The D Programming Language

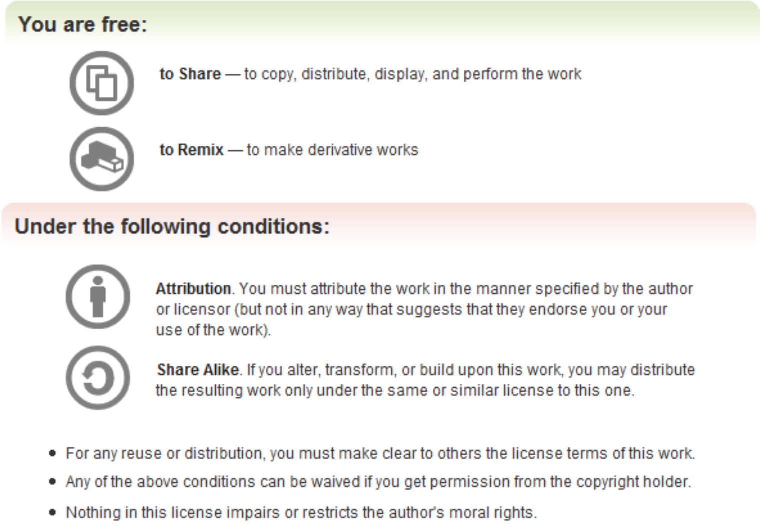
The D^{2}azel Team

EPITA

July, 15t̂h 2007

Licence

This work is licenced under the Creative Commons Attribution-Share Alike 3.0 License.



Version history

Version history

- **1.1** - Jérémie Roquet (July 15th 2007)
- **1.0** - The D^{2}azel team (July 12th 2007)
- **0.2** - Jérémie Roquet (July 10th 2007)
- **0.1** - Paul Baudron (June 28th 2007)

Introduction

Subject

- The D programming language
- The tools
- The libraries

Presentation

- A short introduction to basic concepts
- To make you discover a cool language
- Not a coding demonstration

Outline

Presentation

- The language
- The tools

Language

- Basic features
- Comparison with other languages

Libraries

- What already exists
- What is comming

Part I

Presentation

Introduction

History

- by Walter Bright of Digital Mars
- Appeared in 1999
- A stable version, 1.0, was released on January 2, 2007

Licence

- Proprietary
- Open source

Features

Features

- Compiled
- Imperative
- object-oriented
- Metaprogramming
- Garbage collector

Influenced by C (1972), C++ (1985), C# (2001), Java (1995), Eiffel (1986)

Why D?

Why D?

- To combine the power and high performance of C and C++ with the programmer productivity
- Needs of quality assurance, documentation, management, portability and reliability.

Garbage Collection

Methods for allocating memory in D

- Static data, allocated in the default data segment
- Stack data, allocated on the CPU program stack
- Garbage collected data, allocated dynamically on the garbage collection heap

Garbage Collection

Garbage collected programs can be faster

- Most destructors are empty (so they are not executed)
- No special mechanism to establish if an exception happens
- Less code necessary to manage memory (smaller programs)
- When memory is not tight, the program runs at full speed
- Modern garbage collectors do heap compaction

Garbage Collection

Other advantages

- No memory leaks
- Few hard-to-find pointer bugs
- Faster to develop and debug

Garbage Collection

Downsides

- The program can arbitrarily pause
- The time it takes for a collection to run is not bounded
- Programs must carry around with them the garbage collection implementation

Garbage Collection

Where/When is it run?

- In a special thread

How it works?

- Looking for all pointers “roots”
- Recursively scanning all allocated memory
- Freeing all memory that has no active pointers to it
- Possibly compacting the remaining used memory

Compilers

Two implementations

- Digital Mars DMD (Win32 and x86 Linux)
- GCC D Compiler (Windows, MAC OS X and Linux)

Linkage

Programs:

- Are linked with gcc
- Can be linked with C libraries...

dsss

- Build system
- Package management
- Dependance management

Editors

Editors

- Emacs
- Eclipse
- ...

Part II

Language

Hello world

The code

```
import std.stdio;

void main(char[][] args)
{
    writefln("Hello World !");
}
```

Some explanations

- Import the writefln function
- Implement the program main function
- Write "Hello World !" to standard output

Comments

Single line comments

```
// This is a comment
```

Multi line comments

```
/* This is a
        multi line comment */
```

Multi line nested comments

```
/+ This is a multi line
        /+ Nested +/
        comment +/
```

Documentation

Single line documentation

```
/// This is a documentation
```

Multi line documentation

```
/** This is a
        multi line documentation */
```

Multi line nested documentation

```
/++ This is a multi line
       /+ Nested +/
        documentation +/
```

Documentation

Standard sections

- Generate standard documentation
- Authors, bugs, date, deprecated, history, license, version
- Code examples
- Return value, exceptions

Module declaration

Declaration

```
module moduleName;
module packageName.moduleName;
```

- Implicit scope
- Compiled only once
- May contain any type of code

Import

Simple module importation

```
import moduleName;
import module1,
        module2,
        module3;
```

- No file inclusion
- No header parsing
- Load symbols from binary or look in memory

Import

Public module importation

```
public import moduleName;
```

- Transfer symbols

Import

Static module importation

```
static import moduleName;
```

- Force full namespace qualification
- Prevent name collisions

Access to current module

```
.myFunction();
```

Import

Module importation & renaming

```
import moduleName = newName;
```

- Prevent name collisions
- Ease dependancies change

Import

Selective module importation

```
import moduleName : elementName;
```

- Prevent name collisions

Selective module importation & renaming

```
import moduleName : elementName = newElementName;
```

Mixin

Simple module mix

```
mixin(moduleName);
```

- Code factorisation
- No file inclusion
- No header parsing
- Load symbols from binary or look in memory

Simplest types

Void

Means no type

Bool

Always initialized to false

Integers

Declaration

```
int i = 42; // Decimal
int j = 0b101010; // Binary
int k = 052; // Octal
int l = 0x2A; // Hexadecimal
```

With underscores

```
int i = 65_535; // 65535
int j = 0x34_32; // [0, 0, '4', '2']
```

Integers

And also

- byte, ubyte
- short, ushort
- int, uint
- long, ulong
- cent, ucent

Always initialized to 0

Floating point numbers

Different types

- float
- double
- real

Always initialized to NaN

Floating point numbers

Decimal exponential notation

```
float f = 1.0e3;  // f = 1000
float g = 1.0e+3; // g = 1000
float h = 1.0e−3; // h = 0.001
```

Hexadecimal exponential notation

```
float f = 0x1.Ap3;  // f = 13
float g = 0x1.Ap+3; // g = 13
float h = 0x1.Ap−3; // h = 13/64
```

Imaginary numbers

Different types

- ifloat
- idouble
- ireal

Always initialized to NaN * 1.0i

Complex numbers

Different types

- cfloat
- cdouble
- creal

Always initialized to NaN * (1.0 + 1.0i)

Characters

Different types

- char
- wchar
- dchar

- char initialized to 0xFF
- wchar initialized to 0xFFFF
- dchar initialized to 0x0000FFFF

Pointers

Declaration

```
int i = 42;
int* pi; // int pointer
pi = &i; // pi points to i
int j = *pi; // j = 42
```

Multiple variable definition

```
int i, j; // Integers
int i, *j; // Error, not the same type
int* i, j; // Pointers to integer
```

Always initialized to NULL

Arrays

Static array

```
int[42] tab; // Array of 42 elements
int tab[42]; // Works also
```

- Fields initialized to the default type value
- Limited to 16Mib

Dynamic array

```
int[] tab; // Empty dynamic array
int tab[]; // Works also
```

- Initialized with length of 0

Arrays

Jagged array

```
int[][] tab; // Empty matrix
```

- Rows can be of different lengths

Multidimensional array

```
int[3][3] tab; // Fixed size matrix
int[3, 3] tab; // Work also
```

- Contiguous in memory

Arrays

Properties

```
int[42] tab;
int l = tab.lenght; // Gives the array length
```

Complex constructions

C++

```
// Array of 42 pointers to arrays of pointers to int
int* (*i[42])[];
```

D

```
// Array of 42 pointers to arrays of pointers to int
int*[]*[42] i;
int* (*i[42])[]; // Works also
```

Strings

Strings ?

- There is no string in D
- We use characters arrays instead

Declaration

```
char[] str = "toto"; // This is a UTF-8 string
wchar[] str2 = "hello"w; // This is a UTF-16 string
dchar[] str3 = "42"d; // This is a UTF-32 string
char[] strx = x"FFAA"; // This is a hexadecimal liter
```

Strings

Multiple lines strings

```
char[] loremIpsum = "Lorem ipsum dolor sit amet,
consetetur sadipscing elitr.
Sed diam nonumy eirmod tempor invidunt ut
labore et dolore magna aliquyam erat.
Sed diam voluptua. At vero eos et accusam et
justo duo dolores et ea rebum.
Stet clita kasd gubergren, no sea sanctus est.";
```

Strings

Special characters in C++

```
char hello[] = "Hello world!\n";
char regexp[] = "\"[^\\\\]*(\\\\.[^\\\\]*)*\"";
```

Special characters in D

```
char[] hello = "Hello world!" \n;
char[] regexp = \" r"[^\\]*(\\.[^\\]*)*" \";
char[] regexp = \" ‘[^\\]*(\\.[^\\]*)*‘ \";
```

Associative arrays

Declaration

```
// Integer associative array indexed by strings
float[char[]] price;
// Add an expensive fruit to the array
price["apple"] = 2.40;
price.remove("apple"); // Remove it
```

Functions & delegates

Functions

```
// Function taking an array and returning an integer
int function(int[]) myFunction;
// Dummy function
int toto(int[] titi){ return titi[length - 1] }
myFunction = &toto; // refers to toto
int i = myFunction([42]); // i = 42
// The C function pointer syntax works also
int (*anotherFunction)(int[]);
```

Delegates

Works also for nested functions or non-static methods

Functions & delegates

Declaration

```
// Function taking an array and returning an integer
int delegate(int[]) myFunction;
void outerFunction
{
        // Dummy inner function
        int toto(int[] titi){ return titi[0] }
        myFunction = &toto; // refers to toto
        int i = myFunction([42]); // i = 42
}
```

Arrays manipulation

Initialization

```
int[42] fourtyTwos; // Array of 42 integers
fourtyTwos[] = 42; // Set all fields to 42
fourtyTwos[21..42] = 0; // Set fields 21 to 42 to 0
fourtyTwos[30..$] = 21; // Set last fields to 21
```

Array assignement

```
// tab[21] = 42, tab[22] = 21
int[42] tab = [21:42, 21];
```

Arrays manipulation

Copy

```
int[42] first;
int[42] second;
first = second; // first and second are the same
first[] = second[] // copy second into first
first[0..2] = second[2..4] // copy two fields
```

Slicing

```
int[42] first;
int[] second;
// second has the 21 first elements of first
second = first[0..21];
```

Arrays manipulation

Concatenation

```
int[] first;
int[] second;
int[] third;
third = first ~ second;
first ~= second;
```

Arrays manipulation

Properties

```
int[] tab;
tab.sort();
tab.reverse();
tab.dup();
++tab.length;
```

Arrays manipulation

Functions taking arrays

```
int[] stack;
void push(int[] tab, int elt)
{
        ++tab.length;
        tab[length - 1] = elt;
}
tab.push(42);
```

opNeg

Usage

```
-myVar;
```

Binary operators

Usage

```
a + b;  a - b;
a * b;  a / b;
a % b;
a & b;  a | b;
a ~ b;
a << b;
a >> b;  a >>> b;
```

- opAdd, opSub, opMul, opDiv, opMod
- opAnd, opOr, opXor
- opCat
- opShl, opShr, opUShr

Comparison operators

Usage

```
a == b;
a != b;
a < b;
a > b;
a <= b;
a >= b;
```

- opEquals
- opComp

Reverse binary operators

Usage

```
a + b; a - b;
a * b; a / b;
a % b;
a & b; a | b;
a ~ b;
a << b;
a >> b; a >>> b;
```

- opAdd_r, opSub_r, opMul_r, opDiv_r, opMod_r
- opAnd_r, opOr_r, opXor_r
- opCat_r
- opShl_r, opShr_r, opUShr_r

Floating point operators

Usage

```
a == b;
a != b;
a !< b;
a !> b;
a !<= b;
a !>= b;
a != b;
a <> b;
a !<> b;
a <>= b;
a !<>= b;
```

- Builtin NaN handling

opAssign

Usage

```
a = b;
a += b; // ++
a -= b; // --
a *= b; a /= b;
a %= b;
a &= b; a |= b;
a ~= b;
a <<= b;
a >>= b; a >>>= b;
```

- Any operator, suffixed by "Assign"
- Example : opAddAssign

opCast

Usage

```
class MyClass
{
    MyType opCast()
        {
                return myValue;
        }
}
cast(MyType) myVar;
```

- Allow complex conversions
- Almost transparent to the user

opCall

Usage

```
MyClass f;
f(myArg);
```

- Allow function objects (functors)

opIndex, opIndexAssign

opIndex

```
class MyClass
{
    MyType opIndex(MyType index1, MyType index2)
        {
            return myValue;
        }
}
myVar[i, j];
```

opIndex, opIndexAssign

opIndexAssign

```
class MyClass
{
    MyType opIndexAssign(MyType value,
                    MyType index1, MyType index2)
        {
        }
}
myVar[i, j] = myValue;
```

opSlice, opSliceAssign

opSlice

```
class MyClass
{
    MyType opSlice(MyType start, MyType end)
        {
                return mySub;
        }
}
myVar[i..j];
```

opSlice, opSliceAssign

opSliceAssign

```
class MyClass
{
    MyType opSliceAssign(MyType value,
                         MyType start, MyType end)
        {
        }
}
myVar[i..j] = myValue;
```

opIn, opIn_r

Usage

```
a in b;
// Not yet implemented
a !in b;
```

- determine if an object is in an array
- may be redefined

is operator

Usage

```
a is b;
a !is b;
```

- determine if a reference is null
- determine if two references are equals
- cannot be redefined

is operator

is type

```
static if (is (T[]))
        // Something
```

- type category

is operator

is type specialisation

```
static if (is (T: BaseType))
        // Something
```

- type hierarchy

is operator

is type identity

```
static if (is (T == MyType)) // ...
static if (is (T == super)) // ...
static if (is (T == class)) // ...
static if (is (T == function)) // ...
static if (is (T == return)) // ...
```

- type hierarchy

Goto

Usage

```
myLabel:
// Many useful things
goto myLabel;
```

- Quick and easy
- Dirty

If else

Usage

```
if (condition)
{
        // Something
}
else
{
        // Something else
}
```

- 'condition' must be a boolean

Switch

Usage

```
switch (token)
{
        case "if":
                parseIf();
                goto case "else";
        case "else":
                parseElse();
                break;
        default:
                throw new EUnknownToken();
}
```

- Works also on arrays

While loop

Usage

```
while (condition)
{
        break;
        continue;
}
```

Do ... while

```
do
{
        break;
        continue;
} while (condition)
```

For loop

Usage

```
for (initialization; condition; modification)
{
        break;
        continue;
}
```

Foreach loop

Usage

```
char[] str = "bonjour";
foreach (char c; str)
{
        writefln("%c", c);
}
```

foreach_reverse

- The same, but in reverse order

Foreach loop

With additional parameter

```
char[] str = "bonjour";
foreach (uint i, char c; str)
{
        writefln("%ui -> %c", i, c);
}
```

Foreach loop

On associative arrays

```
float[char[]] prices;
prices["apple"] = 2.40;
foreach (char[] item, float price; prices)
{
        writefln("%s -> %f", item, price);
}
```

Foreach loop

On objects

```
class myObject
{
        int opApply(int delegate(ref int) myFunc)
        {
                // Applies myFunc to each element
        }
}
foreach (int elt , myObject o)
{
        // Do something with elt
}
```

Foreach loop

On delegates

```
void apply(void delegate(int) func)
{
        func(42);
        func(21);
}
foreach (int elt , apply(void delegate(int)))
{
        writefln("%i", elt);
}
```

Named loops

Named while with named break

```
outerLoop: while (condition)
{
        innerLoop: for (uint i = 0; i < 42; ++i)
                break outerLoop;
}
```

Named for with named continue

```
outerLoop: for (uint i = 0; i < 42; ++i)
{
        innerLoop: while (condition)
                continue outerLoop;
}
```

Static scopes

Using static if

```
static if (condition)
        int myVar;
else
        float myVar;
```

- Evaluated at compile time (like # if in C++)
- Has access to D values (templates. . .)

Virtual functions

Declaration of a virtual function in C++

```
virtual myMethod();
```

- Non virtual by default

Declaration of a final function in D

```
final myMethod()
{
        // Cannot be overriden unless private
}
```

- Virtual by default

Default policies

Scalars, unions, pointers and structs

- Value types
- Passed as copy by default

Arrays and objects

- Reference types
- Passed as reference by default

Explicit policies

in, out policies

```
MyType myFunction(in MyType myIn, out MyType myOut)
{
}
```

- Safer argument policy
- Force references

Explicit policies

ref, inout policies

```
MyType myFunction(ref MyType myRef,
       inout MyType myInOut)
{
}
```

- Force reference
- ref is deprecated

Explicit policies

lazy policy

```
MyType myFunction(lazy myArg)
{
}
myFunction(myVar++);
```

- Evaluated if necessary
- Optimized
- Safe

Forward references

No need to declare before use

```
void first()
{
   second();
}
void second()
{
}
```

Function literals

No need to declare before use

```
void myFunction(void delegate(int) d)
{
    int a = d(42);
}
myFunction(delegate(int i) { return i + 1 });
```

Nested functions

Usage

```
MyType myOuterFunction()
{
        MyType myVar;
        MyType myInnerFunction()
        {
        }
}
```

- Access to outer function scope

Variadic functions

Usage

```
MyType myFunction(...)
{
}
MyType myFunction(int[] myArg...)
{
}
```

- Variable arguments count

Extern functions

Usage

```
extern (C) printf();
extern (D) writefln();
// Not yet implemented
extern (C++) makeCoffee();
```

- C and D are supported
- C++ is in progress
- Pascal should be supported on Windows

Type alias

Declaration

```d
alias uint score;
score myScore = 42;
uint uintScore = myScore;

alias std.c.string cstr;
alias myVar.myField[myIndex] myAlias;
```

- Compatible with the original type
- Work also for variables, modules or any symbol

Type definition

Declaration

```
typedef uint score = 42;
int toUint(score s)
{
        return cast(uint)s;
}
```

- Strong type identity
- Custom initializer

Enumerations

Declaration

```
enum MyEnum : MyType
{
        MY_FIRST_VALUE,
        MY_SECOND_VALUE = myValue, // ',' is allowed
} // ';' is forbidden
```

- Default type is int
- Default initializer is the first value

Unions

Declaration

```
union MyUnion
{
        myType1 myField1;
        myType2 myField2; // ';' is allowed
} // ';' is forbidden
```

- Value type
- P.O.D. (No identity)
- No inheritance

Unions

Usage

```
MyUnion myVar =
{
        myField = myValue
};
```

Structs

Declaration

```
struct MyStruct
{
        myType1 myField1 = myValue;
        myType2 myField2;
} // ';' is forbidden
```

- Value type
- P.O.D. (No identity)
- No inheritance

Structs

Usage

```
MyStruct myVar =
{
        myValue1,
        myValue2
};
```

Struct literal

```
myFunction(MyStruct(myValue1, myValue2));
```

Structs

Explicit fields initialization

```
MyStruct myVar =
{
        myField2: myValue1,
        myField1: myValue2
};
```

Structs

Struct initializer

```
struct MyStruct
{
        MyType myField;
        static MyStruct opCall(MyType myValue)
        {
                MyStruct myVar;
                myVar.myField = myValue;
                return myVar;
        }
}
```

Structs

Fields alignment

```
struct MyStruct
{
        align(32) MyType myField1;
        align(32) MyType myField1;
}
```

Interfaces

Declaration

```
interface MyInterface
{
        MyType1 myMethod1(MyType2 myArg1);
        MyType3 myMethod2(MyType4 myArg2);
} // ';' is forbidden
```

- Reference type
- Only methods
- No implementation
- Inheritance

Classes

Declaration

```
class MyClass : MyParentClass MyInterface
{
        MyType myMethod(MyType myArg)
        {
        }
        MyType myMember1 = myValue;
        MyType myMember2;
} // ';' is forbidden
```

- Reference type
- Object identity
- Inheritance
- Interface implementation

Classes

Usage

```
MyClass myVar;
MyClass.myStaticMethod();
```

Classes

Protection attributes

- private
- package
- protected
- public
- export

Classes

Constructors

```
class MyClass
{
        this()
        {
        }
        this(MyType myArg)
        {
        }
} // ';' is forbidden
```

Classes

Destructor

```
class MyClass
{
        ~this()
        {
        }
} // ';' is forbidden
```

- Always virtual
- Called on delete or by the garbage collector

Classes

Allocators

```
class MyClass
{
        new(uint size, MyType myArg)
        {
        }
} // ';' is forbidden
new(myValue) MyClass();
```

- Always virtual
- Called on deletion or by the garbage collector

Classes

Access to parent class in C++

```
MyBaseClass();
MyBaseClass::myMethod();
```

Access to parent class in C#

```
base();
base.myMethod();
```

Access to parent class in Java or D

```
super();
super.myMethod();
```

Classes

Sealed class in C#

```
sealed class MyClass
{
}
```

Final class in D

```
final class MyClass
{
}
```

Classes

Method overriding

```
class MyClass : MyBaseClass
{
        override MyType myOverridenMethod()
        {
        }
} // ';' is forbidden
```

- More explicit
- Allow easier error detection

Classes

Block overriding

```
class MyClass : MyBaseClass
{
        override
        {
                MyType myOverridenMethod()
                {
                }
        }
} // ';' is forbidden
```

- Allow multiple overriding

Classes

Covariant method return type

```
class MyClass : MyBaseClass
{
        override typeof(this) myOverridenMethod()
        {
        }
} // ';' is forbidden
```

- Also possible in C++
- Not possible in C#

Classes

Abstract classes

```
abstract class MyClass
{
        MyType myMethod(MyType myArg)
        {
        }
        MyType myMember = myValue;
}
```

- Cannot be instanciated
- Provide base functionalities

Classes

Abstract methods

```
class MyClass
{
        abstract MyType myMethod(MyType myArg)
        {
        }
        MyType myMember = myValue;
}
```

- Make the enclosing class abstract
- Provide base functionalities

Classes

Multiple astract methods

```
class MyClass
{
        abstract
        {
                MyType myMethod(MyType myArg)
                {
                }
        }
}
```

- Make the enclosing class abstract
- Provide base functionalities

Classes

Nested classes

```
class MyClass1
{
        MyType myVar;
        class MyClass2
        {
        }
}
```

- Access to outer classes members

Classes

Functions classes

```
MyType myFunction
{
        MyType myVar;
        class MyClass
        {
        }
}
```

- Also possible in C++
- Not possible in C#

Classes

Access to outer classes

```
doSomethingWith(this.outer);
```

Classes

Anonymous classes

```
auto MyVar = new class MyParentClass MyInterface
{
        // Class content
};
```

Classes

Properties (setter)

```
class MyClass
{
        public:
                MyType myProperty(MyType value)
                {
                        return _myAttribute = value;
                }
        private:
                MyType _myAttribute;
}
```

- Transparent for the user

Classes

Properties (getter)

```
class MyClass
{
        public:
                MyType myProperty()
                {
                        return _myAttribute;
                }
        private:
                MyType _myAttribute;
}
```

- Transparent for the user

Classes

Properties usage

```
MyClass myVar;
myVar.myProperty = myValue;
doSomethingWith(myVar.myProperty);
// And soon ...
myVar.myProperty += myValue;
```

- Will be lvalue in next version

Types modifiers

Invariants

```
invariant int myInvariant = 42;
```

- Value never changes
- Can be stored in ROM
- Cannot be referenced

Types modifiers

Constants

```
int myVar = 42;
const (int*) myConstPtr;
myConstPtr = &myVar;
```

- Value cannot change through const variables
- Value can change elsewhere in the program
- Cannot be stored in ROM
- May be "const" referenced

Types modifiers

Finals

```
final int myFinal;
myFinal = 42;
```

- Value cannot be changed after the first assignment

Types modifiers

Multiple modifier in C++

```
const int* const* myVar;
int const* const* myVar;
```

- Value cannot be changed after the first assignment

Multiple modifier in D

```
const (int*)* myVar;
```

- Value cannot be changed after the first assignment

Types modifiers

Invariant methods

```
class MyClass : MyParentClass MyInterface
{
        invariant MyType myMethod(MyType myArg)
        {
        }
}
```

- Ensure that nothing referenced bt this can change

Simple template declaration

In C++

```
template<typename T>
T myFunction(T myArg);
```

In D

```
T myFunction(T)(T myArg)
{
}
```

Simple template declaration

Caracteristics

- Non ambiguous
- Not limited to integral values and types
- Flexible (see further)
- Need to be correct, even if not used
- Importable

Simple template declaration

Class or struct template

```
class MyClass(T)
{
}
struct MyStruct(T)
{
}
```

Simple template usage

In C++

```
myFunction<MyType>(myArg);
```

In D

```
myFunction!(MyType)(myArg);
```

Simple template usage

… Or even simpler, in C++

```
myFunction(myArg);
```

… Or in D

```
myFunction(myArg);
```

Multiple templates

In C++

```
template<typename T> T myFunction(T myArg);
template<typename T> T myVar;
```

```
template MyTemplate(T)
{
        T myFunction(T myArg)
        {
        }
        T myVar;
}
```

Multiple templates

Usage

```
MyTemplate!(MyType).myFunction(myArg);
MyTemplate!(MyType).myVar;
```

Complex template declaration

```
template MyTemplate(T: T*)
{
}
template MyTemplate(T: T[U], U)
{
}
```

Complex template declaration

Caracteristics

- Allow complex template patterns
- Compatible with inheritance

Complex template declaration

Default parameters

```
template MyTemplate(T: T[U], U = int)
{
}
template MyTemplate(T, U = T*)
{
}
```

Flexible templates

```
template MyTemplate(alias T)
{
}
```

- Allows complex template patterns
- Compatible with inheritance

Tuple parametrized templates

```
template myTemplate(T ...)
{
        void myFunction(T myTuple)
        {
        }
}
```

- Allow complex template patterns
- Allow factorization of tuple functions

Using templates and delegates for curryfication

See http://www.digitalmars.com/d/template.html

- Allow partial application of functions

Metaprogrammation

```
uint fact(int n)
{
        if (n == 1)
                return 1;
        else
                return n * fact(n - 1);
}
```

- No need to use templates in D
- Also no need to use inline functions

Exception creation

Usage

```
throw new Exception;
```

- Throw an exception

Exception handling

Usage

```
try
{
}
catch (MyExceptionType myException)
{
}
```

- Catch an exception

Finally clause

Usage

```
try
{
}
finally
{
}
```

- Executed wherever the flow is directed to

With

Usage

```
with (myObject)
{
        field1 = value1;
        field2 = value2;
}
```

Deprecated

Deprecated code

```
deprecated
{
        void deprecatedFunction()
        {
        }
}
```

- Generates warnings
- May generate errors

Volatile

Volatile statement

```
volatile
{
        a = *sharedMemory;
}
```

- Safe variables access
- Concurrent threads

Synchronized

Synchronized statement

```
synchronized
{
        if (action = Action.ADD)
                credit += value;
        else
                credit -= value;
}
```

- One execution at a time
- Concurrent threads

Scope

In scope variable

```
while (condition)
{
        scope MyType myVar;
        // Something useful
} // myVar is destroyed here
```

- Control over garbage collection
- The variable in destroyed out of scope

Special scopes

Success scope

```
doSomething()
{
        scope(success)
        {
        }
}
```

- Run at successful exit
- Called before the destructor

Special scopes

Failure scope

```
doSomething()
{
        scope(failure)
        {
        }
}
```

- Run at forced exit
- Called before the destructor

Special scopes

Exit scope

```
doSomething()
{
        scope(exit)
        {
        }
}
```

- Run after any exit
- Called after the destructor

ASM scopes

Usage

```
int myAdd(int x)
{
        asm
        {
                mov EAX, x[EBP];
        }
}
```

- Full control over the machine
- Fully integrated in D language

Assert

Usage

```
assert(predicate);
```

- Throw an exception

Input contracts

Usage

```
MyType myFunction(MyType myArg)
in
{
}
body
{
}
```

- Throw an exception
- Not in release version
- Evaluated before the function is called
- Not inherited

Output contracts

Usage

```
MyType myFunction(MyType myArg)
out (result)
{
}
body
{
}
```

- Throw an exception
- Not in release version
- Evaluated after the function has returned
- Inherited

Class invariants

Usage

```
class MyClass
{
        MyType myAttribute;
        invariant()
        {
                assert(somePredicate);
        }
}
```

- Predicates than never change

Unit tests

Usage

```
class MyClass
{
          MyType myAttribute;
          unittest
          {
                    assert(somePredicate);
          }
}
```

- Executed when compiling with -unittest switch

Static if

Usage

```
class MyClass(T)
{
        static if (is (T[]))
                MyType myMethod()
                {
                }
}
```

Static assert

Usage

```
static_assert(myPredicate);
```

Version

Usage

```
version (Win64)
{
}
version (X86_64)
{
}
```

Debug

Usage

```
debug (debugLevel)
{
}
debug (debugId)
{
}
```

Pragma

Usage

```
pragma (ExtensionName)
{
}
```

- Add features to the compiler
- Proprietary extensions

Example

```
pragma(msg, "Hello world!");
```

Fully functional inline asm

- Labelized asm
- Easy acces to member of aggregate
- Easy acces to stack variables (cdecl convetion)

Part III

Libraries

Phobos - Aim

Provide a standard library to handle common task.

- std.conv : equivalent for atoi(), atol()... with error handling (overflow, whitespaces...)
- std.ctype : ASCII character classification functions such as islower(dchar c), isspace(dchar c)...
- std.file : File handling. Common functions like read(const char[] name) and other features such as listdir(<...>), copy(<...>)...
- std.stdio : Common function such as writef(...), readln(_iobuf* fp = stdin)
- std.stream : Stream handling, with read(out byte x), readLine(), writeLine(const(char)[] s)...
- std.c.* : libc

Extra Features of Phobos

Some features that can help to make coffe.

- std.boxer : Put heap and value type in box type. (allow to put different objects in the same list of box)
- std.cpuid : Give information such as vendor, if mmx is present... on CPU at runtime (only for x86)
- std.uni : Like std.ctype but for Unicode character
- std.gc : Advanced garbage collector operation. Allow to run full collection, enable or disable GC, and a lot more.
- std.traits : Template to got information on type during compil time
- std.zip / std.zlib : Compression handling

http://digitalmars.com/d/phobos/phobos.html

Extra Features of Phobos

Some features that can help to make coffe.

- std.boxer : Put heap and value type in box type. (allow to put different objects in the same list of box)
- std.cpuid : Give information such as vendor, if mmx is present. . . on CPU at runtime (only for x86)
- std.uni : Like std.ctype but for Unicode character
- std.gc : Advanced garbage collector operation. Allow to run full collection, enable or disable GC, and a lot more.
- std.traits : Template to got information on type during compil time
- std.zip / std.zlib : Compression handling

http://digitalmars.com/d/phobos/phobos.html

Tango - Features

Provide another standard library with new functionalities.

Like in Phobos

- tango.core.Memory : Garbage collector operation
- tango.core.Traits : Type information during compil time
- tango.stdc.* : libc

Not in Phobos

- tango.core.Variant : Variant like in Boost (union with dynamic type checking)
- tango.net.* : Class handling a lot of protocol such as ftp with tango.net.ftp.FtpClient, http with tango.net.http.HttpClient...
- tango.io.digest.* : Digest algorithm like md5, sha1, crc32...

http://dsource.org/projects/tango/docs/current/

Tango - Features

Provide another standard library with new functionalities.

Like in Phobos

- tango.core.Memory : Garbage collector operation
- tango.core.Traits : Type information during compil time
- tango.stdc.* : libc

Not in Phobos

- tango.core.Variant : Variant like in Boost (union with dynamic type checking)
- tango.net.* : Class handling a lot of protocol such as ftp with tango.net.ftp.FtpClient, http with tango.net.http.HttpClient. . .
- tango.io.digest.* : Digest algorithm like md5, sha1, crc32. . .

http://dsource.org/projects/tango/docs/current/

Arclib is Based on

Based on

- OpenGL
- OpenAL
- SDL
- SDL_image
- FreeType

Arclib Features

Features

- Font Rendering
- 2D Physics Engine
- Easy access to input
- Easy GUI system
- DOM XML parser
- Loading png/jpg/tga/bmp/gif/pcx/lbm/xpm/pnm graphics files
- Font Rendering

http://dsource.org/projects/arclib
#arclib@irc.freenode.net

Arclib Features

Features

- Font Rendering
- 2D Physics Engine
- Easy access to input
- Easy GUI system
- DOM XML parser
- Loading png/jpg/tga/bmp/gif/pcx/lbm/xpm/pnm graphics files
- Font Rendering

http://dsource.org/projects/arclib
#arclib@irc.freenode.net

Must See

See : http://dsource.org/projects/

Conclusion

To remember

- A cool language with advanced features
- A cool community
- A cool presentation

To go further

- Ask us
- Read the language reference
- Join the forums
- Try it by yourself

Questions

Any question ?

Bibliography

Useful links

- http://www.digitalmars.com/d/ – The official D reference