do
{
  statement;
} while ( /* boolean expression */ );
```

The only difference is that the condition is evaluated *after* executing the statement inside the

brackets. Thus, the statement is executed at least once.

Example:

```
int i = 1;
do
{
  writefln("Executed the statement with i = %d", i);
} while ( i < 1 );
```

Output:

```
Executed the statement with i = 1
```

Opposed to:

```
int i = 1;
while ( i < 1 )
{
  writefln("Executed the statement with i = %d", i);
}
```

Output:

```

```

(Doesn't output anything because i is 1 at the first check!)

For statement

```
for ( /* initializer */ ; /* test */ ; /* increment */ )
  statement;
```

The for loop is commonly used for executing a statement a certain number of times, but as it's syntax is very flexible it can be used for almost all kinds of loops.

It works by first executing the initializer, which can also be used to declare new variables - variable declared here have the scope of just that for loop, what means they will be available only during the *test*, the *increment* and the *statement*.

Second, it evaluates the *test* statement, and if it is true it executes the loop statement, followed by the *increment* statement.

This may sound a bit complex at first, but it's kind of simple. Just look at that code:

```
for ( int i = 0; i < 100; i++ )
{
  writef("i is now %d\n", i);
}
```

Note: The writef function just outputs the string and replaces %d by the value of i.

You can read that as: Set i to zero, count i up as long as it is smaller than 100 and execute the "writef..." block for each i.
If you execute this code it will print "i is now 0", "i is now 1", "i is now 2" and so on until "i is now 99". It'll never print "i is now 100" because of the test statement that says that i should be smaller than 100.

The for loop is equivalent to:

```
{
```

```
  /* initializer */
  while ( /* test */ )
  {
    statement;
    /* increment */
  }
}
```

Therefore you can abuse it as a while loop when you just leave the initializer and the increment empty.

Of course you can use the for loop in lots of other ways. Look at this:

```
char[] str = "aaaaaabcdef";
for (int i = 0; str[i] == 'a'; i++)
{
}
```

Note: This code is "unsafe" as it doesn't check array bounds! If there are just 'a'-characters in the string it'll try to read beyond the end of the string!

This says: set i to 0, and as long as the i-th character in str is an 'a' just increment i. This will search your string for the first non-'a' character. An empty loop statement is allowed as long as you use the curly braces, i.e. you can not just write:

```
for (int i = 0; str[i] == 'a'; i++)
  ;  /* illegal !!! */
```

Foreach Statement

Foreach scans an array or hash. This means you could apply code for every element in an array or hash. It does this with a reference. It is also optional to store the refering index number or string (dependant on the hash key type).

```
static int[] intarray = [20, 40, 10, 23];
foreach(int index, int i; intarray)
{
  printf("%d: %d\n", index, i);
}
// equivelant to

for(int index = 0; index < intarray.length; index++)
{
  printf("%d: %d\n", index, intarray[index]);
}
```

In this example index is the index number, the value you'd place in brackets. I is the contained value at that index. To make it a reference which you can store back information to with modifying the array, use the 'inout' keyword.

The same thing works with string hashes.

```
int[char[]] map;
map["test"] = 10;
map["hello"] = 23;
foreach(char[] index, int value; map)
{
   printf("%.*s: %d\n", index, value);
}
```

Try statement

See: Error Handling

```
try
{
  /* statements that could throw an exception */

}
```

If there occurrs an exception between the brackets of **try**, the program will not halt, but keep on running after the whole statement.
The **catch** statement is used to describe, what should happen if there was an exception:

```
try
{
  /* exception? */
}
catch
{
  /* only executed if there occurred an exception */
}
```

You should avoid using catch this way, when your program does not give a good routine for any possible exception.
Sometimes it is very useful to do different things when different exceptions have been thrown.

```
try
{
  /* ? */
}
catch(exc_type)
{
  /* only executed if exc_type was thrown */
}
```

This way, you can decide what to do if there was exactly the exception of type **exc_type** thrown.
Note: It is illegal to create a catch in a catch!

Example:

```
int i = 0;
int v = 0;
try {
  i     = 5/v;
}
catch(StringException)
{
  /* valid */
}
catch
{
  try {
    i = 5/v;
  }
  catch
  {
    /* valid */
  }
}

try {
  i     = 5/v;
}
catch(StringException)
{
  catch(FileException) {
```

```
    /* invalid */
  }
}

try {
  i     = 5/v;
}
catch
{
  /* valid */
}
catch(StringException)
{
  /* invalid */
}
```

In addition you can place a **finally** statement after try or catch to do things that must be done :-) For example to close the file if there was a read error.

```
try
{
  /* exception? */
}
finally
{
  /* this will be executed anyways */
}
```

Example:

```
int i = 0;
try {
  int v = 0;
  i     = 5/v;
} catch {
  i = -1;
  writefln("Exception catched!");
} finally
  writefln("Result is %d",i);
```

Output:

```
Exception catched!
Result is -1
```

See also:
Throw statement

Throw statement

```
throw Expression
```

With this statement you can throw an exception. This exception can be caught with the **try** and **catch** statements and can change the behaviour of the program sensible at runtime.

Note: Exceptions are much slower than return codes, but usually they are more applicable in most situations.

You can use the implemented exceptions, but sooner or later it is handy to declare your own exception:

```
class myPersonalCriticalError : Exception {
  this(char[] msg) {
    super(msg);
```

```
  }
}
```

This way you define a new class called **myPersonalCriticalError** which can be used to throw an exception:

```
try {
  throw new myPersonalCriticalError("This error MUST be!");
}
```

The given string can be retrieved in the catch statement by using **toString**.

Example:

```
class myException : Exception
{
  this(char[] msg)
  {
    super(msg);
  }
}

try {
  throw new myException("Hello World!");
} catch (myException ex) {
  writefln("My exception was thrown:");
  writefln(ex.toString());
}
```

Output:

```
My exception was thrown:
Hello World!
```

See also:
Try statement, Classes, Error Handling

Types

void

Void does not contain any data.
It is for example useful for pointers, as in *std.md5.sum*:

```
void sum(ubyte[16] digest, void[] data)
```

The function does not return anything, but can take arbitrary *data* arrays.

Void comes in handy when you use pointers to data of an undefined type.

bit

A single bit represents zero or one (true or false).
The default initialised value is *false*.

Note: bool is just an alias for bit.

Integer data types

All integer types represent a signed or unsigned number with 2^x bit (with 2 < x < 8).
The default initialised value is always zero.

Example:

```
byte b;       // signed 8 bits [-128 .. +127] is zero.

short s = 5;  // signed 16 bits [-32768.. +32767] is five.

int i;        // signed 32 [-4294967296 .. +4294967295]

long l;       // signed 64 [-1.8e19 .. +1.8e19]

cent c;       // signed 128 [-1,7e+38 .. +1,7e+38] is also zero.


ubyte b = -1  // unsigned 8 bits [0 .. +255] is 255!

ushort;
uint;
ulong;        // unsigned 64 [0 .. 18446744073709551616]

ucent;        // unsigned 128 [0 .. 3,4e+38]
```

Floatingpoint data types

All floatingpoint data types can be used for signed numbers.
The default initialisied value is NaN, NaN * 1.0i (for imaginary) or NaN+NaN * 1.0i (for complex)

Note: Real implementation on Intel CPUs is 80 bits.

```
float f;    // 32 bit floating point

double d;   // 64 bit floating point

real r;     // largest hardware implemented floating point


ifloat if;  //imaginary float

idouble id; // imaginary double
```

```
ireal ir;   // imaginary real

cfloat cf;  // complex number of two float values
cdouble cd; // complex double
creal cr;   // complex real
```

Character data types

```
char ch;   // unsigned 8 bit UTF-8
wchar wch; // unsigned 16 bit UTF-16
dchar dch; // unsigned 32 bit UTF-32
```

If you want to use strings, create an array of char.

Derived data types

Derieved data types are:

- pointer
  Points to any data type in memory.
- array
  Arrays are lists of data types with a specified length.
- function
  These are simply pointers to a function with a defined signature.

Arrays

```
int[5] b;
```

B is now an array of five ints.

```
int[] a;
```

This statement creates an array **a** that can contain integers.
The main difference is, that this array is not a static compiled in array and so can change it's length:

```
int[] nonstatic;
int[3] static1;
int[5] static2;

nonstatic = static1;  // this works (as a reference)


static1   = static2;  // this is NOT valid.
```

In D, every array is a reference to a data structure. After the above code, a change in nonstatic will change static, too!
There is also the possibility to create a pointer to an int structure. This pointer is no array, but it is the real reference 'as variable', can do more than an array but can't use the functions of an array so easily:

```
int* p;
int[5] a;
int[] b;

p = a;                // this works as a reference

p = null;             // this 'kills' the reference


b = a;                // possible.

//b = p;              // impossible!


b[0..2] = 5;          // this is valid, too :-)
```

More array features:

Array Properties

Every array has the following properties:

```
array.length  // Number of elements in array

array.reverse // Reverse element order

array.sort    // Sort element order


array.ptr     // Return the pointer to the first element of array

array.dup     // Duplicate the array in new allocated memory


array.sizeof  // Gives specific information about it's size.
```

Example:

```
char[] textarray   = "  hallo Welt! !";
int l              = textarray.length;  // l == 11

char*  textpointer = textarray.dup;     // the new pointer now points to a duplicated
string,

textpointer[2]     = 'H';               // change the string.

textarray[0..l-2]  = textpointer[2..l]; // move the string.

writefln(textarray);
```

Array Concatenation

You can concatenate arrays (strings are arrays of course) with the cat operator **~**.

Example 1:

```
char[] a = "He";
char[] b = "o world!";
char[] d = "l";
char[] c = a ~ d ~ "l" ~ b;     // c == "Hello world!"
```

Example 2:

```
static int[] a  = [1,2,3];
static int[] b  = [2,3];
int[] c  = a ~ b; // c is an array [1,2,3,2,3]
```

See also:
Array Setting and Operations

Array Slicing

Slicing an array means to specify a subarray of it, by creating a new reference inside it:

```
int[5] foo;   // declare array of 5 ints

int[]  bar;

bar = foo[1..3];  // foo[1..3] is a 2 element array consisting of

                  // foo[1] and foo[2]
```

Slicing is not for copying the contents of an array!

See also:
Array Copying, Array Settings and Operations

Array Copying

To copy the content of one array to another, you could use a for statement. But D copies arrays better and often faster, if you specify on both target sides which elements you are interested in:

```
int[20] foo;
int[3] bar;

bar[0..2]  = foo[5..7];
```

This copies foo[5], foo[6] and foo[7] to bar[0], bar[1] and bar[2].

Note: Do not use more than your variable can handle!
Note: You can not copy overlapping parts:

```
int[2] foo;
int[5] bar;

foo[0..3] = bar[1..4];    // invalid!!


bar[0..3] = bar[1..4];    // also invalid!
```

And you can copy all elements at once too:

```
int[3] bar;
int[3] foo;

bar[]     = foo[];        // copies all elements :-)
```

See also:
Array Slicing, Array Settings and Operations

Array operations

You can use the slice operator "[" to assign one value to multiple values inside an array:

```
int[3] foo;
bar[]     = 2;

foo[]     = 5;
bar[0..2] = 2;
```

This is the same as writing:

```
foo[0]    = 5;
foo[1]    = 5;
foo[2]    = 5;
```

But D goes one step further. The following two examples do the same:

```
for (int i=0; i<2; i++)
{
  bar[i] = foo[i] +5;
}
```

```
bar[] = foo[] +5;
```

See also:
Array Copying, Array Slicing

User Defined Types

- alias
- typedef
- enum
- struct
- union
- class

Alias
An alias is simply an extra name for a type. It does not matter what kind of a type it is, buildin or user.

An alias may also be an alias for a function or namespace.

```
alias int thisisint;
thisisint x = 5;
int function(int u) {}
int function(thisisint) {} // error; conflicts with int version of function

alias function myfunction;
myfunction(x); // this will call 'function'
```

This will just create an int with value 5. 'thisisint' will be fully treated as an int; when doing overloading one must take care of this. If you want it to behave as though it's it's own type instead of a copy reference, you must use typedef.

Typedef
A typedef is almost the same as an alias, though it may not be used to regroup/label functions or namespaces. Typedefs will act as though it's it's own type, so if you do

```
typedef int myint;
void function(int x) { }
void function(myint x) { }
myint i = 4;
int z = 5;
function(i); // the 'myint' version will be called

function(z); // the int version will be called
```

Enum
An enum is an emuration of constants. An enum may be derived from a basic type to assign to it a size in bytes (equal to the type derived). Enums are useful when creating certain properties.

```
enum Day : int { // where int is derived type, may be left out

  Monday,
  Thuesday,
  Wednesday
}
printf("%d", Day.Monday); // will print 0

printf("%d", Day.Thuesday); // will print 1]

Day myDay = Day.Monday;
```

Usually everytime you add an element, the element value is increased by one. You may also be explicit about the constant value assigned to an element:

```
enum Day {
  Monday = 10,
  Thuesday // will be 11

}
```

Classes

A class foo is described with the class statement:

```
class foo
{
  /* your code here */
}
```

A class can contain variables and zero or more constructors , which are called when an object of this class is generated:

```
class foo
{
  int a = 0;
  this()
  {
    a++;
  }
  this(int b)
  {
    a = b;
  }
}

foo myFoo  = new foo();  // myFoo.a  is 1

foo myFoo2 = new foo(5); // myFoo2.a is 5
```

Every class can have none ore one *SuperClass*, from which it is inherited from. It's constructor ("this()") can use **super** to call the SuperClasses constructor:

```
class bar:foo {
  bool isnumber;
  this() {
    isnumber = false;
    super(-1);
  }
  this(int b) {
    super(b+1);
    isnumber = true;
  }
}

bar myBar  = new bar();  // myBar.a  is -1 , myBar.isnumber == false

bar myBar2 = new bar(8); // myBar2.a is  9 , myBar.isnumber == true
```

*In this case, SuperClass is **foo**, the class name is **bar** and the constructor of foo is called with -1 or with b+1.*

Delegates

Delegates can be seen as "method pointers". They contain a reference to the object and a pointer to the method, therefore it can not point to functions or static methods.

Example:

```
int delegate(int) foobar;   // foobar is a delegate which takes an int and returns an
int
```

```
class Foo {
  int mymethod(int x) {
    return x;
  }
}

Foo foo = new Foo();
foobar = &foo.mymethod;
writefln(foobar(128));      // call mymethod(128)
```

Error Handling

Error Handling in D is done via Exceptions. Exceptions are special objects that hold information about the error that occured (in most cases this will be an error message like "File not found").

First, you would declare your own Exception like this:

```
class FileNotFoundException : Exception
{
  this(char[] msg)
  {
    super(msg);
  }
}
```

Declaring your own Exception type isn't neccessary, though. You can use the Exception class directly, but be aware that the code that wants to "catch" you exception will not be able to distinguish it from other Exception types.

If an error occurs, the program "throws" an exception:

```
throw new FileNotFoundException("I cant find the file " ~ filename);
```

The execution of code will stop at this point and if there is no "catch"-Statement for your exception, the program will exit.
Now if you don't want your program to crash on a certain exception (perhaps because you expected that a function could fail and/or you want to handle the error yourself) you can use the "try/catch" statement. Let's assume the execption could get thrown during a function called *ReadMyFile*. If you want to catch it, you would write:

```
try {
  ReadMyFile(filename);
} catch (FileNotFoundException ex) {
  /* We caught the exception, now we can handle it */
  writefln("I couldn't read the file, but it doesn't really matter.");
  writefln("I will just go on with the normal program.");
}
```

There is but one problem with exceptions: Let's assume you write a function *ReadFiles* that allocates some memory without garbage collection (or it uses any type of resource that has to be freed again) and then calls the *ReadMyFile* function. Your function doesn't really bother if the file is present or not, but if an exception is thrown, you would have to free your resources!
You can easily accomplish that with the finally statement:

```
function ReadFiles()
{
  /* Allocate critical resources ... */
  try {
```

```
    ReadMyFile("foo.txt");
    ReadMyFile("bar.txt");
  }
  finally
  {
    /* Deallocate critical resources... */
  }
}
```

The statements in the finally block only get executed when any exeception is thrown during the try block.

Memory Management

To do: Describe Garbage collection and how it works in D. Link to std.gc where applicable.

The standard runtime library: Phobos

The standard runtime library of the D language is called Phobos.

It aims to be platform-independent and doesn't just contain wrappers for the C standard library. It makes use of D's features, it uses contract programming for improving code quality and exceptions for error handling.

std.base64

Encode/decode base64 format.

std.base64.encode

```
char[] encode(char[] str)
```

This will encode a char array to base64.
To do: What is base64 encoding, where do you need it?

Example:

```
writef(encode("all your base64 are belong to foo"));
```

Prints:

```
YWxsIHlvdXIgYmFzZTY0IGFyZSBiZWxvbmcgdG8gZm9v
```

std.base64.decode

```
char[] decode(char[] estr)
```

This will decode a base64 encoded char array, the opposite of base64.encode.

Example:

```
writef(decode(encode("Anything you want to...\n")));
```

Prints:

```
Anything you want to...
```

std.boxer

This module is a set of types and functions for converting any object (value or heap) into a generic box type, allowing the user to pass that object around without knowing what's in the box, and then allowing him to recover the value afterwards.

Example:

```
// Convert the integer 45 into a box.

Box b = box(45);

// Recover the integer and cast it to real.

real r = unbox!(real)(b);
```

That is the basic interface and will usually be all that you need to understand. If it cannot unbox the object to the given type, it throws UnboxException. As demonstrated, it uses implicit casts to behave in the exact same way that static types behave. So for example, you can unbox from int to real, but you cannot unbox from real to int: that would require an explicit cast.

This therefore means that attempting to unbox an int as a string will throw an error instead of formatting it. In general, you can call the toString method on the box and receive a good result, depending upon whether std.string.format accepts it.

Boxes can be compared to one another and they can be used as keys for associative arrays.

There are also functions for converting to and from arrays of boxes.

Example:

```
// Convert arguments into an array of boxes.

Box[] a = boxArray(1, 45.4, "foobar");

// Convert an array of boxes back into arguments.

TypeInfo[] arg_types;
void* arg_data;

boxArrayToArguments(a, arg_types, arg_data);

// Convert the arguments back into boxes using a

// different form of the function.

a = boxArray(arg_types, arg_data);
```

One use of this is to support a variadic function more easily and robustly; simply call "boxArray(_arguments, _argptr)", then do whatever you need to do with the array.

std.boxer.Box

struct Box
Box is a generic container for objects (both value and heap), allowing the user to box them in a generic form and recover them later.

TypeInfo type
Property for the type contained by the box. This is initially null and cannot be assigned directly.

void* data
Property for the data pointer to the value of the box. This is initially null and cannot be assigned directly.

bit unboxable(TypeInfo type)
Return whether the value could be unboxed as the given type without throwing an error.

char[toString()
Attempt to convert the boxed value into a string using std.string.format; this will throw if that function cannot handle it. If the box is uninitialized then this returns "".

bit opEquals(Box other)
Compare this box's value with another box. This implicitly casts if the types are different, identical to the regular type system.

float opCmp(Box other)
Compare this box's value with another box. This implicitly casts if the types are different, identical to the regular type system.

uint toHash()
Return the value's hash.

Box box(...)
Box the single argument passed to the function. If more or fewer than one argument is passed, this will assert.

Box box(TypeInfo type, void* data)
Box the explicitly-defined object. type must not be null; data must not be null if the type's size is greater than zero.

Box[boxArray(TypeInfo[types, void* data)
Box[boxArray(...)
Convert a list of arguments into a list of boxes.

void boxArrayToArguments(Box[arguments, out TypeInfo[types, out void* data)
Convert a list of boxes into a list of arguments.

bit unboxable!(T)(Box value)
Return whether the value can be unboxed as the given type; if this returns false, attempting to do so will throw UnboxException.

T unbox!(T)(Box value)
This template function converts a boxed value into the provided type. To use it, instantiate the template with the desired result type, and then call the function with the box to convert.

This will implicitly cast base types as necessary and in a way consistent with static types - for

example, it will cast a boxed byte into int, but it won't cast a boxed float into short. If it cannot cast, it throws UnboxException.

T unbox!(T)(Box value)

Prototype:

```
T unbox!(T)(Box value)
```

This template function converts a boxed value into the provided type. To use it, instantiate the template with the desired result type, and then call the function with the box to convert.

This will implicitly cast base types as necessary and in a way consistent with static types - for example, it will cast a boxed byte into int, but it won't cast a boxed float into short. If it cannot cast, it throws UnboxException.

Example:

```
Box b = box(4.5);
bit u = unboxable!(real)(b); // This is true.

real r = unbox!(real)(b);
```

std.boxer.UnboxException

Prototype:

```
class UnboxException : Error
```

This class is thrown if unbox is unable to cast the value into the desired result.

this(Box object, TypeInfo outputType)
Assign parameters and create the message in the form "Could not unbox from type to ."

box object
This is the box that the user attempted to unbox.

TypeInfo outputType
This is the type that the user attempted to unbox the value as.

std.compiler

This module holds the information about the D compiler that you are using.

std.compiler.name

```
char [] name
```

This string contains the vendor specific name of the compiler.

Example:

```
writefln(std.compiler.name);
```

Prints (if you are using DMD):

```
Digital Mars D
```

std.compiler.Vendor

```
enum Vendor
{
  DigitalMars = 1
}
```

The master list of D compiler vendors. Currently, Digital Mars is the only vendor for a D compiler, so there is only one entry.
This is just a type definition, if you want to get the vendor of your compiler use std.compiler.vendor.

std.compiler.vendor

```
Vendor vendor = /* ... */;
```

Defines the vendor that produced this compiler. The std.compiler.Vendor type is an enum.

Example:

```
if (std.compiler.vendor = std.compiler.Vendor.DigitalMars)
  writefln("Hey, i like your compiler!");
```

std.compiler.version_major

```
uint version_major;
```

Holds the major version number.

See also: std.compiler.version_minor

std.compiler.version_minor

```
uint version_minor;
```

Holds the minor version number.

See also: std.compiler.version_major

Example:

```
writefln("Your compiler has the version number %d.%d",
  std.compiler.version_major,
  std.compiler.version_minor);
```

std.compiler.D_major

```
uint D_major;
```

The major version of the D language specification supported by the compiler.

See also: std.compiler.D_minor

std.compiler.D_minor

```
uint D_minor;
```

The minor version of the D language specification supported by the compiler.

See also: std.compiler.D_major

Example:

```
writefln("Your compiler supports D version %d.%d", std.compiler.D_major,
std.compiler.D_minor);
```

std.conv

```
byte toByte(char[] s)
ubyte toUbyte(char[] s)
short toShort(char[] s)
ushort toUshort(char[] s)
int toInt(char[] s)
uint toUint(char[] s)
long toLong(char[] s)
ulong toUlong(char[] s)
```

Functions **to*type*** convert a string **s** into numeric type *type*.
They differ from the C functions atoi() and atol() by not allowing whitespace or overflows.

See also:
std.string.atoi, std.string.atof

std.ctype

This module provides functionality for classification of simple characters.

std.ctype.isalnum

```
int isalnum(dchar c)
```

Returns a value != 0 if c is an alphanumeric character (a-z, A-Z, 0-9).

std.ctype.isalpha

```
int isalpha(dchar c)
```

Returns a value != 0 if c is an alphabetic character (a-z, A-Z).

std.ctype.iscntrl

```
int iscntrl(dchar c)
```

Returns a value != 0 if c is a control character (Usually characters 0-31).

std.ctype.isdigit

```
int isdigit(dchar c)
```

Returns a value != 0 if c is an digit (0-9).

std.ctype.islower

```
int islower(dchar c)
```

Returns a value != 0 if c is an lower case character (a-z).

std.ctype.isupper

```
int isupper(dchar c)
```

Returns a value != 0 if c is an upper case character (A-Z).

std.ctype.isxdigit

```
int isxdigit(dchar c)
```

Returns a value != 0 if c is an hexadecimal digit (a-f, A-F, 0-9).

std.ctype.ispunct

```
int ispunct(dchar c)
```

Returns a value != 0 if c is a punctuation character.

std.ctype.isspace

```
int isspace(dchar c)
```

Returns a value != 0 if c is a space character (Space, tab, line break etc.).

std.ctype.isgraph

```
int isgraph(dchar c)
```

Returns a value != 0 if c is a printing character, except space (Alphanumerical or punctuation).

std.ctype.isprint

```
int isprint(dchar c)
```

Returns a value != 0 if c is a printing character, including space (like std.ctype.isgraph plus space).

std.ctype.isascii

```
int isascii(dchar c)
```

Returns a value != 0 if c is an character from the ASCII character set (Characters 0-127).

std.ctype.tolower

```
dchar tolower(dchar c)
```

Returns the lower case character for c, if c is upper case, unmodified c else.

std.ctype.toupper

```
dchar toupper(dchar c)
```

Returns the upper case character for c, if c is lower case, unmodified c else.

std.date

Dates are represented in several formats. The date implementation revolves around a central type, d_time, from which other formats are converted to and from.

std.date.d_time

```
alias long d_time
```

d_time represents the time elapsed since January 1, 1970. The time unit of d_time is *Ticks*, which - a bit unclear - milliseconds or even smaller intervals. They are more precisely defined by the TicksPerSecond value.

Note: Negative values are for dates preceding 1970.

See also:
TicksPerSecond

std.date.TicksPerSecond

```
int TicksPerSecond
```

A constant giving the number of Ticks per second for this implementation. At least 1000.

See also:
d_time

std.date.toString

```
char[] toString(d_time t)
```

toString converts **t** into a text string of the form: "Www Mmm dd hh:mm:ss GMT+-TZ yyyy", for example, "Tue Dec 08 02:04:57 GMT-0800 1997".

Notes:
* If t is invalid, "Invalid date" is returned.
* t is assumed to be a UTC date and not a local date.
* The function returns the local date as a string. It does the UTC to Local conversion automatically.

See also:
toUTCString, toDateString, toTimeString, parse

std.date.toUTCString

```
char[] toUTCString(d_time t)
```

toUTCString converts **t** into a text string of the form: "Www, dd Mmm yyyy hh:mm:ss UTC".

Note: If t is invalid, "Invalid date" is returned.

See also:
toString

std.date.toDateString

```
char[] toDateString(d_time t)
```

Converts the **t** into a text string of the form: "Www Mmm dd yyyy", for example, "Tue Dec 08 1997".

Note: If t is invalid, "Invalid date" is returned.

See also:
toString, toTimeString

std.date.toTimeString

```
char[] toTimeString(d_time t)
```

This function converts **t** into a text string of the form: "hh:mm:ss GMT+-TZ", for example, "02:04:57 GMT-0800".

Note: If t is invalid, "Invalid date" is returned.

See also:
toString, toDateString

std.date.parse

```
d_time parse(char[] s)
```

Parses **s** as a textual date and returns it as a *d_time*

Note: If the string is not a valid date, d_time.init is returned.

See also:
toString

std.date.toISO8601YearWeek

```
void toISO8601YearWeek(long t, out int year, out int week);
```

It calculates year and week (1 - 53) from t. The ISO 8601 week 1 is the first week of the year that includes January 4. The first day of the week is monday.

Example

```
import std.date;
import std.stdio;

int main() {
        int year, week;
        toISO8601YearWeek( 1333, year, week );
        writef( year, '\n', week );
        return 0;
}
```

std.date.UTCtoLocalTime

```
long UTCtoLocalTime(long t);
```

Convert from UTC to local time.

Example

```
import std.date;
import std.stdio;

int main() {
        long time;
        time = UTCtoLocalTime( 1 );
        writef( time );
        return 0;
}
```

std.date.LocalTimetoUTC

```
long LocalTimetoUTC(long t);
```

Convert from local time to UTC.

Example

```
import std.date;
import std.stdio;

int main() {
        long time;
        time = LocalTimetoUTC( 1 );
        writef( time );
        return 0;
}
```

std.date.getUTCtime

```
long getUTCtime();
```

Get current UTC time.

Example

```
import std.date;
import std.stdio;

int main() {
        long time;
        time = getUTCtime();
        writef( time );
        return 0;
}
```

std.date.DosFileTime

```
typedef DosFileTime;
```

Type representing the DOS file date/time format.

std.date.toDtime

```
long toDtime( DosFileTime time );
```

Convert from DOS file date/time to d_time.

std.date.toDosFileTime

```
DosFileTime toDosFileTime( long t );
```

Convert from d_time to DOS file date/time.

std.file

Provides basic file functions, like *read*, *write* or *append*.

Dsource Examples

- Maze Solver

std.file.FileException

```
class FileException
```

This exception is thrown if there occurs a file I/O error.

See also:
Try statement

std.file.exists

```
int exists(char[] name)
```

This function checks, if **name** does exist. It can be used for files or directories.

Returns zero, if file was not found, otherwise not zero.

Example:

```
if (exists(args[0])!=0) {
  writefln("I do exist!");
} else {
  writefln("I do not exist :-(");
}
```

See also:
isfile, isdir, read, getSize

std.file.isfile

```
int isfile(char[] name)
```

This function checks, if **name** is a file.
It throws an error if **name** does not exist in any form (file, directory, link, ...).

Returns zero, if it was not a file, otherwise not zero.

See also:
FileException, exists, isdir, read

std.file.isdir

```
int isdir(char[] name)
```

This function checks, if **name** is a directory.
It throws an error if **name** does not exist in any form (file, directory, link, ...).

Returns zero, if it was not a directory, otherwise not zero.

See also:
FileException, exists, isfile, chdir, listdir

std.file.read

```
void[] read(char[] name)
```

This function reads a file **name** and returns an array of bytes.
It can throw a **FileException**, if the file does not exist (anymore) or can not be read.

Example:

```
// Import standard-in-out

import std.stdio;
import std.file;

// Main function

int main(char[][] args)
{
  char[] content;
  // This program reads a file and prints it to the screen

  try
  {
    content = cast(char[])read(args[1]);
    writefln(content);
  }
  catch (FileException xy)
  {
    writefln();
    writefln("A file exception occured: " ~ xy.toString());
  }
  catch (Exception xx)
  {
    writefln();
    writefln("An exception occured: " ~ xx.toString());
  }
  return 0;
}
```

See also:
FileException, exists, isfile, write, append, remove, rename

std.file.write

```
void write(char[] name, void[] buffer)
```

This function writes **buffer** to **name**.
It can throw a **FileException**

Example:

```
// Import standard-in-out

import std.stdio;
import std.file;

// Main function

int main(char[][] args)
{
  try
  {
    write("test.txt","Hallo Welt!");
    writefln("Successfully wrote file 'test.txt'");
```

```
  }
  catch (FileException xy)
  {
    writefln("A file exception occured: " ~ xy.toString());
  }
  catch (Exception xx)
  {
    writefln("An exception occured: " ~ xx.toString());
  }
  return 0;
}
```

See also:
FileException, exists, isfile, read, append

std.file.append

```
void append(char[] name, void[] buffer)
```

This function appends **buffer** to the file **name**.
It can throw a **FileException**.

Example:

```
// Import standard-in-out

import std.stdio;
import std.file;

// Main function

int main(char[][] args)
{
  try
  {
    append("test.txt","\nHallo Welt!!\n");
    writefln("Successfully appended to file 'test.txt'");
  }
  catch (FileException xy)
  {
    writefln("A file exception occured: " ~ xy.toString());
  }
  catch (Exception xx)
  {
    writefln("An exception occured: " ~ xx.toString());
  }
  return 0;
}
```

See also:
FileException, exists, isfile, read, write

std.file.remove

```
void remove(char[] name)
```

This function deletes the file **name**.
It can throw a **FileException**.

See also:
exists, write, rename

std.file.rename

```
void rename(char[] old, char[] new)
```

Renames the file **old** to **new**.
This function can throw a **FileException**.

See also:
remove, FileException

std.file.getSize

```
uint getSize(char[] name)
```

Retrieves the size of file **name** in bytes.
Can throw **FileException**.

See also:
exists, getAttributes

std.file.getAttributes

```
uint getAttributes(char[] name)
```

Retrieves the attributes of file **name**.
Can throw **FileException**.

See also:
exists, getSize

std.file.getcwd

```
char[] getcwd()
```

Get current directory.

See also:
chdir, mkdir, listdir

std.file.chdir

```
void chdir(char[] name)
```

Change directory.

See also:
getcwd, mkdir, rkdir, listdir

std.file.mkdir

```
void mkdir(char[] name)
```

Creates a new directory called **name**.

See also:
chdir, rmdir

std.file.rmdir

```
void rmdir(char[] name)
```

This function removes the directory **name**.

See also:
chdir, mkdir, isdir

std.file.listdir

```
char[][] listdir(char[] pathname)
```

Listdir returns an array of strings that represents the directory content.

See also:
isfile, isdir, read, chdir

std.format

This module implements the workhorse functionality for string and I/O formatting. It's comparable to C99's vsprintf().

std.format.FormatError

```
class FormatError : Error
```

This error gets thrown when there is a mismatch between a format and its corresponding argument

std.format.doFormat

```
void doFormat(void delegate(dchar) putc, TypeInfo[] arguments, va_list argptr)
```

Interprets variadic argument list pointed to by argptr whose types are given by arguments[, formats them according to embedded format strings in the variadic argument list, and sends the resulting characters to putc. Mismatched arguments and formats result in a FormatError being thrown.

The variadic arguments are consumed in order. Each is formatted into a sequence of chars, using the default format specification for its type, and the characters are sequentially passed to putc. If a char[, wchar[, or dchar[argument is encountered, it is interpreted as a format string. As many arguments as specified in the format string are consumed and formatted according to the format specifications in that string and passed to putc. If there are too few remaining arguments, a FormatError is thrown. If there are more remaining arguments than needed by the format specification, the default processing of arguments resumes until they are all consumed.

Any strings used as formats or that are output, and any characters, must be valid UTF characters.

Format string

Format strings consist of characters interspersed with *format specifications*. Characters are simply copied to the output (such as putc) after any necessary conversion to the corresponding UTF-8 sequence.

A *format specification* always starts with a percent '%' character. For writing a literal '%' you just immediately follow it by another '%'.

Example: "100%%" will result in "100%".

Each format specification consumes an argument following the format string. It describes how the argument should be printed. It consists of the follwing elements:

1. The literal '%' character which signals the start of a format specification.

2. **Flags** (optional):
   '-' will left-justify the resulting string in the field.
   '+' will prefix any positive number with a plus sign.
   '#' means "Use alternative formatting".
   - For octal format ('o') this will require that the string starts with a zero digit.
   - For hexadecimal format ('x','X') this will prefix the string with '0x' or '0X', if the value is non-zero.
   - For floating point format ('e','E','f','F') this will always insert the decimal point
   - For the automatic floating point format ('g','G') this will prohibit the elusion of trailing zeros.
3. The **minimum field width** (optional) as numerical value. If it is '*', the next argument (which must be an int) is taken as the minimum field width, a negative value here will do the same as the '-' flag (left-justify).

4. The **precision** (optional) which must start with a dot, followed by either a numerical value or '*'. '*' will do almost the same as in the minimum field with: the next argument (which must be an int) is taken as the precision, a negative value here will set the precision to zero.

5. The **formatting character** (required). This character does the main conversion. Possible are:

    - 's': String
      The corresponding argument is formatted in a manner consistent with its type.

    - 'b','d','o','x','X': Binary, decimal, octal, hex (lower case), hex (upper case)
      The corresponding argument must be an integral type and is formatted as an integer. Only 'd' respects signed arguments.

    - 'e','E': Exponential floating point format
      Formats a floating point number with one digit before the decimal point plus *precision* digits after, followed by either 'e' or 'E' (depending on the formatting character used) and the exponent with at least 2 digits. The default precision is 6.

    - 'f','F': Floating point format (decimal)
      At least one digit before the decimal point plus *precision* digits after. Default precision is 6.

    - 'g','G': Automatic floating point format
      This will use the decimal floating point format only if the exponent is greater than -5 or less than *precision*. The *precision* specifies the number of significant digits, and defaults to one.

    - 'a','A': Hexadecimal floating point format
      Formats a floating point number starting with '0x' (or '0X' respectively) with one hexadecimal digit before the decimal point plus *precision* hexadecimal digits after, followed by either 'p' or 'P' (depending on the formatting character used) and the decimal (!) exponent to the base of 2. If no *precision* is given, as many hexadecimal digits as necessary to exactly represent the mantissa are generated.

std.gc

This module provides functionality for controlling the garbage collector.

The garbage collector normally works behind the scenes without needing any specific interaction. These functions are for advanced applications that benefit from tuning the operation of the collector.

std.gc.addRoot

```
void addRoot(void *p);
```

Add p to list of roots. Roots are references to memory allocated by the collector that are maintained in memory outside the collector pool. The garbage collector will by default look for roots in the stacks of each thread, the registers, and the default static data segment. If roots are held elsewhere, use *addRoot* or std.gc.addRange to tell the collector not to free the memory it points to.

std.gc.removeRoot

```
void removeRoot(void *p);
```

Remove p from list of roots.

See also: std.gc.addRoot

std.gc.addRange

```
void addRange(void *pbot, void *ptop);
```

Add range to scan for roots.

std.gc.removeRange

```
void removeRange(void *pbot);
```

Remove range.

See also: std.gc.addRange

std.gc.fullCollect

```
void fullCollect();
```

Run a full garbage collection cycle. The collector normally runs synchronously with a storage allocation request (i.e. it never happens when in code that does not allocate memory). In some circumstances, for example when a particular task is finished, it is convenient to explicitly run the collector and free up all memory used by that task. It can also be helpful to run a collection

before starting a new task that would be annoying if it ran a collection in the middle of that task. Explicitly running a collection can also be done in a separate very low priority thread, so that if the program is idly waiting for input, memory can be cleaned up.

std.gc.genCollect

```
void genCollect();
```

Run a generational garbage collection cycle. Takes less time than a std.gc.fullCollect, but isn't as effective.

std.gc.minimize

```
void minimize();
```

Minimize physical memory usage.

std.gc.disable

```
void disable();
```

Temporarilly disable garbage collection cycle. This is used for brief time critical sections of code, so the amount of time it will take is predictable. If the collector runs out of memory while it is disabled, it will throw an OutOfMemory exception. The *disable* function calls can be nested, but must be matched with corresponding std.gc.enable calls.

std.gc.enable

```
void enable();
```

Reenable garbage collection cycle after being disabled with std.gc.disable. It is an error to call more *enable*s than *disable*s.

std.intrinsic

std.math

The core library module Math includes the usual math functions like sin, cos or atan.

std.math doc is not complete, yet!

std.math - Constants

Math provides the following real type constants:

```
const real PI; // ~=3.14159265

const real LOG2; // Logarithmus of two. ~=0.301029995

const real LN2; // ~= 0.693147180

const real LN10; // ~= 2.3025
const real E; // ~= 2.718

const real SQRT2; // the same as sqrt(2);

const real SQRT1_2; // the same as sqrt(0.5);

const real PI_2; // PI*0.5
const real PI_4; // PI*0.25
const real M_1_PI ;
const real M_2_PI;
const real LOG2T;
const real LOG2E ;
const real LOG10E;
const real M_2_SQRTPI ;
```

std.math.sin

```
real sin(real x)
```

Returns the sinus of x (radians).

If x is INFINITY, the function returns -NAN.

Example:

```
//don't forget to import std.stdio and std.math :-)


writefln("sin(0)    is %f",sin(0));
writefln("sin(PI/2) is %f",sin(PI/2));
writefln("sin(PI/4) is %f",sin(PI/4));
writefln("sin(INF)  is %f",sin(float.infinity));
writefln("sin(-INF) is %f",sin(-float.infinity));
```

Output:

```
sin(0)    is 0.000000
```

```
sin(PI/2) is 1.000000
sin(PI/4) is 0.707107
sin(INF)  is -nan
sin(-INF) is -nan
```

See also:
std.math - Constants (PI)

std.math.cos

```
real cos(real x)
```

Returns the cosinus of x (radians).

If x is INFINITY, the function returns -NAN.

Example:

```
//don't forget to import std.stdio and std.math :-)


writefln("cos(0)    is %f",cos(0));
writefln("cos(PI/2) is %f",cos(PI/2));
writefln("cos(PI/4) is %f",cos(PI/4));
writefln("cos(INF)  is %f",cos(float.infinity));
writefln("cos(-INF) is %f",cos(-float.infinity));
```

Output:

```
cos(0)    is 1.000000
cos(PI/2) is 0.000000
cos(PI/4) is 0.707107
cos(INF)  is -nan
cos(-INF) is -nan
```

See also:
std.math - Constants (PI)

std.math.tan

```
real tan(real x)
```

Returns the tangent of x (radians).

If x is INFINITY, the function returns NAN.

Example:

```
//don't forget to import std.stdio and std.math :-)


writefln("tan(0)    is %f",tan(0));
writefln("tan(PI/2) is %f",tan(PI/2));
writefln("tan(PI/4) is %f",tan(PI/4));
writefln("tan(INF)  is %f",tan(float.infinity));
writefln("tan(-INF) is %f",tan(-float.infinity));
```

Output:

```
tan(0)    is 0.000000
tan(PI/2) is 16331778728383843.837
tan(PI/4) is 1.000000
```

```
tan(INF)  is nan
tan(-INF) is nan
```

See also:
std.math - Constants (PI)

std.math.sinh

```
real sinh(real x)
```

See also:
std.math.cosh, std.math.sin

std.math.cosh

```
real cosh(real x)
```

Returns the cosh of x.

If x is INFINITY, the function returns INFINITY.

Example:

```
//don't forget to import std.stdio and std.math :-)


writefln("cosh(0)         is %f",cosh(0));
writefln("cosh(1)         is %f",cosh(1));
writefln("cosh(nan)       is %f",cosh(float.nan));
writefln("cosh(-INF)      is %f",cosh(-float.infinity));
```

Output:

```
cosh(0)         is 1.000000
cosh(1)         is 1.543081
cosh(nan)       is nan
cosh(-INF)      is inf
```

See also:
std.math.cos

std.math.pow

```
real pow(real x, real y)
```

Returns x^y.

Example:

```
// We use cast here to tell the compiler what we are talking about.


writefln("pow(3,4)    is %d",cast(int)pow(cast(float)3,cast(float)4));
writefln("pow(4,3)    is %d",cast(int)pow(cast(float)4,cast(float)3));
```

Output:

```
pow(3,4)    is 81
pow(4,3)    is 64
```

See also:
std.math.sqrt, std.math.hypot

std.math.sqrt

```
real sqrt(real x)
creal sqrt(creal x)
```

This function returns the square root of x.

If x<0, the function returns NaN, where INFINITY results to INFINITY.

Example:

```
// The compiler does not know, if you want a real or a creal when typing "9".

// This is why we use cast(float) here.


writefln("sqrt(0)    is %f",sqrt(cast(float)0));
writefln("sqrt(9)    is %f",sqrt(cast(float)9));
writefln("sqrt(INF)  is %f",sqrt(cast(float)float.infinity));
writefln("sqrt(-1)   is %f",sqrt(cast(float)-1));
```

Output:

```
sqrt(0)    is 0.000000
sqrt(9)    is 3.000000
sqrt(INF)  is inf
sqrt(-1)   is -nan
```

See also:
std.math.pow, std.math.hypot

std.math.floor

```
real floor(real x)
```

Rounds x down.

Example:

```
writefln("floor(2.3)    is %d",cast(int)floor(2.3));
writefln("ceil(3.2)    is %d",cast(int)ceil(3.2));
```

Output:

```
floor(2.3)    is 2
ceil(3.2)    is 4
```

See also:
std.math.ceil

std.math.ceil

```
real ceil(real x)
```

Rounds x up.

Example:

```
writefln("ceil(3.2)    is %d",cast(int)ceil(3.2));
```

Output:

```
ceil(3.2)    is 4
```

See also:
std.math.floor

std.math.isnan

```
int isnan(real x)
```

Returns true, if x is not a number (NaN).

See also:
std.math.isinf

std.math.isinf

```
int isinf(real x)
```

Returns true, if x is infinite.

See also:
std.math.isnan, std.math.isfinite

std.math.isfinite

```
int isfinite(real x)
```

Returns true, if x is finite.
Every normal number like -5, 8 or 5.3 is finite.

See also:
std.math.isnan, std.math.isinf

std.math.frexp

```
real frexp(real value, out int exp)
```

Calculates and returns **x** and **exp** so that

```
value      = x*2^exp
with  0.5 <= |x| < 1.0
```

X has same sign as **value**.

See also:
std.math.ldexp

std.math.ldexp

```
real ldexp(real n, int exp)
```

Computes n * 2^exp.

See also:
std.math.frexp

std.math.hypot

```
real hypot(real x, real y)
```

Returns:

```
sqrt(pow(x,2) + pow(y,2))
```

See also:
std.math.pow, std.math.pow

std.math.copysign

!! missing !!

std.math.signbit

!! missing !!

std.md5

Md5 digests are 16 byte checksums.
You simply can compute a string to a md5 coded byte list by calling **sum**.
If your data is buffered, you better use **MD5_CTX**.

See also:
std.md5.sum, std.md5.MD5_CTX

std.md5.sum

```
void sum(ubyte[16] digest, void[] data)
```

Computes **data** to md5 sum **digest**.

Example:

```
import std.md5;

int main(char[][] args) {
  ubyte[16]   md5sum;
  sum(md5sum,"Hello World!");
  printDigest(md5sum);
  return 0;
}
```

Output:

```
ed076287532e86365e841e92bfc50d8c
```

See also:
printDigest, MD5_CTX

std.md5.MD5_CTX

```
struct MD5_CTX
```

Use this, when the data you want to compute is buffered. Otherwise use **sum**.

Example:

```
import std.md5;

int main(char[][] args) {
  ubyte[16]   md5sum;

  MD5_CTX     my_md5_ctx;

  my_md5_ctx.start();

  // first block of data

  my_md5_ctx.update("Hello");

  /* get second block of data */
  /* in this example we do this the stupiest way :-) */
  char[]      second  = " World!";

  // second block of data

  my_md5_ctx.update(second);

  my_md5_ctx.finish(md5sum);

  printDigest(md5sum);
  return 0;
}
```

Output:

```
ed076287532e86365e841e92bfc50d8c
```

See also:
sum, start, update, finish

std.md5.start

```
void start()
```

Begins an MD5 message-digest operation.

See also:
MD5_CTX

std.md5.update

```
void update(void[] input)
```

Continues an MD5 message-digest operation, processing another message block input, and updating the context.

See also:
MD5_CTX

std.md5.finish

```
void finish(ubyte[16] digest)
```

Ends an MD5 message-digest operation and writes the result to **digest**.

See also:
MD5_CTX

std.md5.printDigest

```
void printDigest(ubyte[16] digest)
```

Print MD5 digest to standard output.

See also:
sum, MD5_CTX

std.mmfile

std.outbuffer

std.path

Manipulate file names, path names, etc.

std.path.curdir

```
const char[] curdir;
```

String representing the current directory.

Example:

```
writefln(curdir);
```

See also:
pardir, isabs

std.path.pardir

```
const char[] pardir;
```

String representing the parent directory.

Example:

```
writefln(pardir);
```

See also:
curdir

std.path.getDirName

```
char[] getDirName(char[] name)
```

Get the path of the file **name**.

Example:

```
writefln(getDirName(args[0]));
```

Output could be:

```
c:\d\projects\test
```

See also:
getBaseName, getDrive, getExt, isabs

std.path.getBaseName

```
char[] getBaseName(char[] name)
```

Get the filename of **name**.

Example:

```
writefln(getBaseName(args[0]));
```

Output could be:

```
test.exe
```

See also:
getDirName, getDrive, getExt

std.path.isabs

```
int isabs(char[] path)
```

Determine if absolute path name.

Returns 1 if **path** is an absolute path, otherwise 0.

Example:

```
writefln(isabs(args[0]));
writefln(isabs("test"));
writefln(isabs(getDirName(args[0])));
writefln(isabs(getDirName("..")));
```

Output (on Windows):

```
1
0
1
0
```

See also:
getDirName, curdir

std.path.getDrive

```
char[] getDrive(char[] fullname)
```

Get drive of **name**.

Example:

```
writefln(getDrive(args[0]));
```

Output could be:

```
C:
```

See also:
getDirName, getBaseName, getExt

std.path.getExt

```
char[] getExt(char[] name)
```

Retrieves the extension of file **name**.

Example:

```
writefln(getExt(args[0]));
```

Output (compiled as *.exe):

```
exe
```

See also:
getDirName, getBaseName, getDrive, defaultExt, addExt

std.path.defaultExt

```
char[] defaultExt(char[] fullname, char[] ext)
```

Put a default extension on fullname if it doesn't already have an extension.

See also:
getExt, addExt

std.path.addExt

```
char[] addExt(char[] fullname, char[] ext)
```

Add file extension or replace existing extension.

See also:
getExt, defaultExt

std.path.sep

```
const char[] sep;
```

Represents the character, which is used to separate directory names in a path.

Example:

```
writefln(sep);
writefln(altsep);
```

Output on Windows:

```
\
/
```

See also:
altsep, pathsep, linesep

std.path.altsep

```
const char[] altsep;
```

Represents the alternative character, which is used to separate directory names in a path. Default seperator is **sep**.

See also:

sep

std.path.pathsep

```
const char[] pathsep;
```

Represents the string, which seperates two paths.

Example:

```
writefln(pathsep);
```

Output:

```
;
```

See also:
sep, linesep

std.path.linesep

```
const char[] linesep;
```

Represents the string, which seperates two lines.

See also:
sep, pathsep

std.path.fnmatch

```
int fnmatch(char[] name, char[] pattern)
```

Match filename strings with pattern[, using the following wildcards:

```
* match 0 or more characters
? match any character
[chars] match any character that appears between the []
[!chars] match any character that does not appear between the [! ]
```

Returns 0 if no match was found, otherwise the result is not 0.

Note: Matching is case sensitive on a file system that is case sensitive.

See also:
fncharmatch

std.path.fncharmatch

```
int fncharmatch(dchar c1, dchar c2)
```

Match file name characters.

Returns 0 if no match was found, otherwise the result is not 0.

Note: Matching is case sensitive on a file system that is case sensitive.

See also:
fnmatch

std.path.join (*)

!! missing !!

std.process

std.random

This module is used for random number generation.

std.random.rand

```
uint rand()
```

This function returns a random number.

Note: Use rand_seed to create a repeatable sequence of random numbers.

Example:

```
import std.stdio;
import std.random;
void main()
{
    for(int i = 1; i <= 5; i++)
        writefln("rand() returns %d", rand());
}
```

Output could be:

```
rand() returns 2018360261
rand() returns 4236961988
rand() returns 1858414318
rand() returns 2437703434
rand() returns 1482694331
```

See also:
rand_seed

std.random.rand_seed

```
void rand_seed(uint seed, uint index)
```

Use this function to create a repeatable sequence of random numbers.
Seed is used to start the sequence, while **index** tells the function to begin with the *index*-th number. Incrementing index is the same as calling rand() one more time after rand_seed.

Note: The random number generator is seeded at program startup with a good number. There is no need to do that every time yourself.

Example:

```
import std.stdio;
import std.random;

void main()
{
    const int max = 3;
    writefln("Before rand_seed() called...");
    for(int i = 1; i <= max; i++)
       writefln("rand() is %12d", rand());

    rand_seed(1,0);
    writefln("\nAfter rand_seed(1,0) called...");
    for(int i = 1; i <= max; i++)
       writefln("rand() is %12d", rand());

    rand_seed(1,1);
    writefln("\nAfter rand_seed(1,1) called...");
    for(int i = 1; i <= max; i++)
       writefln("rand() is %12d", rand());
```

```
}
```

Output could be:

```
Before rand_seed() called...
rand() is   3504812646
rand() is    904386554
rand() is    373390471

After rand_seed(1,0) called...
rand() is     48551977
rand() is   1352404003
rand() is    370828309

After rand_seed(1,1) called...
rand() is   1352404003
rand() is    370828309
rand() is   1385419666
```

See also:
rand

std.recls

std.regexp

Regular Expressions help searching strings with complex patterns.

//to do: nice introduction, I`m not really that good in this topic but I just used it for a xml-lib

Dsource Examples

- RegExp

std.regexp.find

```
int find(rchar[] string, char[] pattern, char[] attributes = null)
```

Search **string** for the first match with **pattern** with **attributes**. Returns the index in **string** of the match if found, -1 if no match.

Example:

```
import std.stdio;
import std.regexp;

int main() {
  // find the index of the first word that contains the letter n

  char[] str = "Regular expressions can be fun!";
  int index = find(str,"[a-z]*n[a-z]*","i"); // ignore case

  writefln("Index: %d",index);

  // determine if the string contains an integer

  index = find(str,"-?[0-9]+");
  if(index == -1) {
    writefln("No integers found.");
  }
  return 0;
}
```

Output:

```
Index: 8
No integers found.
```

std.regexp.rfind

```
int rfind(rchar[] string, char[] pattern, char[] attributes = null)
```

Search **string** for the last match with **pattern** with **attributes**. Returns the index in **string** of the match if found, -1 if no match.

Example:

```
import std.stdio;
import std.regexp;

int main() {
  // find the index of the last capitalized word

  char[] str = "I think next Tuesday is Sarah's birthday.";
  int index = rfind(str,"[A-Z][a-zA-Z']*");
  if(index != -1) {
    writefln("Index: %d",index);
  }
  else {
    writefln("Not found");
```

```
  }
  return 0;
}
```

Output:

```
Index: 24
```

std.regexp.search

```
RegExp search(char[] string, char[] pattern, char[] attributes = null)
```

std.regexp.split

```
char[][] split(char[] string, char[] pattern, char[] attributes = null)
```

Split **string** into an array of strings, using regular expression **pattern** with **attributes** as the separator.

Example:

```
import std.stdio;
import std.regexp;

int main() {
  char[] str = "You can, if you want, split on any|pattern";
  char[][] words = split(str,"[,| ]+");
  foreach(word; words) {
    writefln("%s",word);
  }
  return 0;
}
```

Output:

```
You
can
if
you
want
split
on
any
pattern
```

std.regexp.sub

```
char[] sub(char[] string, char[] pattern, char[] format, char[] attributes = null)
char[] sub(char[] string, char[] pattern, char[] delegate(RegExp) dg, char[] attributes
= null)
```

Searches **string** for matches with regular expression **pattern** with **attributes** and replaces those matches with a new string composed of **format** merged with the result of the matches.

If the multiple line attribute is set, ^ and $ represent the start and end of a line. If the multiple line attribute is not set, ^ and $ represent the start and end of the entire string.

If the global attribute is set, replace all matches. Otherwise, replace the first match.

Example:

```
import std.stdio;
import std.regexp;

int main() {
  // replace numbers with #

  char[] phone = "905-555-1234";
  phone = sub(phone,"[0-9]","#","g"); // global

  writefln("%s",phone);

  char[] str;
  // wrap all vowels with brackets

  str = sub("Hello","([aeiou])","[$1]","gi"); // global, ignore case

  writefln("%s",str);

  // $` is the stuff left of the match

  // $' is the stuff right of the match

  // $& is the match itself

  // remember $` is left because ` is on the left side of the keyobard

  str = sub("Hello","[aeiou]","($`$&$')","i"); // ignore case

  writefln("%s",str);

  return 0;
}
```

Output:

```
###-###-####
H[e]ll[o]
H(Hello)llo
```

std.regexp.Regexp.replace

```
rchar[] replace(rchar[] string, rchar[] format)
```

Finds regular expression matches in **string** and replaces those matches with a new string composed of **format** merged with the result of the matches.

If the multiple line attribute is set, ^ and $ represent the start and end of a line. If the multiple line attribute is not set, ^ and $ represent the start and end of the entire string.

If the global attribute is set, replace all matches. Otherwise, replace the first match.

Example:

```
import std.stdio;
import std.regexp;

int main() {
  char[] contacts =
     "Joe: joe@asdf.com\n"
    ~"Sarah: sarah@asdf.com\n"
    ~"Rob: rob@asdf.com";
  char[] simple_email_pattern = "([a-z]+@[a-z]+\\.[a-z]+)";
  char[] mailto = "<a href=\"mailto:$1\">$1</a>";

  // wrap all emails with mailto <a> tags

  RegExp reg = new RegExp(simple_email_pattern,"gi"); // global, ignore case

  char[] html = reg.replace(contacts,mailto);
```

```
  // wrap a <b> tag around the first word of every line
  reg.compile("^([a-z]+)","gmi"); // globlal, multiline, ignore case
  html = reg.replace(html,"<b>$1</b>");
  // make every line a paragraph
  reg.compile("^(.+)$","gm"); // global, multiline
  html = reg.replace(html,"<p>$1</p>\n");
  writefln("%s",html);
  return 0;
}
```

Output:

```
<p><b>Joe</b>: <a href="mailto:joe@asdf.com">joe@asdf.com</a></p>
<p><b>Sarah</b>: <a href="mailto:sarah@asdf.com">sarah@asdf.com</a></p>
<p><b>Rob</b>: <a href="mailto:rob@asdf.com">rob@asdf.com</a>
```

std.socket

std.socketstream

std.stdint

std.stdio

This module implements standard I/O functions.

Dsource Examples

- writef and printf

std.stdio.writef

```
void writef( s [, s[, s[...]]] );
```

Formats all arguments per "format strings" and writes them to *stdout*.

Example:

```
// Main function

int main(char[][] args)
{
  char[] world = "world";
  writef("Hello"," ",world~"! ",1024,'\n');
  // Alternate method.

  writef("Hello %s! %d\n",world,1024);
  return 0;
}
```

Output:

```
Hello world! 1024
Hello world! 1024
```

See also:
writefln, fwritef

std.stdio.writefln

```
void writefln( s [, s[, s[...]]] );
```

Does the same as **writef**, but appends a newline to the output.

See also:
writef, fwritef, fwritefln

std.stdio.fwritef

```
void fwritef(FILE* fp, s[, s[, [...]]]);
```

Does the same as writef, but writes the concatenated arguments to the stream **fp**,.

See also:
writef, fwritefln

std.stdio.fwritefln

```
void fwritefln(FILE* fp, s[, s[, s[...]]]);
```

Does the same as writefln, sents concated arguments to the stream fp instead of stdout.

See also:
writefln, fwritef

std.stream

This module defines the InputStream and OutputStream interfaces as well as the Stream class for general Stream I/O. There are also some implementations of Stream for:
* a Stream wrapper for buffering streams
* a Stream wrapper for endian conversion
* file I/O (Raw and buffered)
* array stream
* memory stream
* MmFileStream
* SliceStream (Wrapper for a slice of another stream)

Dsource Examples

- File Input
- FileMode
- I/O
- ReadLine

std.stream.InputStream

```
interface InputStream
```

The interface InputStream defines function for readable Streams.

std.stream.InputStream.readExact

```
void readExact(void* buffer, size_t size);
```

Read exactly size bytes into the buffer, throwing a ReadException if it is not correct.

std.stream.InputStream.read

```
size_t read(ubyte[] buffer);
void read(out byte x);
void read(out ubyte x);
void read(out short x);
void read(out ushort x);
void read(out int x);
void read(out uint x);
void read(out long x);
void read(out ulong x);
void read(out float x);
void read(out double x);
void read(out real x);
void read(out ifloat x);
void read(out idouble x);
void read(out ireal x);
void read(out cfloat x);
void read(out cdouble x);
void read(out creal x);
void read(out char x);
void read(out wchar x);
void read(out dchar x);
```

Read a basic type or counted string, throwing a ReadException if it could not be read. Outside of byte, ubyte, and char, the format is implementation-specific and should not be used except as opposite actions to write.
The first call is somewhat special in that it reads a block of data to fill the buffer array. It reads *buffer.length* bytes at maximum and returns the number of bytes read.

std.stream.InputStream.readLine

```
char[] readLine();
char[] readLine(char[] result);
wchar[] readLineW();
wchar[] readLineW(wchar[] result);
```

Reads a line from the stream. The line can be terminated by either CR, LF, CR/LF, or EOF. The readLineW versions of the function reads Unicode wchars.

If you already have a char array, you can use the second version of the *readLine* or *readLineW* functions, as these will try to reuse your char array and reallocate memory if neccessary.

std.stream.InputStream.opApply

```
int opApply(int delegate(inout char[] line) dg);
int opApply(int delegate(inout ulong n, inout char[] line) dg);
```

```
int opApply(int delegate(inout wchar[] line) dg);
int opApply(int delegate(inout ulong n, inout wchar[] line) dg);
```

This operator (for the foreach statement) iterates through the stream line-by-line.
Breaking the foreach loop will leave the Stream at the position after line last read.
The string storing the line may be reused in each iteration of the loop.

For example:

```
Stream file = new BufferedFile("sample.txt");
foreach(ulong n, char[] line; file) {
  stdout.writefln("line %d: %s",n,line);
}
file.close();
```

std.stream.InputStream.readString

```
char[] readString(size_t length);
wchar[] readStringW(size_t length);
```

Reads a string of the given length, *readStringW* is the unicode wide-char version.

std.stream.InputStream.getc

```
char getc();
wchar getcw();
```

Reads the next character from the stream and handles characters pushed back by std.stream.InputStream.ungetc. *getcw* is for Unicode wide-strings, as usual.

std.stream.InputStream.ungetc

```
char ungetc(char c);
wchar ungetcw(wchar c);
```

Pushes one character back on the stream, it will be the next character read by std.stream.InputStream.getc. *ungetcw* is for Unicode wide-chars, as usual.

std.stream.InputStream.scanf

```
int vscanf(char[] fmt, va_list args);
int scanf(char[] format, ...);
```

Scan a string from the input using a similar form to C's scanf.

std.stream.InputStream.available

```
size_t available();
```

Returns the number of bytes that can be read immediately.
Network sockets, for example, return the number of bytes currently held in the buffer - reading

more than this would require you to wait for further packets.

std.stream.InputStream.eof

```
bool eof();
```

Return whether the current file position is the same as the end of the file. This does not require actually reading past the end, as with stdio. For non-seekable streams this might only return true after attempting to read past the end.

std.stream.InputStream.isOpen

```
bool isOpen();
```

Return true if the stream is currently open.

std.stream.OutputStream

std.stream.Stream

std.string

This module provides basic string operations:

incomplete
std.string has more useful stuff, but i was too lazy :-)

See also:
Array Concatenation
Dsource Example
- split and splitLines
- Maze Solver

std.string - Constants

```
char[] hexdigits  = "0123456789ABCDEF";
char[] digits     = "0123456789"
char[] octdigits  = "01234567"
char[] lowercase  = "abcdefghijklmnopqrstuvwxyz"
char[] uppercase  = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
char[] letters    = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
char[] whitespace = " \t\v\r\n\f"
```

See also:
StringException

std.string.StringException

```
class StringException
```

Thrown on errors in string functions.

See also:
Try statement, Throw statement

std.string.atoi

```
long atoi(char[] s)
```

This function converts a string of to a long integer.

If s is not convertable (for example if s=="a"), atoi should create a **StringException**, but it does not!
Note: atoi returns zero or the first part of numbers it could convert.

Example:

```
try {
  writefln(atoi(cast(char[])"55"));
} catch {
  writefln("Exception with '55'");
}
try {
  writefln(atoi(cast(char[])"55a"));
} catch {
  writefln("Exception with '55a'");
}
try {
  writefln(atoi(cast(char[])"z"));
} catch {
```

```
  writefln("Exception with 'z'");
}
```

Output:

```
55
55
0
```

See also:
Try statements, atof, toString

std.string.atof

```
real atof(char[] s)
```

This function converts a string to a 32 bit floating point number.

See also:
atoi, toString

std.string.toString

```
char[] toString(bit arg)
char[] toString(char arg)
char[] toString(byte arg)
char[] toString(ubyte arg)
char[] toString(short arg)
char[] toString(ushort arg)
char[] toString(int arg)
char[] toString(uint arg)
char[] toString(long arg)
char[] toString(ulong arg)
char[] toString(float arg)
char[] toString(double arg)
char[] toString(real arg)
char[] toString(ifloat arg)
char[] toString(idouble arg)
char[] toString(ireal arg)
char[] toString(cfloat arg)
char[] toString(cdouble arg)
char[] toString(creal arg)
```

As you can see, the toString method really can convert integer and floating point numbers easily.
In addition you can define **radix** to convert an unsigned integer in a radix from 2 to 36:

```
char[] toString(long arg, uint radix);
char[] toString(ulong arg, uint radix);
```

Note: The characters **A** through **Z** are used to represent values 10 through 36.
Note: **arg** is treated as a signed value only if radix is 10!

Of course there is also a function to convert a C-style 0-terminated string **s** to *char[*.

```
char[] toString(char* s)
```

std.string.tolower

```
char[] tolower(char[] s)
```

Returns **s** in lower chars.

See also:
toupper, lowercase

std.string.toupper

```
char[] toupper(char[] s)
```

Returns **s** in upperchars.

See also:
tolower, uppercase, capitalize

std.string.capitalize

```
char[] capitalize(char[] s)
```

Capitalizes first character of string.

See also:
toupper, capwords

std.string.capwords

```
char[] capwords(char[] s)
```

Capitalizes all words in string, removes leading and trailing whitespace and replaces all sequences of whitespace with a single space.

See also:
capitalize, toupper

std.string.splitlines

```
char[][] splitlines(char[] s)
```

Spilts **s** into an array of strings, using CR, LF, or CR-LF as delimiter.

See also:
find, split

std.string.find

```
int find(char[] s, dchar c)
int find(char[] s, char[] sub)
```

Find first occurrance of **sub** or **c** in string **s**. Return index where it was found, otherwise returns -1.

There are also case-sensitive versions available:

```
int ifind(char[] s, dchar c)
int ifind(char[] s, char[] sub)
```

See also:
splitlines, rfind

std.string.rfind

```
int rfind(char[] s, dchar c)
int rfind(char[] s, char[] sub)
```

Find last occurrance of **sub** or **c** in string **s**. Return index where it was found, otherwise returns -1.

There are also case-sensitive versions available:

```
int ifind(char[] s, dchar c)
int ifind(char[] s, char[] sub)
```

See also:
find

std.string.split

```
char[][] split(char[] s)
char[][] split(char[] s, char[] delim)
```

Splits **s** into an array of words, using **delim** or whitespace as the delimiter.

See also:
join, splitlines

std.string.join

```
char[] join(char[][] s, char[] sep)
```

Joins the array of strings **s** into a string, using **sep** as the delimiter.

See also:
split

std.system

std.thread

std.uri

std.utf

std.zip

Read/write data in the zip archive format. Makes use of the etc.c.zlib compression library.

Dsource Examples

- zip

std.zip.ZipException

std.zip.ArchiveMember

std.zip.ArchiveMember.madeVersion

std.zip.ArchiveMember.extractVersion

std.zip.ArchiveMember.flags

std.zip.ArchiveMember.compressionMethod

std.zip.ArchiveMember.time

std.zip.ArchiveMember.crc32

std.zip.ArchiveMember.compressedSize

std.zip.ArchiveMember.expandedSize

std.zip.ArchiveMember.diskNumber

std.zip.ArchiveMember.internalAttributes

std.zip.ArchiveMember.externalAttributes

std.zip.ArchiveMember.name

std.zip.ArchiveMember.extra

std.zip.ArchiveMember.comment

std.zip.ArchiveMember.compressedData

std.zip.ArchiveMember.expandedData

std.zip.ZipArchive

std.zip.ZipArchive.data

std.zip.ZipArchive.diskNumber

std.zip.ZipArchive.diskStartDir

std.zip.ZipArchive.numEntries

std.zip.ZipArchive.totalEntries

std.zip.ZipArchive.comment

std.zip.ZipArchive.directory

std.zip.ZipArchive.addMember

std.zip.ZipArchive.deleteMember

std.zip.ZipArchive.build

std.zip.ZipArchive.expand

std.zlib

Compress/decompress data using the zlib library.

Dsource Example

- zlib

std.zlib.ZlibException

```
class ZlibException: object.Exception;
```

Errors throw a ZlibException.

Example
Todo

std.zlib.adler32

```
uint adler32(uint adler, void[] buf);
```

Compute the Adler32 checksum of the data in buf[. adler is the starting value when computing a cumulative checksum.

Example
Todo

std.zlib.crc32

```
uint crc32(uint crc, void[] buf);
```

Compute the CRC32 checksum of the data in buf[. crc is the starting value when computing a cumulative checksum.

Example
Todo

std.zlib.compress

```
void[] compress( void[] srcbuf, int level );
```

```
void[] compress( void[] buf );
```

Compresses the data in srcbuf[using compression level level. The default value for level is 6, legal values are 1..9, with 1 being the least compression and 9 being the most. Returns the compressed data.

Example
Todo

std.zlib.uncompress

```
void[] uncompress(void[] srcbuf, uint destlen = 0u, int winbits = 15);
```

Decompresses the data in srcbuf[.

Parameters: uint destlen (size of the uncompressed data. It need not be accurate, but the decompression will be faster if the exact size is supplied.)

Returns: The decompressed data.

Example
Todo

std.zlib.Compress

```
class Compress;
```

Used when the data to be compressed is not all in one buffer.

```
this(int level);
```

```
this();
```

Construct. level is the same as for std.zlib.compress().

Example
Todo

std.zlib.Compress.compress

```
void[] compress(void[] buf);
```

Compress the data in buf and return the compressed data. The buffers returned from successive calls to this should be concatenated together.

Example
Todo

std.zlib.Compress.flush

```
void[] flush(int mode = 4);
```

Compress and return any remaining data. The returned data should be appended to that returned by compress().

Params: int mode: one of the following:
- Z_SYNC_FLUSH
- Z_FULL_FLUSH
- Z_FINISH

Example
Todo

std.zlib.UnCompress

```
class UnCompress;
```

Used when the data to be decompressed is not all in one buffer.

```
this(uint destbufsize);
```

```
this();
```

Construct. destbufsize is the same as for std.zlib.uncompress().

Example
Todo

std.zlib.UnCompress.uncompress

```
void[] uncompress(void[] buf);
```

Decompress the data in buf and return the decompressed data. The buffers returned from successive calls to this should be concatenated together.

Example
Todo

std.zlib.UnCompress.flush

```
void[] flush();
```

Decompress and return any remaining data. The returned data should be appended to that returned by uncompress(). The UnCompress object cannot be used further.

Example
Todo

Other common libraries and tools

Add your library here!

I'd be happy if we get derelict here... :)

The Mango Tree

Mango is a collection of D packages with an orientation toward server-side programming. These packages may be used together or in isolation and, in many cases, can be just as applicable to client-side development. Mango is targeted for Win32 and linux platforms.

You may visit the official site and download the newest version.
There are also good examples on how to use mango.

This packages are located under the Mango Tree:

Dsource Examples

- Mango I/O

mango.io

This is a high-performance buffered IO package. Primary functionality includes memory Buffers, external Conduits (files, sockets, etc), file-system manipulation and many more..

mango.io.FileConduit

```
class FileConduit : Conduit, ISeekable
```

Implements a means of reading and writing a generic file. File conduit extends the generic conduit by providing file-specific methods to set the file size, seek to a specific file position, and so on. Also provided is a factory for create a memory-mapped Buffer upon a file.

See also:
Constructor of FileConduit, Conduit

mango.io.FileConduit.FileConduit.this

```
this (char[] name, FileStyle style = FileStyle.ReadExisting);
this (FileProxy proxy, FileStyle style = FileStyle.ReadExisting);
this (FilePath path, FileStyle style = FileStyle.ReadExisting);
```

Use FileConduit to create a new file handler.

Example:

```
FileConduit fc = new FileConduit ("test.txt");

// stream directly to console

Stdio.stdout.copy (fc);
```

mango.io.SocketConduit

```
class SocketConduit : Socket, IConduit, ISocketReader
```

A wrapper around the bare Socket to implement the IConduit abstraction.

See also:
Conduit

mango.io.Conduit

```
class Conduit : Resource, IConduit
```

Conduits are the primary means of accessing external data, and are usually routed through a Buffer.

See also:
FileConduit, SocketConduit

mango.http

HTTP client and server can be realised with this package.

mango.http.server

An HTTP server.

mango.http.client

An HTTP client, that support headers, query parameters, cookies, timeout etc.

mango.http.utils

mango.servlet

A servlet-style engine that sits upon mango.server.
Includes most of the things that Java servlet programmers would expect.

mango.cache

Some simple caching mechanisms. Includes MRU caching via a queue, and level-two caching (to disk) via class-serialization.

Note: The mango.cluster package derives from mango.cache, so it's really easy to switch between a local cache implementation and a clustered version.

mango.log

An implementation of the popular Log4J package, for logging and tracing of runtime behaviour. Mango.log exposes an extensible and customizable framework, has been optimised heavily for minimal runtime overhead, does not impact the GC once initialised, and operates with either Mango.io or Phobos streams. Mango.log can generate remote log messages for use with Chainsaw, and has a browser-based 'Administrator' which allows remote, dynamic inspection and adjustment of the log settings within an executing program.

mango.cluster

A clustering package for servers and other network aware applications.

mango.icu

Set of wrappers around the ICU I18N project. Mango.icu exposes the C API in a manner that takes advantage of D arrays and so on. See UCalendar, UChar, UConverter, UDateFormat, ULocale, UMessageFormat, UNumberFormat, UResourceBundle, UString, UText, UCollator, USet, UTransform, USearch, UNormalize, UDomainName, UBreakIterator, URegex, and UTimeZone. Numeric formatting is handled by a range of subclasses,including UDecimalFormat, UCurrencyFormat, UPercentFormat, UScientificFormat, and USpelloutFormat. There's a set of adapter classes in UMango which can be used to bind the ICU converters to Mango.io, thereby enabling Reader/Writer streaming with the full suite of ICU transcoders.

mango.convert

Includes a range of formatting modules, including Sprint, Format, Integer, Double, and Atoi. The precision floating-point converters from David Gay are wrapped as DGDouble, and can be configured instead of Double. There's also a date parser and formatter, Rfc1123. Finally, there's a complement of high-performance Unicode converters, along with a module for handling unicode BOM. Mango.convert is templated for char, wchar, and dchar. The printf() compatable Sprint and Print convert char, wchar, and dchar arguments appropriately for the target representation.

mango.sys

Includes a number of low-level utilities such as ByteSwap?, and various O/S-specific functions. Also includes a great time-conversion module, Epoch, which hides the complexity of the relevant O/S functions.

mango.text

Unicode token-parsing and string handling. Includes String, ImmutableString, SimpleIterator, QuoteIterator, RegexIterator and LineIterator. Iterators support streaming via internal use of IBuffer, which also enables lockstep reading across multiple iterator (and reader) instances when bound to a common source.

The Build Utility

This is a tool to make it easier to build applications from their component source code, object code, and libraries. Traditionally, tools such as *make* were needed to pull together the related files to form an application. This utility works differently as it analyzes the source code and all its related files, to determine which files need compiling or recompiling, and which need linking. It then constructs the appropriate command lines for kicking off the process.

You can get a copy of the tool from

The DSource **Build** Project

For example, if the top level file in your application was called *myapp.d*, typically all you would need to do to compile and link this is

```
build myapp
```

There is no need to create and maintain a *makefile* as all the relationships are worked out in realtime.

Related Links

Websites
Digitalmars, the home of D http://www.digitalmars.com/d/index.html

Forums
German Community http://sprungmarke.net

Wikis

DocWiki, infos, examples, more examples:) http://www.docwiki.net
dsource

About this documentation

Contributors:

Markus Dangl < Sky >
sky AT quit-clan.de

Alexander Panek < JimPanic >
alexander.panek AT brainsware.org

Stefan Dangl < StangLS >
stangls AT quit-clan.de

Derek Parnell
ddparnell *AT* bigpond *DOT* com

Dieter Dannerbeck
no.spam.dd *AT* gmx.de

Documentation changelog:

2006-01-17 by dannerbeck_dieter:
Finished std.date, wrote zlib. Related Links section and changes on "A simple Tutorial" to make it SkyOS friendly:)

2005-02-14 by Sky:
Wrote std.ctype and std.format

2005-01-29 by StangLS:
Wrote documentation for std.md5 and a little bit of the mango library.

2005-01-30 by Sky:
Moved most of the questions from the index to seperate pages in "About this documentation".
Wrote the first edition of the style guide.
Changed the "long title" of all phobos pages to comply with the style guide.

2005-01-29 by StangLS:
- added some documentation to std.conv, std.data, std.file, std.math, std.path, std.random, std.stdio and std.string
- added 2.3. Types
- added 2.2.7. Try statement and 2.2.8. Throw statement
- wrote a tutorial, and some text to 1.3.2. From Java

2005-01-27 by Sky:
Added the for statement and did std.base64 & std.compiler
Fixed a few typos in the tutorial and changed come [code] tags to [quote] tags where appropriate

2005-01-27 by Hannibal:
Added a few words about pre-/post-incrementals (i++; ++i),*this* and *super()* , switch- and do-while-statement.

2005-01-27 by Sky:
I began by writing the "tutorial" and some small things.