Introduction to the D programming language

Marc Fuentes - SED

Why an another language?

- Fortran (90) is fast, has nice array features, but I/O and objects are not very powerful

Why an another language?

- Fortran (90) is fast, has nice array features, but I/O and objects are not very powerful
- C++ is fast and efficient, but its syntax is a pain

Why an another language?

- Fortran (90) is fast, has nice array features, but I/O and objects are not very powerful
- C++ is fast and efficient, but its syntax is a pain
- Python has a good syntax, but interpeters are slow

Why an another language?

- Fortran (90) is fast, has nice array features, but I/O and objects are not very powerful
- C++ is fast and efficient, but its syntax is a pain
- Python has a good syntax, but interpeters are slow
- D is a good trade-off !

Why an another language?

- Fortran (90) is fast, has nice array features, but I/O and objects are not very powerful
- C++ is fast and efficient, but its syntax is a pain
- Python has a good syntax, but interpeters are slow
- D is a good trade-off !

Why an another language?

- Fortran (90) is fast, has nice array features, but I/O and objects are not very powerful
- C++ is fast and efficient, but its syntax is a pain
- Python has a good syntax, but interpeters are slow
- D is a good trade-off !

General presentation

- at first glance: an improved C/C++ with less complexity

General presentation

- at first glance: an improved C/C++ with less complexity
- more paradigms:

General presentation

- at first glance: an improved C/C++ with less complexity
- more paradigms:
  - imperative: foreach, standard library, ranges, tuples

General presentation

- at first glance: an improved C/C++ with less complexity
- more paradigms:
  - imperative: foreach, standard library, ranges, tuples
  - functional: anonymous functions, closures (delegates), immutable types

General presentation

- at first glance: an improved C/C++ with less complexity
- more paradigms:
  - imperative: foreach, standard library, ranges, tuples
  - functional: anonymous functions, closures (delegates), immutable types
  - object-oriented: classes, interfaces

General presentation

- at first glance: an improved C/C++ with less complexity
- more paradigms:
  - imperative: foreach, standard library, ranges, tuples
  - functional: anonymous functions, closures (delegates), immutable types
  - object-oriented: classes, interfaces
  - generic: templates for functions, classes, structs, scopes

General presentation

- at first glance: an improved C/C++ with less complexity
- more paradigms:
  - imperative: foreach, standard library, ranges, tuples
  - functional: anonymous functions, closures (delegates), immutable types
  - object-oriented: classes, interfaces
  - generic: templates for functions, classes, structs, scopes
  - generative: mixins, CTFE

General presentation

- at first glance: an improved C/C++ with less complexity
- more paradigms:
  - imperative: foreach, standard library, ranges, tuples
  - functional: anonymous functions, closures (delegates), immutable types
  - object-oriented: classes, interfaces
  - generic: templates for functions, classes, structs, scopes
  - generative: mixins, CTFE
  - by contract: pre/post conditions , invariants

General presentation

- at first glance: an improved C/C++ with less complexity
- more paradigms:
  - imperative: foreach, standard library, ranges, tuples
  - functional: anonymous functions, closures (delegates), immutable types
  - object-oriented: classes, interfaces
  - generic: templates for functions, classes, structs, scopes
  - generative: mixins, CTFE
  - by contract: pre/post conditions , invariants
  - concurrency: synchronized statements, shared variables, atomic operations

General presentation

- at first glance: an improved C/C++ with less complexity
- more paradigms:
  - imperative: foreach, standard library, ranges, tuples
  - functional: anonymous functions, closures (delegates), immutable types
  - object-oriented: classes, interfaces
  - generic: templates for functions, classes, structs, scopes
  - generative: mixins, CTFE
  - by contract: pre/post conditions , invariants
  - concurrency: synchronized statements, shared variables, atomic operations
- initial aim is systems programming: native compiler, in-line assembler, etc..

D as an improved C/C++

- it has basic C-like statements: `for`, `if`, `else`, {}, `end`

D as an improved C/C++

- it has basic C-like statements: `for`, `if`, `else`, {}, `end`
- D has a static typing system: inferring simple types via `auto`

D as an improved C/C++

- it has basic C-like statements: `for`, `if`, `else`, {}, `end`
- D has a static typing system: inferring simple types via `auto`
- arrays, dynamic arrays and associative arrays are first-class entities ; arrays support slicing

D as an improved C/C++

- it has basic C-like statements: `for`, `if`, `else`, {}, `end`
- D has a static typing system: inferring simple types via `auto`
- arrays, dynamic arrays and associative arrays are first-class entities ; arrays support slicing
- easy string processing (concatenation `~` , operator `in`)

D as an improved C/C++

- it has basic C-like statements: `for`, `if`, `else`, {}, `end`
- D has a static typing system: inferring simple types via `auto`
- arrays, dynamic arrays and associative arrays are first-class entities ; arrays support slicing
- easy string processing (concatenation ~ , operator `in`)
- D has a garbage collector

D as an improved C/C++

- it has basic C-like statements: `for`, `if`, `else`, {}, `end`
- D has a static typing system: inferring simple types via `auto`
- arrays, dynamic arrays and associative arrays are first-class entities ; arrays support slicing
- easy string processing (concatenation ~ , operator `in`)
- D has a garbage collector
- in the standard library, most functions use *ranges*

D as an improve C/C++

```
#!/usr/bin/env rdmd
import std.stdio;
import std.conv;
import std.algorithm;

void main(string [] args) {
    if (args.length < 2) {
        writefln("%s n", args[0]);
    }
    else {
        int n = to!int(args[1]);
        auto carres = new int[n];
        foreach(int i, ref x ; carres) x = i;
        carres[1 .. $-1] *= carres[1 .. $-1];
        writeln(carres);
   }
}
```

Functional aspects I

- you can declare pure functions

```
pure int f(int x) { return x + 2; }  //OK
pure int g(int x) {
writeln("coucou");
return x+2;
} // Error: pure function 'g' cannot call impure function 'writeln'
```

Functional aspects I

- you can declare pure functions

```
pure int f(int x) { return x + 2; }  //OK
pure int g(int x) {
writeln("coucou");
return x+2;
} // Error: pure function 'g' cannot call impure function 'writeln'
```

- D supports anonymous functions

```
auto f = (int x) { return x+2; }
map!(x=>x+2)([1, 2, 3]);
```

Functional aspects I

- you can declare pure functions

```
pure int f(int x) { return x + 2; }  //OK
pure int g(int x) {
writeln("coucou");
return x+2;
} // Error: pure function 'g' cannot call impure function 'writeln'
```

- D supports anonymous functions

```
 auto f = (int x) { return x+2; }
 map!(x=>x+2)([1, 2, 3]);
```

- D supports closures *via* `delegate`

```
T3 delegate(T1) comp(T1,T2,T3)(T3 function(T2) f, T2 function(T1) g) {
return delegate (T1 x) { return f(g(x));};
}

void main() {
  auto f = ((int x)=>cast(double)(x)+1.);
  auto g = ((int x)=>x*x);
  writefln("fog(3) = %f", comp(f,g)(3));
}
```

Functional aspects I

- you can declare pure functions

```
pure int f(int x) { return x + 2; }  //OK
pure int g(int x) {
writeln("coucou");
return x+2;
} // Error: pure function 'g' cannot call impure function 'writeln'
```

- D supports anonymous functions

```
 auto f = (int x) { return x+2; }
 map!(x=>x+2)([1, 2, 3]);
```

- D supports closures *via* `delegate`

```
T3 delegate(T1) comp(T1,T2,T3)(T3 function(T2) f, T2 function(T1) g) {
return delegate (T1 x) { return f(g(x));};
}

void main() {
  auto f = ((int x)=>cast(double)(x)+1.);
  auto g = ((int x)=>x*x);
  writefln("fog(3) = %f", comp(f,g)(3));
}
```

- immutable type qualifier

```
immutable int z = 2;
z = 3 // Error!
```

User-defined types: structures

- User can define new types by `struct` keyword

User-defined types: structures

- User can define new types by `struct` keyword
- **value types** semantics

User-defined types: structures

- User can define new types by `struct` keyword
- **value types** semantics
- no inheritance, thus no dynamic polymorphism

User-defined types: structures

- User can define new types by `struct` keyword
- **value types** semantics
- no inheritance, thus no dynamic polymorphism
- user cannot define a default constructor

User-defined types: structures

- User can define new types by `struct` keyword
- **value types** semantics
- no inheritance, thus no dynamic polymorphism
- user cannot define a default constructor
- user can define `postblit` constructor

```
import std.stdio;

struct gauza {
    int[] val;
    this(int z) {
        val  = new int[1];
        val[0] = z;

    }
    this (this) {
         val = val.dup;
     }
 }

void main() {
    auto z = gauza(3);
    auto w = z;
    w.val[0] = 2;
    writeln(z.val[0]);  // writes 3
}
```

User-defined types: structures

- User can define new types by `struct` keyword
- **value types** semantics
- no inheritance, thus no dynamic polymorphism
- user cannot define a default constructor
- user can define `postblit` constructor

```
import std.stdio;

struct gauza {
    int[] val;
    this(int z) {
        val  = new int[1];
        val[0] = z;

    }
    this (this) {
         val = val.dup;
     }
 }

void main() {
    auto z = gauza(3);
    auto w = z;
    w.val[0] = 2;
    writeln(z.val[0]);  // writes 3
}
```

- `structs` have lifetime limited to the scope

User-defined types: classes

- Using `class`, user can also create new types

User-defined types: classes

- Using `class`, user can also create new types
- **ref types** semantics

```
import std.stdio;
class gauza {
    int[] val;
    this(int z) {
        val  = new int[1];
        val[0] = z;
    }
 }

void main() {
    auto z = new gauza(3);
    auto w = z;
    w.val[0] = 2;
    writeln(z.val[0]);  // writes 2
}
```

User-defined types: classes

- Using `class`, user can also create new types
- **ref types** semantics

```
import std.stdio;
class gauza {
    int[] val;
    this(int z) {
        val  = new int[1];
        val[0] = z;
    }
 }

void main() {
    auto z = new gauza(3);
    auto w = z;
    w.val[0] = 2;
    writeln(z.val[0]);  // writes 2
}
```

- all classes derive from `Object`

User-defined types: classes

- Using `class`, user can also create new types
- **ref types** semantics

```
import std.stdio;
class gauza {
    int[] val;
    this(int z) {
        val  = new int[1];
        val[0] = z;
    }
 }

void main() {
    auto z = new gauza(3);
    auto w = z;
    w.val[0] = 2;
    writeln(z.val[0]);  // writes 2
}
```

- all classes derive from `Object`
- only **single** inheritance of classes is allowed, but ...

User-defined types: classes

- Using `class`, user can also create new types
- **ref types** semantics

```
import std.stdio;
class gauza {
    int[] val;
    this(int z) {
        val  = new int[1];
        val[0] = z;
    }
 }

void main() {
    auto z = new gauza(3);
    auto w = z;
    w.val[0] = 2;
    writeln(z.val[0]);  // writes 2
}
```

- all classes derive from `Object`
- only **single** inheritance of classes is allowed, but ...
- multiple inheritance of **interfaces** is possible

User-defined types: classes

- Using `class`, user can also create new types
- **ref types** semantics

```
import std.stdio;
class gauza {
    int[] val;
    this(int z) {
        val  = new int[1];
        val[0] = z;
    }
 }

void main() {
    auto z = new gauza(3);
    auto w = z;
    w.val[0] = 2;
    writeln(z.val[0]);  // writes 2
}
```

- all classes derive from `Object`
- only **single** inheritance of classes is allowed, but ...
- multiple inheritance of **interfaces** is possible
- objects have infinite lifetime

Generic programming

- D supports parametrized functions

```
T[] amap(alias fun, T)(T[] a) {
    T[] b;
    b.length =  a.length;
    foreach( int i, z ; a)
          b[i] = fun(z);
    return b;
}

void main() { auto z = amap!(x=>x*x)([1, 2, 3]); }
```

Generic programming

- D supports parametrized functions

```
T[] amap(alias fun, T)(T[] a) {
   T[] b;
   b.length =  a.length;
   foreach( int i, z ; a)
         b[i] = fun(z);
   return b;
}

void main() { auto z = amap!(x=>x*x)([1, 2, 3]); }
```

- parametrized structs

```
struct zutabe(T) {
    T[] datuak;
    this(uint neurria) { datuak = new T[neurria]; }
    this(this) { datuak = datuak.dup; } // postblit constructor
    void opIndexAssign(T val, uint i) { datuak[i] = val; }
    T opIndex(uint i) { return datuak[i]; }
}
void main() {
     zutabe!double nire_zutabe;
     foreach(i; 0 .. 3) {
             nire_zutabe[i] = cast(double)(i);
     }
}
```

Generic programming

- D supports parametrized functions

```
T[] amap(alias fun, T)(T[] a) {
   T[] b;
   b.length =  a.length;
   foreach( int i, z ; a)
         b[i] = fun(z);
   return b;
}

void main() { auto z = amap!(x=>x*x)([1, 2, 3]); }
```

- parametrized structs

```
struct zutabe(T) {
    T[] datuak;
    this(uint neurria) { datuak = new T[neurria]; }
    this(this) { datuak = datuak.dup; } // postblit constructor
    void opIndexAssign(T val, uint i) { datuak[i] = val; }
    T opIndex(uint i) { return datuak[i]; }
}
void main() {
     zutabe!double nire_zutabe;
     foreach(i; 0 .. 3) {
             nire_zutabe[i] = cast(double)(i);
     }
}
```

- parametrized classes

Generic programming

- D supports parametrized functions

```
T[] amap(alias fun, T)(T[] a) {
   T[] b;
   b.length =  a.length;
   foreach( int i, z ; a)
         b[i] = fun(z);
   return b;
}

void main() { auto z = amap!(x=>x*x)([1, 2, 3]); }
```

- parametrized structs

```
struct zutabe(T) {
    T[] datuak;
    this(uint neurria) { datuak = new T[neurria]; }
    this(this) { datuak = datuak.dup; } // postblit constructor
    void opIndexAssign(T val, uint i) { datuak[i] = val; }
    T opIndex(uint i) { return datuak[i]; }
}
void main() {
     zutabe!double nire_zutabe;
     foreach(i; 0 .. 3) {
             nire_zutabe[i] = cast(double)(i);
     }
}
```

- parametrized classes
- parametrized interfaces

Generic programming

- D supports parametrized functions

```
T[] amap(alias fun, T)(T[] a) {
   T[] b;
   b.length =  a.length;
   foreach( int i, z ; a)
         b[i] = fun(z);
   return b;
}

void main() { auto z = amap!(x=>x*x)([1, 2, 3]); }
```

- parametrized structs

```
struct zutabe(T) {
    T[] datuak;
    this(uint neurria) { datuak = new T[neurria]; }
    this(this) { datuak = datuak.dup; } // postblit constructor
    void opIndexAssign(T val, uint i) { datuak[i] = val; }
    T opIndex(uint i) { return datuak[i]; }
}
void main() {
     zutabe!double nire_zutabe;
     foreach(i; 0 .. 3) {
             nire_zutabe[i] = cast(double)(i);
     }
}
```

- parametrized classes
- parametrized interfaces
- parametrized scopes

Compile Time Features I

- lot of things could be do at compile times with `static`:

Compile Time Features I

- lot of things could be do at compile times with `static`:
  - CTFE : can evaluate result of a function at compile time

```
 int fact(int n) {
     if (n == 0) return 1;
     return n * fact(n - 1);
}

void main() {
     immutable int n = 8;
     static int z = fact(n); //computed at compile time
     writeln(z);
}
```

Compile Time Features I

- lot of things could be do at compile times with `static`:
  - CTFE : can evaluate result of a function at compile time

```
 int fact(int n) {
     if (n == 0) return 1;
     return n * fact(n - 1);
}

void main() {
     immutable int n = 8;
     static int z = fact(n); //computed at compile time
     writeln(z);
}
```

  - you can use also `static assert`

```
static assert(ma_fonction_a_tester()); // compile-time test
```

Compile Time Features I

- lot of things could be do at compile times with `static`:
  - CTFE : can evaluate result of a function at compile time

```
 int fact(int n) {
     if (n == 0) return 1;
     return n * fact(n - 1);
}

void main() {
     immutable int n = 8;
     static int z = fact(n); //computed at compile time
     writeln(z);
}
```

  - you can use also `static assert`

```
static assert(ma_fonction_a_tester()); // compile-time test
```

  - for branching use `static if`

```
static if ([2].ptr.sizeof == 8)
    enum bool_64 = true;
else
    enum  bool_64=false;
```

Compile Time Features I

- lot of things could be do at compile times with `static`:
  - CTFE : can evaluate result of a function at compile time

```
 int fact(int n) {
     if (n == 0) return 1;
     return n * fact(n - 1);
}

void main() {
     immutable int n = 8;
     static int z = fact(n); //computed at compile time
     writeln(z);
}
```

  - you can use also `static assert`

```
static assert(ma_fonction_a_tester()); // compile-time test
```

  - for branching use `static if`

```
static if ([2].ptr.sizeof == 8)
    enum bool_64 = true;
else
    enum  bool_64=false;
```

- `mixin` allows us to transform strings into code

```
mixin("auto z = 2");
writeln(z);
```

Compile Time Features II

Compile Time Features II

```
import std.conv;
int fact(int n) {
    if (n == 0)
       return 1;
    return n * fact(n - 1);
}

string FactTaulaEgin(string izena, uint max = 100) {
  string emaitza = "immutable int["~to!string(max)~"] "
         ~izena~" = [";
  foreach(i ; 0 .. max) {
     emaitza ~= to!string(fact(i)) ~ ", ";
  }
  return emaitza ~ "];";
}
void main() {
  mixin(FactTaulaEgin("la_mia_tavola", 10));
  assert(la_mia_tavola[8] == 40320);
}
```

Compile Time Features II

```
import std.conv;
int fact(int n) {
    if (n == 0)
       return 1;
    return n * fact(n - 1);
}

string FactTaulaEgin(string izena, uint max = 100) {
  string emaitza = "immutable int["~to!string(max)~"] "
         ~izena~" = [";
  foreach(i ; 0 .. max) {
     emaitza ~= to!string(fact(i)) ~ ", ";
  }
  return emaitza ~ "];";
}
void main() {
  mixin(FactTaulaEgin("la_mia_tavola", 10));
  assert(la_mia_tavola[8] == 40320);
}
```

- mixin templates: injecting code with surrounding scope

```
mixin template  accessX() {
  int x;
  @property int getX() { return x; }
}

struct A { mixin accessX; }

void main() {
    auto z = A(2);
    writeln(z.getX);
}
```

Concurrency I

- message passing functions between threads

```
import std.stdio;
import std.variant;
import std.concurrency;

void un_thread()
{
    receive(
        (int i) { writeln("Received an int."); },
        (float f) { writeln("Received a float."); },
        (Variant v) { writeln("Received some other type."); }
    );
}

void main()
{
     auto tid = spawn(&un_thread);
     send(tid, 42);
}
```

Concurrency I

- message passing functions between threads

```
import std.stdio;
import std.variant;
import std.concurrency;

void un_thread()
{
    receive(
        (int i) { writeln("Received an int."); },
        (float f) { writeln("Received a float."); },
        (Variant v) { writeln("Received some other type."); }
    );
}

void main()
{
     auto tid = spawn(&un_thread);
     send(tid, 42);
}
```

- by default, data is not shared between threads: to enable, use `immutable` data or `shared` keyword

Concurrency I

- message passing functions between threads

```
import std.stdio;
import std.variant;
import std.concurrency;

void un_thread()
{
    receive(
        (int i) { writeln("Received an int."); },
        (float f) { writeln("Received a float."); },
        (Variant v) { writeln("Received some other type."); }
    );
}

void main()
{
     auto tid = spawn(&un_thread);
     send(tid, 42);
}
```

- by default, data is not shared between threads: to enable, use `immutable` data or `shared` keyword
- `synchronized` statement implements a mutex on a shared variable

```
synchronized (zut) {
       zut++;
}
```

Concurrency II

- `synchronized` statements are also available but a class (not structure) level.

Concurrency II

- `synchronized` statements are also available but a class (not structure) level.
- user can use atomic statements to modify shared variables

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

shared int accu = 0;

void th_add(int i) {
      atomicOp!"+="(accu, i);
}

void main() {
   foreach (i ; 0 .. 1000) {
     spawn(&th_add, i);
   }
   thread_joinAll();
   writeln(accu);
}
```

Concurrency II

- `synchronized` statements are also available but a class (not structure) level.
- user can use atomic statements to modify shared variables

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

shared int accu = 0;

void th_add(int i) {
      atomicOp!"+="(accu, i);
}

void main() {
   foreach (i ; 0 .. 1000) {
     spawn(&th_add, i);
   }
   thread_joinAll();
   writeln(accu);
}
```

- using `std.parallelism` there are high-level functions to do parallel jobs

```
auto sumk2(int n) {
  auto numbers = iota(1.,to!double(n+1));
  auto carres = map!"1/(a*a)"(numbers);
  return taskPool.reduce!"a+b"(carres);// instead of return reduce!"a+b"(carres);
}
```

Compilers

- DMD: compiler developed by W. Bright, creator of the language. Runs on GNU/Linux, OS X and Windows for x86 and x86_64 architectures. it is possible to use rdmd in the shebang as "JIT" compiler

Compilers

- DMD: compiler developed by W. Bright, creator of the language. Runs on GNU/Linux, OS X and Windows for x86 and x86_64 architectures. it is possible to use rdmd in the shebang as "JIT" compiler
- GDC: GNU D Compiler, developed par I. Buclaw. Use dmd as front-end, and GNU tools for backend. Runs on same OSes.

Compilers

- DMD: compiler developed by W. Bright, creator of the language. Runs on GNU/Linux, OS X and Windows for x86 and x86_64 architectures. it is possible to use rdmd in the shebang as "JIT" compiler
- GDC: GNU D Compiler, developed par I. Buclaw. Use dmd as front-end, and GNU tools for backend. Runs on same OSes.
- LDC: LLVM-Based D compiler uses LLVM tools as backend. Runs on GNU Linux, OSX and Windows. Only x86_64 and x86 architectures supported.

References

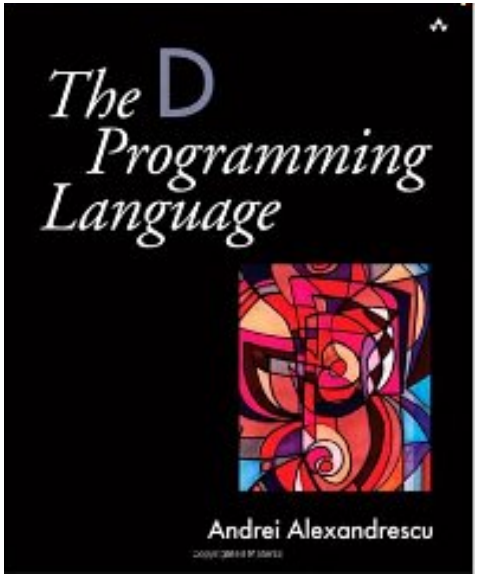
- D programming language page: `http://dlang.org`

References

- D programming language page: `http://dlang.org`
- D Programming Language Tutorial by Ali Çehreli: `http://ddili.org/ders/d.en`

References

- D programming language page: `http://dlang.org`
- D Programming Language Tutorial by Ali Çehreli: `http://ddili.org/ders/d.en`
- the book *The D programming language* by Andrei Alexandrescu



Conclusion

- For my point of view: D ..

Conclusion

- For my point of view: D ..
  - is a beautiful Swiss army knife to code:-)

Conclusion

- For my point of view: D ..
  - is a beautiful Swiss army knife to code:-)
  - is a good evolution of C++

Conclusion

- For my point of view: D ..
  - is a beautiful Swiss army knife to code:-)
  - is a good evolution of C++
  - is fast

Conclusion

- For my point of view: D ..
  - is a beautiful Swiss army knife to code:-)
  - is a good evolution of C++
  - is fast
  - has a powerful and expressive syntax

Conclusion

- For my point of view: D ..
  - is a beautiful Swiss army knife to code:-)
  - is a good evolution of C++
  - is fast
  - has a powerful and expressive syntax
- cons:

Conclusion

- For my point of view: D ..
  - is a beautiful Swiss army knife to code:-)
  - is a good evolution of C++
  - is fast
  - has a powerful and expressive syntax
- cons:
  - it is not widespread

Conclusion

- For my point of view: D ..
  - is a beautiful Swiss army knife to code:-)
  - is a good evolution of C++
  - is fast
  - has a powerful and expressive syntax
- cons:
  - it is not widespread
  - not "source" compatible with legacy code: although foreign function interface is possible