Competitive Advantage with D

Ali Çehreli

C++Now 2017 • Aspen, Colorado

Copyleft

Ali's email address:

acehreli@yahoo.com

Advantage?

- How to present D to C++ experts?
    - Simpler?
    - Quicker?
    - More powerful?
    - Pragmatic?
    - etc.

Advantage?

- How to present D to C++ experts?
    - Simpler?
    - Quicker?
    - More powerful?
    - Pragmatic?
    - etc.

- "Time is money... time is all life... time is all we really have. I don't have time to waste [...]
  – *Manu Evans*

If you remember just one thing...

(à la Dan Saks's CppCon 2016 presentation "extern c: Talking to C Programmers about C++")

If you remember just one thing...

(à la Dan Saks's CppCon 2016 presentation "extern c: Talking to C Programmers about C++")

The talk where Dan repeats

"If you're arguing, you're losing." – Mike Thomas

If you remember just one thing...

(à la Dan Saks's CppCon 2016 presentation "extern c: Talking to C Programmers about C++")

The talk where Dan repeats

"If you're arguing, you're losing." – Mike Thomas

```
static if
```

Ali the C++ programmer

- Former "in-house C++ expert" at various companies;
  C++ learner between 1996 and 2007
    - Follower of *good* C++ books
    - Former participant of **`comp.lang.c++.moderated`**
    - Member of the ACCU since 2000
    - Co-organizer of the Silicon Valley ACCU meetups

Ali the C++ programmer

- Former "in-house C++ expert" at various companies;
  C++ learner between 1996 and 2007
    - Follower of *good* C++ books
    - Former participant of **`comp.lang.c++.moderated`**
    - Member of the ACCU since 2000
    - Co-organizer of the Silicon Valley ACCU meetups

- Earned bragging rights
    - Question to an ACCU meetup speaker:
      – "Why C++?"
      – "Because it's hard."

Ali the C++ programmer

- Former "in-house C++ expert" at various companies;
  C++ learner between 1996 and 2007
    - Follower of *good* C++ books
    - Former participant of **`comp.lang.c++.moderated`**
    - Member of the ACCU since 2000
    - Co-organizer of the Silicon Valley ACCU meetups
- Earned bragging rights
    - Question to an ACCU meetup speaker:
      – "Why C++?"
      – "Because it's hard."
- Not an expert beyond C++03 since 2007

Ali the C++ programmer

- Former "in-house C++ expert" at various companies;
  C++ learner between 1996 and 2007
    - Follower of *good* C++ books
    - Former participant of **`comp.lang.c++.moderated`**
    - Member of the ACCU since 2000
    - Co-organizer of the Silicon Valley ACCU meetups
- Earned bragging rights
    - Question to an ACCU meetup speaker:
      – "Why C++?"
      – "Because it's hard."
- Not an expert beyond C++03 since 2007
- Not really programmed in C++ since 2015

Ali the C++ programmer

- Former "in-house C++ expert" at various companies;
  C++ learner between 1996 and 2007
    - Follower of *good* C++ books
    - Former participant of **`comp.lang.c++.moderated`**
    - Member of the ACCU since 2000
    - Co-organizer of the Silicon Valley ACCU meetups
- Earned bragging rights
    - Question to an ACCU meetup speaker:
      – "Why C++?"
      – "Because it's hard."
- Not an expert beyond C++03 since 2007
- Not really programmed in C++ since 2015
- Grateful to be among such a select group of people

Ali the D programmer

- Read "The Case for D" article by Andrei Alexandrescu in 2009
- Agreed immediately presumably because I was in a *correct frame*
  cf. Dan Saks's presentation

Ali the D programmer

- Read "The Case for D" article by Andrei Alexandrescu in 2009
- Agreed immediately presumably because I was in a *correct frame*
  cf. Dan Saks's presentation

- Translated the article to Turkish as "Neden D"
- Started writing an HTML D tutorial in Turkish

Ali the D programmer

- Read "The Case for D" article by Andrei Alexandrescu in 2009
- Agreed immediately presumably because I was in a *correct frame*
  cf. Dan Saks's presentation

- Translated the article to Turkish as "Neden D"
- Started writing an HTML D tutorial in Turkish

- Translated the tutorial to English (gentle persuasion by Andrei and Mengü)
- Self-published "Programming in D" in 2015 (freely available online and mostly up-to-date)

Ali the D programmer

- Read "The Case for D" article by Andrei Alexandrescu in 2009
- Agreed immediately presumably because I was in a *correct frame*
  cf. Dan Saks's presentation

- Translated the article to Turkish as "Neden D"
- Started writing an HTML D tutorial in Turkish

- Translated the tutorial to English (gentle persuasion by Andrei and Mengü)
- Self-published "Programming in D" in 2015 (freely available online and mostly up-to-date)

- A founding member and the secretary of The D Language Foundation, a 501(c)(3) non-profit public charity
- Co-organizer of the Silicon Valley DLang meetups

Ali the D programmer

- Read "The Case for D" article by Andrei Alexandrescu in 2009
- Agreed immediately presumably because I was in a *correct frame*
  cf. Dan Saks's presentation

- Translated the article to Turkish as "Neden D"
- Started writing an HTML D tutorial in Turkish

- Translated the tutorial to English (gentle persuasion by Andrei and Mengü)
- Self-published "Programming in D" in 2015 (freely available online and mostly up-to-date)

- A founding member and the secretary of The D Language Foundation, a 501(c)(3) non-profit public charity
- Co-organizer of the Silicon Valley DLang meetups

"Mere mortal"...

You

What's your experience with D?

a) Used it **professionally**

You

What's your experience with D?

a) Used it **professionally**

b) Used it **personally** (more than "hello world")

You

What's your experience with D?

a) Used it **professionally**

b) Used it **personally** (more than "hello world")

c) Wrote **hello world**

You

What's your experience with D?

a) Used it **professionally**

b) Used it **personally** (more than "hello world")

c) Wrote **hello world**

d) Heard of it but **never tried** it

You

What's your experience with D?

a) Used it **professionally**

b) Used it **personally** (more than "hello world")

c) Wrote **hello world**

d) Heard of it but **never tried** it

e) Not heard of it before the conference

You

What's your experience with D?

a) Used it **professionally**

b) Used it **personally** (more than "hello world")

c) Wrote **hello world**

d) Heard of it but **never tried** it

e) Not heard of it before the conference

f) Still has **no idea** what D is

Contents

- History and community
- Success stories and case studies
- General introduction
- Fundamental differences from C++
- Various useful features
- Software engineering support

Walter Bright

- Former mechanical engineer at Boeing
- Devoured the source code of Tiny Pascal in BYTE magazine
- "Who the hell do you think you are thinking you can write a C compiler?". Challenge accepted!
- Author of
    - Empire, a turn-based war game (1971)
    - Zortech, the first C++ compiler (late '80s)
    - Digital Mars compilers for C, C++, and D (currently)
- Creator of D1 in 2001 and D in 2007

Walter Bright

- Former mechanical engineer at Boeing
- Devoured the source code of Tiny Pascal in BYTE magazine
- "Who the hell do you think you are thinking you can write a C compiler?". Challenge accepted!
- Author of
    - Empire, a turn-based war game (1971)
    - Zortech, the first C++ compiler (late '80s)
    - Digital Mars compilers for C, C++, and D (currently)
- Creator of D1 in 2001 and D in 2007

- Previously said "[Arrays are] C's biggest mistake."

Walter Bright

- Former mechanical engineer at Boeing
- Devoured the source code of Tiny Pascal in BYTE magazine
- "Who the hell do you think you are thinking you can write a C compiler?". Challenge accepted!
- Author of
    - Empire, a turn-based war game (1971)
    - Zortech, the first C++ compiler (late '80s)
    - Digital Mars compilers for C, C++, and D (currently)
- Creator of D1 in 2001 and D in 2007

- Previously said "[Arrays are] C's biggest mistake."
- Recently said at DConf 2017 "I believe memory safety will kill C."

Andrei Alexandrescu

- Prominent figure in C++ and D communities
- Author of "Modern C++ Design" and co-author of "C++ Coding Standards"
  (Walter designed D's templates after reading "Modern C++ Design".)
- Author of "The D Programming Language"
- D language architect since 2007

Andrei Alexandrescu

- Prominent figure in C++ and D communities
- Author of "Modern C++ Design" and co-author of "C++ Coding Standards"
  (Walter designed D's templates after reading "Modern C++ Design".)
- Author of "The D Programming Language"
- D language architect since 2007

- Takes D code to C++ conferences
  ("gun to a knife fight"?)

The D Community

A wonderful group of volunteers as well as some paid individuals:

- Flourishing newsgroups (avaliable on a blazing fast forum interface: http://forum.dlang.org)
- Open source contributors: https://github.com/dlang
- **`irc://irc.freenode.net/d`**
- Annual conference (DConf 2017 held on May 4-7 in Berlin): http://dconf.org
- MSc students of University Politehnica of Bucharest, Romania
  (See DConf 2017 for their presentations)
- etc.

DConf

- Sixth was held in May 2017 in Berlin
- New addition this year: An extra day of hackathon
- Nobody speaks in standardese
- Many presentations list problems they face but end with "we wouldn't trade D for any other language"
- Lots of *D magic* based especially on
    - compile-time function execution (CTFE)
    - user defined attributes (UDA)
    - design by introspection

Success stories and case studies

Sociomantic

"[...] about smarter, easier, more effective display advertising [...]"

- "The Corporation" behind D
  - Sponsored DConf 2016 and 2017
  - Donates resources for D (DConf, branding, marketing materials, etc.)
  - Ports and open-sources code from D1 to D (most notable work-in-progress: their multi-threaded garbage collector)
- "A company based entirely on D"
- "Bootstrapped & organically grown; Profitable since founding"
- Shortlisted for Six Performance Marketing Awards
- UK-based Tesco's Dunnhumby acquired Sociomantic in 2014, for an undisclosed amount (Rumors: between $175-$200 million.)

- See DConf presentations by Don Clugston and many other Sociomantic people

Weka.IO

Disclaimer: I have personal interest in Weka's success.

"[...] software-defined file system that delivers flash performance at cloud scale [...]"

- Israeli startup company
- Raised $32 million
- Main technology based entirely on D
- Sprouted a vibrant community in Israel; 1-2 meetups per month
- Open-sourcing their libraries

- See DConf presentations by CTO Liran Zvibel

Remedy Games

"[...] independently developed cinematic story-driven action games since 1995."

- "The game industry is looking for a C++ alternative"
- Used D in development of Alan Wake and Quantum Break
- Quantum Break currently ships with D code
- Open-sourcing their libraries

- See DConf presentations by Ethan Watson and Manu Evans

vibe.d

Free and open-source asynchronous I/O framework: http://vibed.org

```
// Simple HTTP server
import vibe.d;

shared static this() {
    auto settings = new HTTPServerSettings;
    settings.port = 8080;

    listenHTTP(settings, &handleRequest);
}

void handleRequest(HTTPServerRequest req,
                   HTTPServerResponse res) {
    if (req.path == "/")
        res.writeBody("Hello, World!", "text/plain");
}
```

vibe.d

Free and open-source asynchronous I/O framework: http://vibed.org

```
// Simple HTTP server
import vibe.d;

shared static this() {
    auto settings = new HTTPServerSettings;
    settings.port = 8080;

    listenHTTP(settings, &handleRequest);
}

void handleRequest(HTTPServerRequest req,
                   HTTPServerResponse res) {
    if (req.path == "/")
        res.writeBody("Hello, World!", "text/plain");
}
```

```
// An echo server
import vibe.d;

shared static this() {
    listenTCP(7, (conn) { conn.write(conn); });
}
```

Pegged

A Parsing Expression Grammar (PEG) module: https://github.com/PhilippeSigaud/Pegged

```
import pegged.grammar;

// Note: This is a D string, not D syntax
mixin(grammar(`
Arithmetic:
    Term     < Factor (Add / Sub)*
    Add      < "+" Factor
    Sub      < "-" Factor
    Factor   < Primary (Mul / Div)*
    Mul      < "*" Primary
    Div      < "/" Primary
    Primary  < Parens / Neg / Pos / Number / Variable
    Parens   < "(" Term ")"
    Neg      < "-" Primary
    Pos      < "+" Primary
    Number   < ~([0-9]+)

    Variable <- identifier
`));
```

Pegged

A Parsing Expression Grammar (PEG) module: https://github.com/PhilippeSigaud/Pegged

```
import pegged.grammar;

// Note: This is a D string, not D syntax
mixin(grammar(`
Arithmetic:
    Term     < Factor (Add / Sub)*
    Add      < "+" Factor
    Sub      < "-" Factor
    Factor   < Primary (Mul / Div)*
    Mul      < "*" Primary
    Div      < "/" Primary
    Primary  < Parens / Neg / Pos / Number / Variable
    Parens   < "(" Term ")"
    Neg      < "-" Primary
    Pos      < "+" Primary
    Number   < ~([0-9]+)

    Variable <- identifier
`));
```

```
enum parseTree1 = Arithmetic("1 + 2 - (3*x-5)*6");  // Compile-time
auto parseTree2 = Arithmetic(str);                   // Run-time
```

See Bastiaan Veelo's DConf 2017 talk "Extending Pegged to Parse Another Programming Language".

Example: ctRegex

The pattern is parsed at run time:

```
import std.regex;

    auto re =   regex (`^.*/([^/]+)/?$`);
```

Parsed at compile time:

```
    auto re = ctRegex!(`^.*/([^/]+)/?$`);
```

tsv-utils-dlang

I'm stealing slides from Simon Arneaud's DConf 2017 presentation.

https://github.com/eBay/tsv-utils-dlang

The "Keep Calm and Write Sensible Code" approach

- Tools for processing delimited text files (CSV, TSV, etc)
- Made by **Jon Degenhardt for data mining at eBay**
- Did not worry about avoiding features like GC
- Performance due to common sense like avoiding redundant copying and allocating

tsv-utils-dlang

I'm stealing slides from Simon Arneaud's DConf 2017 presentation.

https://github.com/eBay/tsv-utils-dlang

The "Keep Calm and Write Sensible Code" approach

- Tools for processing delimited text files (CSV, TSV, etc)
- Made by **Jon Degenhardt for data mining at eBay**
- Did not worry about avoiding features like GC
- Performance due to common sense like avoiding redundant copying and allocating

Naive approach

```
// sumByKey is an associative array (unordered map)
string key = keySlice.idup;
sumByKey[key] += value;
```

tsv-utils-dlang

I'm stealing slides from Simon Arneaud's DConf 2017 presentation.

https://github.com/eBay/tsv-utils-dlang

The "Keep Calm and Write Sensible Code" approach

- Tools for processing delimited text files (CSV, TSV, etc)
- Made by **Jon Degenhardt for data mining at eBay**
- Did not worry about avoiding features like GC
- Performance due to common sense like avoiding redundant copying and allocating

Naive approach

```
// sumByKey is an associative array (unordered map)
string key = keySlice.idup;
sumByKey[key] += value;
```

Smart approach (**`idup`** is called only when creating the map entry)

```
auto entryPtr = (keySlice in sumByKey);

if (entryPtr is null) sumByKey[keySlice.idup] = value;
else *entryPtr += value;
```

tsv-utils-dlang performance

Benchmark	Tool/Time	Tool/Time	Tool/Time	Tool/Time
Numeric row filter	*tsv-filter*	mawk	GNU awk	Toolkit 1
(4.8 GB, 7M lines)	4.34	11.71	22.02	53.11
Regex row filter	*tsv-filter*	GNU awk	mawk	Toolkit 1
(2.7 GB, 14M lines)	7.11	15.41	16.58	28.59
Column selection	*tsv-select*	mawk	GNU cut	Toolkit 1
(4.8 GB, 7M lines)	4.09	9.38	12.27	19.12
Join two files	*tsv-join*	Toolkit 1	Toolkit 2	Toolkit 3
(4.8 GB, 7M lines)	20.78	104.06	194.80	266.42
Summary statistics	*tsv-summarize*	Toolkit 1	Toolkit 2	Toolkit 3
(4.8 GB, 7M lines)	15.83	40.27	48.10	62.97
CSV-to-TSV	*csv2tsv*	csvtk	xsv	
(2.7 GB, 14M lines)	27.41	36.26	40.40	

https://github.com/eBay/tsv-utils-dlang/blob/master/docs/Performance.md#top-four-in-each-benchmark

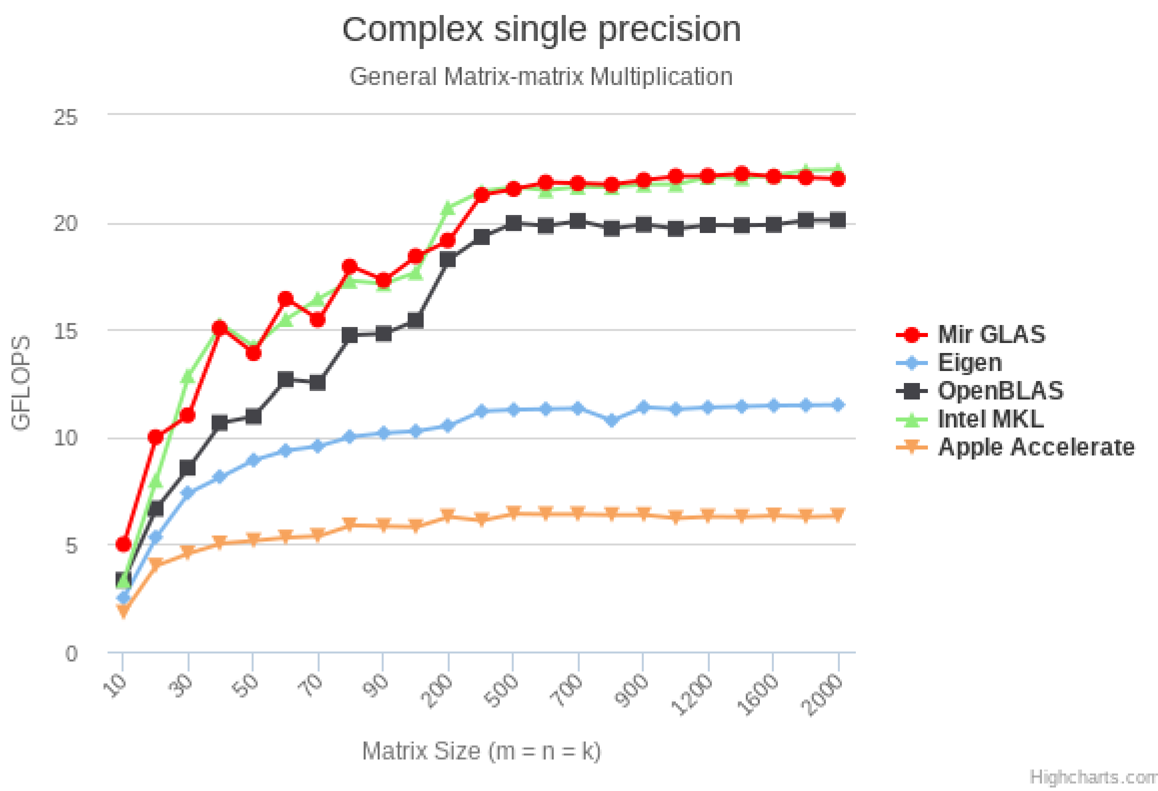
Mir numerical library

https://github.com/libmir/mir

The "D as a Better C" approach

- Collection of numerical libraries in D (think BLAS, NumPy) by **Ilya Yaroshenko**
- Uses `-betterC` flag and avoids D runtime features
- Mir GLAS can be linked to plain C code as BLAS implementation
- High performance through solid engineering and effective use of CPU features like SIMD

Mir performance



Auburn Sounds

https://www.auburnsounds.com/index.html

The `@nogc` approach

- Commercial audio plugins in D
- Mostly relies on `@nogc` for latency-sensitive code

Alternative: put audio handling in thread detached from GC (see `core.thread`)

Xanthe

https://gitlab.com/sarneaud/xanthe

https://theartofmachinery.com/2017/02/28/bare_metal_d.html

The horrible hacks approach

- Short vertical-scrolling shooter game demo that **boots on bare metal x86, by Simon Arneaud**
- Freestanding D
    - No D runtime
    - No C runtime
    - No OS

Other examples of industry use

- **Facebook** in developer tools

Other examples of industry use

- **Facebook** in developer tools
- **Cereris**, developing MVC web framework for embedded systems

Other examples of industry use

- **Facebook** in developer tools
- **Cereris**, developing MVC web framework for embedded systems
- **Emsi**, data driven modeling

Other examples of industry use

- **Facebook** in developer tools
- **Cereris**, developing MVC web framework for embedded systems
- **Emsi**, data driven modeling
- **Funkwerk**, passenger information systems

Other examples of industry use

- **Facebook** in developer tools
- **Cereris**, developing MVC web framework for embedded systems
- **Emsi**, data driven modeling
- **Funkwerk**, passenger information systems
- **???**, large-scale trading system within a hedge fund group (*See Andy Smith*'*s presentation at DConf 2015*)

Other examples of industry use

- **Facebook** in developer tools
- **Cereris**, developing MVC web framework for embedded systems
- **Emsi**, data driven modeling
- **Funkwerk**, passenger information systems
- **???**, large-scale trading system within a hedge fund group (*See Andy Smith*'*s presentation at DConf 2015*)
- Many others, including many startups

Academia use

High appreciation from teachers and students.

- Chuck Allison at Utah Valley University (hosted DConf 2015)

Academia use

High appreciation from teachers and students.

- Chuck Allison at Utah Valley University (hosted DConf 2015)
- Carl Sturtivant at University of Minnesota

Academia use

High appreciation from teachers and students.

- Chuck Allison at Utah Valley University (hosted DConf 2015)
- Carl Sturtivant at University of Minnesota
- Many MSc and PhD work around the world

General introduction

Resources

Main site: http://dlang.org

DUB, the D package registry: http://code.dlang.org

Ali's book: http://ddili.org

D and the odds

D and the odds

- Not taking anything away from the programmer

D and the odds

- Not taking anything away from the programmer
- No major corporation support

D and the odds

- Not taking anything away from the programmer
- No major corporation support
- No killer app

D and the odds

- Not taking anything away from the programmer
- No major corporation support
- No killer app
- No favorite paradigm

D and the odds

- Not taking anything away from the programmer
- No major corporation support
- No killer app
- No favorite paradigm
- No favorite kind of programmer
  academia and industry, young and old, fresh and experienced, genius and mere mortal, etc.

D and the odds

- Not taking anything away from the programmer
- No major corporation support
- No killer app
- No favorite paradigm
- No favorite kind of programmer
  academia and industry, young and old, fresh and experienced, genius and mere mortal, etc.
- No favorite problem domain

D and the odds

- Not taking anything away from the programmer
- No major corporation support
- No killer app
- No favorite paradigm
- No favorite kind of programmer
  academia and industry, young and old, fresh and experienced, genius and mere mortal, etc.
- No favorite problem domain
- No favorite memory management strategy
  e.g. famously accepts garbage collection

D and the odds

- Not taking anything away from the programmer
- No major corporation support
- No killer app
- No favorite paradigm
- No favorite kind of programmer
  academia and industry, young and old, fresh and experienced, genius and mere mortal, etc.
- No favorite problem domain
- No favorite memory management strategy
  e.g. famously accepts garbage collection
- No favorite kind of type
  value types and reference types
  Question to an ACCU meetup speaker:
  – "Have you looked at D?"
  – "A language with reference types? No, thanks."

D and the odds

- Not taking anything away from the programmer
- No major corporation support
- No killer app
- No favorite paradigm
- No favorite kind of programmer
  academia and industry, young and old, fresh and experienced, genius and mere mortal, etc.
- No favorite problem domain
- No favorite memory management strategy
  e.g. famously accepts garbage collection
- No favorite kind of type
  value types and reference types
  Question to an ACCU meetup speaker:
  – "Have you looked at D?"
  – "A language with reference types? No, thanks."
- No bragging rights

D and the odds

- Not taking anything away from the programmer
- No major corporation support
- No killer app
- No favorite paradigm
- No favorite kind of programmer
  academia and industry, young and old, fresh and experienced, genius and mere mortal, etc.
- No favorite problem domain
- No favorite memory management strategy
  e.g. famously accepts garbage collection
- No favorite kind of type
  value types and reference types
  Question to an ACCU meetup speaker:
  – "Have you looked at D?"
  – "A language with reference types? No, thanks."
- No bragging rights
- No facial hair :o)

General introduction

- Familiar features from C, C++, Java, C#, Python, and many others ("compilable Python")

General introduction

- Familiar features from C, C++, Java, C#, Python, and many others ("compilable Python")
- Strongly statically typed with type and attribute inference

General introduction

- Familiar features from C, C++, Java, C#, Python, and many others ("compilable Python")
- Strongly statically typed with type and attribute inference
- Powerful templates and other compile-time features

General introduction

- Familiar features from C, C++, Java, C#, Python, and many others ("compilable Python")
- Strongly statically typed with type and attribute inference
- Powerful templates and other compile-time features
- Multi-paradigm system language

General introduction

- Familiar features from C, C++, Java, C#, Python, and many others ("compilable Python")
- Strongly statically typed with type and attribute inference
- Powerful templates and other compile-time features
- Multi-paradigm system language
- Both low-level and high-level

General introduction

- Familiar features from C, C++, Java, C#, Python, and many others ("compilable Python")
- Strongly statically typed with type and attribute inference
- Powerful templates and other compile-time features
- Multi-paradigm system language
- Both low-level and high-level
- Compiles fast

General introduction

- Familiar features from C, C++, Java, C#, Python, and many others ("compilable Python")
- Strongly statically typed with type and attribute inference
- Powerful templates and other compile-time features
- Multi-paradigm system language
- Both low-level and high-level
- Compiles fast
- Executes fast

General introduction

- Familiar features from C, C++, Java, C#, Python, and many others ("compilable Python")
- Strongly statically typed with type and attribute inference
- Powerful templates and other compile-time features
- Multi-paradigm system language
- Both low-level and high-level
- Compiles fast
- Executes fast
- Readily links with C (and C++)

General introduction

- Familiar features from C, C++, Java, C#, Python, and many others ("compilable Python")
- Strongly statically typed with type and attribute inference
- Powerful templates and other compile-time features
- Multi-paradigm system language
- Both low-level and high-level
- Compiles fast
- Executes fast
- Readily links with C (and C++)
- Many resource management options: RAII, GC, RC, manual, etc.

General introduction

- Familiar features from C, C++, Java, C#, Python, and many others ("compilable Python")
- Strongly statically typed with type and attribute inference
- Powerful templates and other compile-time features
- Multi-paradigm system language
- Both low-level and high-level
- Compiles fast
- Executes fast
- Readily links with C (and C++)
- Many resource management options: RAII, GC, RC, manual, etc.
- Designed by community (D improvement proposals (DIP))

Compilers

All free and open source.

- `dmd`: Digital Mars compiler (the reference compiler)
- `ldc`: Digital Mars front-end, LLVM back-end (tracks `dmd` very closely)
- `gdc`: Digital Mars front-end, GCC back-end
- Many others work in progress, including compilers as libraries

Compilers

All free and open source.

- **`dmd`**: Digital Mars compiler (the reference compiler)
- **`ldc`**: Digital Mars front-end, LLVM back-end (tracks **`dmd`** very closely)
- **`gdc`**: Digital Mars front-end, GCC back-end
- Many others work in progress, including compilers as libraries

Scriptable as well:

```
#!/usr/bin/rdmd

// ... D code here ...
```

Compilers

All free and open source.

- **`dmd`**: Digital Mars compiler (the reference compiler)
- **`ldc`**: Digital Mars front-end, LLVM back-end (tracks **`dmd`** very closely)
- **`gdc`**: Digital Mars front-end, GCC back-end
- Many others work in progress, including compilers as libraries

Scriptable as well:

```
#!/usr/bin/rdmd

// ... D code here ...
```

Code on the command line?

```
rdmd --eval='foreach(line; File("Makefile").byLine)
                 if (line.length < 10)
                     writeln(line);'
```

Hardware support

- X86: dmd, ldc, and gdc
- ARM: ldc and gdc
- MIPS: ldc
- PPC: ldc
- DSPs, FPGAs GPUs: Requires elbow grease (see past DConf talks, e.g. DCompute and DHDL at DConf 2017)

WATs and warts

Like most other languages, D has issues.

WATs and warts

Like most other languages, D has issues.

I will not necessarily expose all of them. `>:)`

WATs and warts

Like most other languages, D has issues.

I will not necessarily expose all of them. `>:)`

Examples:

- Some defaults are like C and C++
- Same integer promotion rules as C
- Implicit type conversion from signed to unsigned (nasty!)
- etc.

Common objections to D

- "The *reference* compiler is not open source"
  - Not true as of April 2017, courtesy of Symantec

Common objections to D

- "The *reference* compiler is not open source"
  - Not true as of April 2017, courtesy of Symantec
- "Split community"
  - D used to be a one-man show
  - Development was inefficient (patches emailed to Walter by early community members)
  - As a reaction, the open source project Tango was created as an alternative (incompatible) runtime and library
  - Tango has since been ported to the official D runtime and Phobos is the official standard library

Common objections to D

- "The *reference* compiler is not open source"
    - Not true as of April 2017, courtesy of Symantec
- "Split community"
    - D used to be a one-man show
    - Development was inefficient (patches emailed to Walter by early community members)
    - As a reaction, the open source project Tango was created as an alternative (incompatible) runtime and library
    - Tango has since been ported to the official D runtime and Phobos is the official standard library
- "It has garbage collector"

Common objections to D

- "The *reference* compiler is not open source"
    - Not true as of April 2017, courtesy of Symantec
- "Split community"
    - D used to be a one-man show
    - Development was inefficient (patches emailed to Walter by early community members)
    - As a reaction, the open source project Tango was created as an alternative (incompatible) runtime and library
    - Tango has since been ported to the official D runtime and Phobos is the official standard library
- "It has garbage collector"
- "It has reference types"

Objection: Garbage collector

We need to talk...

Objection: Garbage collector

We need to talk...

- GC is a form of automatic memory management
- John McCarthy's invention, serving humanity since 1959
- Its Wikipedia entry does not have any controversies or objections
- Used by systems programmers everywhere for *prototyping* with auxiliary languages
- See Herb Sutter's "deferred and unordered destruction library for C++"
- GC can be the fastest option (destruction and deallocation takes time)

D's GC

- Does not run on a separate thread

D's GC

- Does not run on a separate thread
- Does not collect at random times

D's GC

- Does not run on a separate thread
- Does not collect at random times
- Can be disabled (`GC.disable()`)

D's GC

- Does not run on a separate thread
- Does not collect at random times
- Can be disabled (**`GC.disable()`**)
- Can collect on command (**`GC.collect()`**)

D's GC

- Does not run on a separate thread
- Does not collect at random times
- Can be disabled (`GC.disable()`)
- Can collect on command (`GC.collect()`)
- Can be banned (`@nogc` attribute)

D's GC

- Does not run on a separate thread
- Does not collect at random times
- Can be disabled (`GC.disable()`)
- Can collect on command (`GC.collect()`)
- Can be banned (`@nogc` attribute)
- Can tell line numbers where GC allocations occur (`-vgc` compiler switch)

D's GC

- Does not run on a separate thread
- Does not collect at random times
- Can be disabled (`GC.disable()`)
- Can collect on command (`GC.collect()`)
- Can be banned (`@nogc` attribute)
- Can tell line numbers where GC allocations occur (`-vgc` compiler switch)
- Can be profiled (`-profile=gc` compiler switch; `--DRT-gcopt=profile:1` as command line argument to the compiled program)

D's GC

- Does not run on a separate thread
- Does not collect at random times
- Can be disabled (**`GC.disable()`**)
- Can collect on command (**`GC.collect()`**)
- Can be banned (**`@nogc`** attribute)
- Can tell line numbers where GC allocations occur (**`-vgc`** compiler switch)
- Can be profiled (**`-profile=gc`** compiler switch; **`--DRT-gcopt=profile:1`** as command line argument to the compiled program)
- GC can live alongside other memory management techniques (e.g. **`std.experimental.allocator`**, your allocator, **`malloc`**)

D's GC

- Does not run on a separate thread
- Does not collect at random times
- Can be disabled (**`GC.disable()`**)
- Can collect on command (**`GC.collect()`**)
- Can be banned (**`@nogc`** attribute)
- Can tell line numbers where GC allocations occur (**`-vgc`** compiler switch)
- Can be profiled (**`-profile=gc`** compiler switch; **`--DRT-gcopt=profile:1`** as command line argument to the compiled program)
- GC can live alongside other memory management techniques (e.g. **`std.experimental.allocator`**, your allocator, **`malloc`**)

Not a great implementation:

- Not multi-threaded (Sociomantic may finish porting theirs)

D's GC

- Does not run on a separate thread
- Does not collect at random times
- Can be disabled (`GC.disable()`)
- Can collect on command (`GC.collect()`)
- Can be banned (`@nogc` attribute)
- Can tell line numbers where GC allocations occur (`-vgc` compiler switch)
- Can be profiled (`-profile=gc` compiler switch; `--DRT-gcopt=profile:1` as command line argument to the compiled program)
- GC can live alongside other memory management techniques (e.g. `std.experimental.allocator`, your allocator, `malloc`)

Not a great implementation:

- Not multi-threaded (Sociomantic may finish porting theirs)
- Not precise

Garbage collection can be good

```
import std.concurrency;
import std.range;
import core.thread;

void main() {                         // Main thread
    auto w = spawn(&worker);

    foreach (i; 0 .. 100) {
        immutable arr = iota(i).array;
        w.send(arr);
    }

    thread_joinAll();
}

void worker() {                       // Worker thread
    for (;;) {
        receive(
            (immutable(int[]) arr) {
                // ...
            });
    }
}
```

Objection: Reference types

(*In D, structs are value types and classes are reference types.*)

Objection: Reference types

(*In D, structs are value types and classes are reference types.*)

- Almost always it is obvious at design time. (All C++ code below.)

```
struct Point {                                    // ← Value type
    int x; int y;
};

class MyAction : public Action {                  // ← Reference type
    virtual void doIt();
};
```

Objection: Reference types

(*In D, structs are value types and classes are reference types.*)

- Almost always it is obvious at design time. (All C++ code below.)

```
struct Point {                                   // ← Value type
    int x; int y;
};

class MyAction : public Action {                 // ← Reference type
    virtual void doIt();
};
```

- A common use in Ali's pre-C++11 code:

```
class Foo {
    // ...
};
typedef boost::shared_ptr<Foo> FooPtr;           // ← Reference type
```

Objection: Reference types

(*In D, structs are value types and classes are reference types.*)

- Almost always it is obvious at design time. (All C++ code below.)

```
struct Point {                                  // ← Value type
    int x; int y;
};

class MyAction : public Action {                // ← Reference type
    virtual void doIt();
};
```

- A common use in Ali's pre-C++11 code:

```
class Foo {
    // ...
};
typedef boost::shared_ptr<Foo> FooPtr;          // ← Reference type
```

- Reference types make some concepts vanish:
    - Slicing
    - Copying
    - Assignment
    - Moving

Unwanted legacy is dropped

Although D has strong resemblance to C and C++, many legacy features are dropped:

Unwanted legacy is dropped

Although D has strong resemblance to C and C++, many legacy features are dropped:

- No preprocessor

Unwanted legacy is dropped

Although D has strong resemblance to C and C++, many legacy features are dropped:

- No preprocessor
- No zero terminated strings (pointer + length instead)

Unwanted legacy is dropped

Although D has strong resemblance to C and C++, many legacy features are dropped:

- No preprocessor
- No zero terminated strings (pointer + length instead)
- Less implicit conversions

Unwanted legacy is dropped

Although D has strong resemblance to C and C++, many legacy features are dropped:

- No preprocessor
- No zero terminated strings (pointer + length instead)
- Less implicit conversions
- No octal literals (see **`std.conv.octal`**)

```
int a = 042;    // ← compilation ERROR
```

Unwanted legacy is dropped

Although D has strong resemblance to C and C++, many legacy features are dropped:

- No preprocessor
- No zero terminated strings (pointer + length instead)
- Less implicit conversions
- No octal literals (see **`std.conv.octal`**)

```
    int a = 042;    // ← compilation ERROR
```

- Comma operator is crippled

Unwanted legacy is dropped

Although D has strong resemblance to C and C++, many legacy features are dropped:

- No preprocessor
- No zero terminated strings (pointer + length instead)
- Less implicit conversions
- No octal literals (see **`std.conv.octal`**)

```
    int a = 042;    // ← compilation ERROR
```

- Comma operator is crippled
- No support for truly legacy platforms (e.g. 16-bit platforms? 17 bit chars?)

Unwanted legacy is dropped

Although D has strong resemblance to C and C++, many legacy features are dropped:

- No preprocessor
- No zero terminated strings (pointer + length instead)
- Less implicit conversions
- No octal literals (see **`std.conv.octal`**)

```
    int a = 042;    // ← compilation ERROR
```

- Comma operator is crippled
- No support for truly legacy platforms (e.g. 16-bit platforms? 17 bit chars?)
- All types have known sizes (**`int`** is 32, **`long`** is 64, etc.) except **`real`**

Unwanted legacy is dropped

Although D has strong resemblance to C and C++, many legacy features are dropped:

- No preprocessor
- No zero terminated strings (pointer + length instead)
- Less implicit conversions
- No octal literals (see **`std.conv.octal`**)

```
    int a = 042;    // ← compilation ERROR
```

- Comma operator is crippled
- No support for truly legacy platforms (e.g. 16-bit platforms? 17 bit chars?)
- All types have known sizes (**`int`** is 32, **`long`** is 64, etc.) except **`real`**
- Many legacy compiler warnings are errors in D

Unwanted legacy is dropped

Although D has strong resemblance to C and C++, many legacy features are dropped:

- No preprocessor
- No zero terminated strings (pointer + length instead)
- Less implicit conversions
- No octal literals (see **`std.conv.octal`**)

```
    int a = 042;    // ← compilation ERROR
```

- Comma operator is crippled
- No support for truly legacy platforms (e.g. 16-bit platforms? 17 bit chars?)
- All types have known sizes (**`int`** is 32, **`long`** is 64, etc.) except **`real`**
- Many legacy compiler warnings are errors in D
- No shadowing declarations

Unwanted legacy is dropped

Although D has strong resemblance to C and C++, many legacy features are dropped:

- No preprocessor
- No zero terminated strings (pointer + length instead)
- Less implicit conversions
- No octal literals (see **`std.conv.octal`**)

```
    int a = 042;    // ← compilation ERROR
```

- Comma operator is crippled
- No support for truly legacy platforms (e.g. 16-bit platforms? 17 bit chars?)
- All types have known sizes (**`int`** is 32, **`long`** is 64, etc.) except **`real`**
- Many legacy compiler warnings are errors in D
- No shadowing declarations
- No lexical ordering of declarations (at module scope)

```
void main() { int i = foo();  // Compiles }
int foo() { return 42; }
```

Fundamental differences from C++

void main()

No problem...

```
void main() {
    // returns 0
}
```

```
void main() {
    throw new Exception("oops");
    // returns 1
}
```

Cleaner syntax

Designed to be simpler for programmers and tools. (Module system helps.)

Cleaner syntax

Designed to be simpler for programmers and tools. (Module system helps.)

```
struct S { /* ... */ }    // Less semicolons, etc.
```

Cleaner syntax

Designed to be simpler for programmers and tools. (Module system helps.)

```
struct S { /* ... */ }    // Less semicolons, etc.
```

```
i = to!int(str);      // Less noisy templates
i = str.to!int();     // Reads more naturally (UFCS)
i = str.to!int;       // No need for empty parentheses
```

Cleaner syntax

Designed to be simpler for programmers and tools. (Module system helps.)

```
struct S { /* ... */ }    // Less semicolons, etc.
```

```
i = to!int(str);      // Less noisy templates
i = str.to!int();     // Reads more naturally (UFCS)
i = str.to!int;       // No need for empty parentheses
```

```
foo(a => a * 2);      // Lambda

int[42]   a;          // 42 ints
int[3][4] b;          // Consistent: 4 int[3]s

ptr.member;           // No need for ->
```

Cleaner syntax

Designed to be simpler for programmers and tools. (Module system helps.)

```
struct S { /* ... */ }    // Less semicolons, etc.
```

```
i = to!int(str);     // Less noisy templates
i = str.to!int();    // Reads more naturally (UFCS)
i = str.to!int;      // No need for empty parentheses
```

```
foo(a => a * 2);     // Lambda

int[42]   a;         // 42 ints
int[3][4] b;         // Consistent: 4 int[3]s

ptr.member;          // No need for ->
```

```
int function(double) foo;    // Returning int, taking double
int function(double) function(string) bar;  // Consistent
```

Many more...

.init

Every type has a compile-time known initial value and every variable is initialized by default.

```
struct S {
    int i;
    double d;
    char c;      // UTF-8 code unit (nothing else!)
}

pragma(msg, S.init);
```

```
S(0, nan, '\xff')
```

.init

Every type has a compile-time known initial value and every variable is initialized by default.

```
struct S {
    int i;
    double d;
    char c;      // UTF-8 code unit (nothing else!)
}

pragma(msg, S.init);
```

```
S(0, nan, '\xff')
```

```
struct S {
    int i    = 1;
    double d = 2.5;
    char c   = 'a';

/* NOTE: Constructors are for runtime
 *       .init is known at compile time.
 */
}

pragma(msg, S.init);
```

```
S(1, 2.5, 'a')
```

auto

Does not mean *automatic* type inference because D has automatic type inference anyway.

`auto` retains its original meaning in D: automatic storage class.

```
const a = 42;
auto b = 43;     // to satisfy declaration syntax
```

No reference to rvalue

Even if it's **`const`**:

```
void foo(ref const(int) i) {
    // ...
}

void main() {
    foo(1 + 2);    // ← compilation ERROR
}
```

"Turtles all the way down"

```
// C++
struct A {
    int * p;
};

struct B {
    A a;
};

    int i = 42;
    const B b{A{&i}};
    *(b.a.p) = 43;    // Should this compile?
```

"Turtles all the way down"

```
// C++
struct A {
    int * p;
};

struct B {
    A a;
};

    int i = 42;
    const B b{A{&i}};
    *(b.a.p) = 43;    // Should this compile?
```

```
// D
struct A {
    int * p;
}

struct B {
    A a;
}

    int i = 42;
    const b =  B(A(&i));
    *(b.a.p) = 43;        // ← compilation ERROR
```

`immutable`

- **`const`** means "I cannot mutate data through this reference but others may."
- **`immutable`** means "immutable".

```
immutable id = 42;
string s = "hi";   // alias of immutable(char)[]
```

  - Less need to copy data; e.g. an API can take a file name as **`string`** without copying
  - **`immutable`** is implicitly **`shared`**
  - No need to lock **`immutable`** data in concurrency
  - More opportunities for compiler optimizations
- Both are transitive.

Move and copy semantics

- Classes don't have rvalues
- Struct rvalues are moved
- Struct lvalues are *blitted* (bit-level transferred)
- Post-blit function when needed

```
struct S {
// ...

    this(this) {
        // ...
    }
}
```

prvalues, xvalues, glvalues, etc. are not in D vocabulary.

Moving rvalues

```
struct S {   // ← Nothing special needed; already efficient!
    Vector v;
}

Vector make_vector() {    // returns rvalue
    Vector v;
    // ...
    return v;
}

S make_S() {              // returns rvalue
    return S();
}

void main() {
    auto v = Vector([ 1, 2, 3 ]);

    auto s0 = S(v);                 // copies v
    auto s1 = S(make_vector());     // moves rvalue
    auto s2 = s0;                   // copies s0
    auto s3 = make_S();             // moves rvalue
}
```

Moving rvalues

```
struct S {   // ← Nothing special needed; already efficient!
    Vector v;
}

Vector make_vector() {    // returns rvalue
    Vector v;
    // ...
    return v;
}

S make_S() {              // returns rvalue
    return S();
}

void main() {
    auto v = Vector([ 1, 2, 3 ]);

    auto s0 = S(v);                 // copies v
    auto s1 = S(make_vector());     // moves rvalue
    auto s2 = s0;                   // copies s0
    auto s3 = make_S();             // moves rvalue
}
```

```
struct Vector {
    int[] elements;

    this(this) {                    // ← post-blit
        elements = elements.dup;
    }
}
```

Module system

- **`import`**, not **`#include`**
- No **`#ifndef`** header guards
- No need for declaration and definition separation (but possible)
- Definition order does not matter in most cases

```
module deneme;      // The name of this module

import std.stdio;   // Using the standard input+output module

void main() {
    sayHi();        // No need to forward declare
}

void sayHi() {
    writeln("Hi!");
}
```

C arrays (?)

- A convention in C: a pointer to a *single* variable coupled with a termination convention
- C arrays are sometimes value types (e.g. when members of structs) and sometimes reference types (e.g. when function parameters)
- C does not have any special array operation (perhaps with the exception of **`memmove()`**). For example, elements may have to be added only after ensuring room explicitly with **`realloc()`**.

For more details, see Walter Bright's article "C's Biggest Mistake":
http://www.drdobbs.com/article/print?articleId=228701625

Static arrays (fixed-length arrays)

- Fixed number of elements
- Value type
- Bounds checking can be done at compile time

```
int[3] array= [ 10, 42, 100 ];
assert(array.length == 3);
array[0] = 11;
int a = array[5];    // ← compilation ERROR
```

- Syntax is consistent and natural: **`Type[length]`**

```
/* Element types are highlighted: */

char[2] array;                // 2 chars

char[2][1] region;            // One char[2]
writeln(region[0].sizeof);    // Prints 2
```

Static arrays are nothing but a contiguous region of N elements. No pointer member, capacity is 0, length is a part of the type.

Dynamic arrays

- Number of elements can vary
- Implemented as a pair of *pointer* and *length*

```
// Equivalent of:
struct Array_of_T__ {
    size_t length;
    T * ptr;
}
```

- **~** operator to *concatenate*
- **~=** operator to *append*

```
    int[] array= [ 10, 42, 100 ];
    array.length += 10;
    assert(array.length == 13);

    array[0] = 11;
    array ~= 7;                         // append

    auto newArray = array ~ array; // concatenate

    int a = array[20];     // run-time error (by default)
```

This is the *slice* interface...

Strings

```
     char c;    // UTF-8 code unit
    wchar w;    // UTF-16 code unit
    dchar d;    // UTF-32 code unit
```

Strings

```
 char c;    // UTF-8 code unit
wchar w;    // UTF-16 code unit
dchar d;    // UTF-32 code unit
```

```
 string s;    // UTF-8 encoded Unicode string
wstring w;    // UTF-16 encoded Unicode string
dstring d;    // UTF-32 encoded Unicode string
```

Strings

```
 char c;    // UTF-8 code unit
wchar w;    // UTF-16 code unit
dchar d;    // UTF-32 code unit
```

```
 string s;    // UTF-8 encoded Unicode string
wstring w;    // UTF-16 encoded Unicode string
dstring d;    // UTF-32 encoded Unicode string
```

```
alias  string = immutable( char)[];
alias wstring = immutable(wchar)[];
alias dstring = immutable(dchar)[];
```

Slices

One of the most useful features of D. Efficient, convenient, and safe...

```
int[] array = [ 10, 20, 30, 40 ];
int[] slice = array[1..3];            // 20 and 30
```

Slices

One of the most useful features of D. Efficient, convenient, and safe...

```
    int[] array = [ 10, 20, 30, 40 ];
    int[] slice = array[1..3];          // 20 and 30
```

Example:

```
bool is_palindrome(string s) {
    if (s.length < 2) {
        return true;
    }

    return (s[0] == s[$-1]) && is_palindrome(s[1..$-1]);
}

unittest {
    assert(is_palindrome("abccba"));
    assert(!is_palindrome("abca"));
}
```

Disclaimer: This example is wrong. (Accidentally works with ASCII strings.)

Slices

One of the most useful features of D. Efficient, convenient, and safe...

```
    int[] array = [ 10, 20, 30, 40 ];
    int[] slice = array[1..3];          // 20 and 30
```

Example:

```
bool is_palindrome(string s) {
    if (s.length < 2) {
        return true;
    }

    return (s[0] == s[$-1]) && is_palindrome(s[1..$-1]);
}

unittest {
    assert(is_palindrome("abccba"));
    assert(!is_palindrome("abca"));
}
```

Disclaimer: This example is wrong. (Accidentally works with ASCII strings.)

Skipping the program name in **`main()`**:

```
void main(string[] args) {
    foo(args[1..$]);
}
```

Associative arrays

A hash table implementation.

A mapping from **`string`** to **`string`**:

```
string[string] colors = [ "red" : "kırmızı",
                          "blue" : "mavi",
                          "green" : "yeşil" ];

writeln(colors["red"]);   // prints "kırmızı"
```

Associative arrays

A hash table implementation.

A mapping from **`string`** to **`string`**:

```
string[string] colors = [ "red" : "kırmızı",
                          "blue" : "mavi",
                          "green" : "yeşil" ];

writeln(colors["red"]);   // prints "kırmızı"
```

A mapping from **`string`** to **`double`**:

```
double[string] universalConstants;

universalConstants["pi"] = 3.14;
universalConstants["e"] = 2.72;
```

Associative arrays

A hash table implementation.

A mapping from **string** to **string**:

```
string[string] colors = [ "red" : "kırmızı",
                          "blue" : "mavi",
                          "green" : "yeşil" ];

writeln(colors["red"]);   // prints "kırmızı"
```

A mapping from **string** to **double**:

```
double[string] universalConstants;

universalConstants["pi"] = 3.14;
universalConstants["e"] = 2.72;
```

Structs and classes can be key types by overloading the **toHash()** member function.

Universal function call syntax (UFCS)

If an object does not have a matching member function, the compiler tries a free-standing function:

```
auto minutes(int i) {
    // ...
}

    minutes(10);    // usual syntax
    10.minutes;     // UFCS syntax
```

Universal function call syntax (UFCS)

If an object does not have a matching member function, the compiler tries a free-standing function:

```
auto minutes(int i) {
    // ...
}

    minutes(10);    // usual syntax
    10.minutes;     // UFCS syntax
```

Problem: The execution order is inside-out:

```
    writeln(evens(divide(multiply(values, 10), 3)));
```

Universal function call syntax (UFCS)

If an object does not have a matching member function, the compiler tries a free-standing function:

```
auto minutes(int i) {
    // ...
}

    minutes(10);    // usual syntax
    10.minutes;     // UFCS syntax
```

Problem: The execution order is inside-out:

```
    writeln(evens(divide(multiply(values, 10), 3)));
```

UFCS makes code more readable:

```
    values.multiply(10).divide(3).evens.writeln;
```

Ranges

C++ uses the iterator abstraction; pointers are iterators

D uses the range abstraction; slices are ranges

```
struct MyInputRange {
    T front();
    bool empty();
    void popFront();
}
```

See discussion on Eric Niebler's CppCon 2015 range presentation, based on H. S. Teoh's work, a D community member:

https://forum.dlang.org/thread/hatpfdftwkycjxwxcthe@forum.dlang.org

Various useful features

Concurrency and parallelism

- Data is thread-local by default
- Only **`shared`** data can be shared (**`__gshared`** is available as well for C-style globals)
- **`immutable`** is implicitly shared
- **`synchronized`** for easy synchronization
- Standard modules
    - **`std.parallelism`**
    - **`std.concurrency`** for message-passing concurrency
    - **`core.atomic`** for atomic operations
    - **`core.thread`** for **`Thread`**, **`Fiber`**, and others
    - **`core.sync.*`** package for classic synchronization primitives

`std.parallelism` module

To execute independent operations simultaneously to make the program run faster.

- Assuming that the following takes 4 seconds on a single core:

```
    auto students =
        [ Student(1), Student(2), Student(3), Student(4) ];

    foreach (student; students) {
        student.aLengthyOperation();
    }
```

`std.parallelism` module

To execute independent operations simultaneously to make the program run faster.

- Assuming that the following takes 4 seconds on a single core:

```
    auto students =
        [ Student(1), Student(2), Student(3), Student(4) ];

    foreach (student; students) {
        student.aLengthyOperation();
    }
```

- The following takes 1 second on 4 cores:

```
    foreach (student; students.parallel) {
        student.aLengthyOperation();
    }
```

std.concurrency module

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

void main() {
    auto worker = spawn(&func);

    worker.send(42);            // note different types of messages
    worker.send("hello");
    worker.send(Terminate());
}

struct Terminate {}

void func() {
    bool done = false;

    while (!done) {
        receive(
            (int msg) {         // ← three lambdas
                // ...
            },

            (string msg) {      // ←
                // ...
            },

            (Terminate msg) {   // ←
                done = true;
            });
    }
}
```

Fibers

Implemented by the druntime and Phobos (the standard library).

```
// Tree traversal
void traverse(const(Node) * node) {
    if (!node) {
        return;
    }

    traverse(node.left);
    yield(node.element);
    traverse(node.right);
}
```

Generator to present a fiber as an InputRange

```
import std.stdio;
import std.range;
import std.concurrency;

void fibonacciSeries() {
    int current = 0;    // ← Not a parameter anymore
    int next = 1;

    while (true) {
        yield(current);

        const nextNext = current + next;
        current = next;
        next = nextNext;
    }
}

void main() {
    auto series = new Generator!int(&fibonacciSeries);
    writefln("%(%s %)", series.take(10));
}
```

```
0 1 1 2 3 5 8 13 21 34
```

Functional programming

- **`immutable`** data
- **`pure`** functions
- Delegates (enabling closures)
- Lazy evaluations are common

```d
import std.stdio;
import std.conv;
import std.algorithm;

void main(string[] args) {
    const input = args[1..$];

    // Sum of the odd numbers on the command line
    const result = input
                   .map!(to!long)
                   .filter!(a => a % 2)
                   .sum;

    writeln(result);
}
```

Functional programming

- **`immutable`** data
- **`pure`** functions
- Delegates (enabling closures)
- Lazy evaluations are common

```
import std.stdio;
import std.conv;
import std.algorithm;

void main(string[] args) {
    const input = args[1..$];

    // Sum of the odd numbers on the command line
    const result = input
                   .map!(to!long)
                   .filter!(a => a % 2)
                   .sum;

    writeln(result);
}
```

```
$ ./deneme 1 2 3 4 5
9
```

`pure` code

- `pure` code cannot access *mutable* global state

  So, they always produce the same result(s) for a given set of arguments.

- Different from most other functional languages, `pure` functions in D *can mutate local state, even their arguments!*

pure example

```
pure long[] fibonacci(size_t n) {
    long[] result;

    if (n > 0) {
        result ~= 0;
        --n;

        if (n > 0) {
            result ~= 1;
            --n;

            foreach (i; 0 .. n) {
                result ~= result[$-1] + result[$-2];
            }
        }
    }

    return result;
}

void main() {
    import std.stdio;
    writeln(fibonacci(10));
}
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Compile time function execution (CTFE)

```
    enum m = makeMenu([ "Pancake", "Waffle" ]);
```

`m` is generated at compile time.

Similarly:

- **`static const`** instead of **`enum`**
- Template argument
- Array size
- etc. (Any expression that is needed at compile time and can be evaluated at compile time will be executed.)

Compile time function execution (CTFE)

```
enum m = makeMenu([ "Pancake", "Waffle" ]);
```

`m` is generated at compile time.

Similarly:

- **`static const`** instead of **`enum`**
- Template argument
- Array size
- etc. (Any expression that is needed at compile time and can be evaluated at compile time will be executed.)

CTFE is being vastly improved by Stefan Koch.

Templates

Function templates

```
auto min(L, R)(L lhs, R rhs) {
    return rhs < lhs ? rhs : lhs;
}
```

Templates

Function templates

```
auto min(L, R)(L lhs, R rhs) {
    return rhs < lhs ? rhs : lhs;
}
```

struct and class templates

```
struct Point(T = long, int dimensions = 2) {
    T[dimensions] coordinates;

    // ...
}
```

Templates

Function templates

```
auto min(L, R)(L lhs, R rhs) {
    return rhs < lhs ? rhs : lhs;
}
```

struct and class templates

```
struct Point(T = long, int dimensions = 2) {
    T[dimensions] coordinates;

    // ...
}
```

```
alias Point3D = Point!(double, 3);

Point3D center;
```

Eponymous templates

From the previous slide:

```
auto min(L, R)(L lhs, R rhs) {
    return rhs < lhs ? rhs : lhs;
}
```

Eponymous templates

From the previous slide:

```
auto min(L, R)(L lhs, R rhs) {
    return rhs < lhs ? rhs : lhs;
}
```

Lowered to the following syntax behind the scenes:

```
template min(L, R) {
    // You can add any supporting code here...

    auto min(L lhs, R rhs) {
        return rhs < lhs ? rhs : lhs;
    }
}
```

General templates

```
template Foo(T, int i, string s, alias func) {
    import std.stdio : writefln;

    size_t count;

    void print() {
        ++count;
        writefln!"%s, %s, %s, %s"(T.stringof, i, s, func());
    }

    void report() {
        writefln("Called %s times.", count);
    }
}
```

General templates

```
template Foo(T, int i, string s, alias func) {
    import std.stdio : writefln;

    size_t count;

    void print() {
        ++count;
        writefln!"%s, %s, %s, %s"(T.stringof, i, s, func());
    }

    void report() {
        writefln("Called %s times.", count);
    }
}
```

```
void main() {
    alias f = Foo!(long, 42, "hello", () => 123);
    f.print();
    f.print();
    f.report();
}
```

General templates

```
template Foo(T, int i, string s, alias func) {
    import std.stdio : writefln;

    size_t count;

    void print() {
        ++count;
        writefln!"%s, %s, %s, %s"(T.stringof, i, s, func());
    }

    void report() {
        writefln("Called %s times.", count);
    }
}
```

```
void main() {
    alias f = Foo!(long, 42, "hello", () => 123);
    f.print();
    f.print();
    f.report();
}
```

```
long, 42, hello, 123
long, 42, hello, 123
Called 2 times.
```

Template constraints

```
auto foo(T)(T t)
if (is (T : long)) {
    // ...
}

auto copy(RIn, ROut)(RIn from, ROut to)
if (isInputRange!RIn &&
    isOutputRange!(ROut, ElementType!RIn)) {
    // ...
}
```

Variadic templates and compile-time `foreach`

```
void PrintAll(Args...)() {
    import std.stdio : writeln;
    foreach (arg; Args) {
        writeln(arg);
    }
}

void main() {
    PrintAll!(1, 2.5, "hello");
}
```

```
1
2.5
hello
```

Variadic templates and compile-time `foreach`

```
void PrintAll(Args...)() {
    import std.stdio : writeln;
    foreach (arg; Args) {
        writeln(arg);
    }
}

void main() {
    PrintAll!(1, 2.5, "hello");
}
```

```
1
2.5
hello
```

The **`foreach`** loop above is unrolled at compile time:

```
    writeln(1);
    writeln(2.5);
    writeln("hello");
```

User-defined type as template parameter

```
import std.stdio;

struct S {
    int i;
    double d;
}

void foo(S s)() {
    writeln(s);
}

void main() {
    foo!(S(42, 2.5));
}
```

Template parameter summary:

- Type
- Value (**`int`**, **`string`**, user-defined type, ...)
- Alias (any symbol in the program)
- Sequence (**`Args...`**)
- **`this`** (provides the type of the **`this`** reference) in member function templates

Template mixins

Templates are for code generation. Generated code can be mixed in:

```
struct S {
    mixin Foo!(double, -1, "world", &bar);
}

int bar() {
    return 456;
}
```

Template mixins

Templates are for code generation. Generated code can be mixed in:

```
struct S {
    mixin Foo!(double, -1, "world", &bar);
}

int bar() {
    return 456;
}
```

```
    auto s = S();
    5.iota.each!(i => s.print());
    s.report();
```

Template mixins

Templates are for code generation. Generated code can be mixed in:

```
struct S {
    mixin Foo!(double, -1, "world", &bar);
}

int bar() {
    return 456;
}
```

```
    auto s = S();
    5.iota.each!(i => s.print());
    s.report();
```

```
double, -1, world, 456
double, -1, world, 456
double, -1, world, 456
double, -1, world, 456
double, -1, world, 456
Called 5 times.
```

String mixins

Can generate and mix code in at compile time.

```
string makeStructDef(string name, size_t N) {
    import std.string : format;
    return format(`
        struct %s {
            int[%s] arr;
        }
    `, name, N);
}
```

String mixins

Can generate and mix code in at compile time.

```
string makeStructDef(string name, size_t N) {
    import std.string : format;
    return format(`
        struct %s {
            int[%s] arr;
        }
    `, name, N);
}
```

```
pragma(msg, makeStructDef("Point", 3));
```

```
        struct Point {
            int[3] arr;
        }
```

String mixins

Can generate and mix code in at compile time.

```
string makeStructDef(string name, size_t N) {
    import std.string : format;
    return format(`
        struct %s {
            int[%s] arr;
        }
    `, name, N);
}
```

```
pragma(msg, makeStructDef("Point", 3));
```

```
        struct Point {
            int[3] arr;
        }
```

```
mixin (makeStructDef("Point", 3));

void main() {
    Point p;
}
```

Operator overloading

```
struct MyInt {
    int i;

    auto opBinary(string op)(MyInt that) {
        mixin ("return MyInt(this.i " ~ op ~ " that.i);");
    }
}
```

Operator overloading

```
struct MyInt {
    int i;

    auto opBinary(string op)(MyInt that) {
        mixin ("return MyInt(this.i " ~ op ~ " that.i);");
    }
}
```

```
unittest {
    assert(MyInt(1) + MyInt(2) == MyInt(3));
    assert(MyInt(4) - MyInt(3) == MyInt(1));
    // ...
}
```

opDispatch

Borrowing the idea from Adam Ruppe's book "D Cookbook".

```
struct Style {
    int[string] values;

    // Getters
    auto opDispatch(string calledWith)() {
        return values[calledWith];
    }

    // Setters
    auto opDispatch(string calledWith)(int value) {
        values[calledWith] = value;
    }
}
```

opDispatch

Borrowing the idea from Adam Ruppe's book "D Cookbook".

```
struct Style {
    int[string] values;

    // Getters
    auto opDispatch(string calledWith)() {
        return values[calledWith];
    }

    // Setters
    auto opDispatch(string calledWith)(int value) {
        values[calledWith] = value;
    }
}
```

```
unittest {
    auto s = Style([ "color" : 42, "weight" : 100 ]);
    assert(s.color == 42);
    assert(s.weight == 100);

    s.noWay = 7;
    assert(s.noWay == 7);
}
```

Note: Instead of the runtime table lookup above, opDispatch() could have mixed code in e.g. to parse and translate from json format, etc.

AliasSeq (alias sequence)

A heterogeneous list of arguments:

```
import std.meta;

void foo(int, double) {
}

void main() {
    alias a = AliasSeq!(int, "hello", 42, 2.5);
    alias b = a[1..$];                         // drop first arg
    alias c = AliasSeq!(b, double);    // add 'double'
    // c is "hello", 42, 2.5, double

    foo(c[1..3]);    // Call function with an argument list
}
```

`static foreach`

The compile-time **`foreach`** that we saw earlier worked on template arguments and the unrolling was implicit. **`static foreach`** works with compile-time generated ranges.

Timon Gehr implemented **`static foreach`** by hacking for two days at DConf 2017.

```
    static foreach (i; someRange) {
        // The loop body is unrolled for each iteration
    }
```

Traits

`std.traits` module

```
isFunction, Parameters, ...
isType, isNumeric, ...
Fields, hasElaborateDestructor, ...
isAbstractClass, isInstanceOf, ...
```

Traits

`std.traits` module

```
isFunction, Parameters, ...
isType, isNumeric, ...
Fields, hasElaborateDestructor, ...
isAbstractClass, isInstanceOf, ...
```

`__traits` keyword

```
getVirtualFunctions, isTemplate, allMembers, ...
classInstanceSize, compiles, ...
```

```
void foo(T)(T t) {
    static if (__traits(compiles, t.bar(int.init))) {
        t.bar(42);
    } else {
        // ...
    }
}
```

.tupleof

```
import std.stdio;

struct S {
    int i;
    string s;
}

void main() {
    auto s = S(42, "hello");
    foreach (i, member; s.tupleof) {
        writefln("Member %s is %s %s",
                 i, typeof(member).stringof, member);
    }
}
```

```
Member 0 is int 42
Member 1 is string hello
```

`static if` for C++?

Proposed by Walter Bright as a D invention:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3329.pdf

`static if` for C++?

Proposed by Walter Bright as a D invention:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3329.pdf

"Considered":

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3613.pdf

`static if` for C++?

Proposed by Walter Bright as a D invention:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3329.pdf

"Considered":

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3613.pdf

Re-proposed as **`static_if`**:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4461.html

`static if` for C++?

Proposed by Walter Bright as a D invention:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3329.pdf

"Considered":

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3613.pdf

Re-proposed as **`static_if`**:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4461.html

A subset of **`static if`** accepted as **`if constexpr`**:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0292r1.html

`static if`

Very useful...

```
auto r = stdin.byLine.take(5);
auto v = r[2];                    // ← compilation ERROR
```

That indexed access cannot be compiled because

- **`take`** cannot provide it because
- **`byLine`** cannot provide it.

(Essentially, they are both **`InputRange`**s.)

`static if`

Very useful...

```
auto r = stdin.byLine.take(5);
auto v = r[2];                          // ← compilation ERROR
```

That indexed access cannot be compiled because

- **`take`** cannot provide it because
- **`byLine`** cannot provide it.

(Essentially, they are both **`InputRange`**s.)

How does the following indexed access work then?

```
auto r = 10.iota.map!(i => i * 2).take(5);
auto v = r[2];                          // compiles!
```

`static if`

Very useful...

```
auto r = stdin.byLine.take(5);
auto v = r[2];                          // ← compilation ERROR
```

That indexed access cannot be compiled because

- **`take`** cannot provide it because
- **`byLine`** cannot provide it.

(Essentially, they are both **`InputRange`**s.)

How does the following indexed access work then?

```
auto r = 10.iota.map!(i => i * 2).take(5);
auto v = r[2];                          // compiles!
```

This time

- **`take`** provides it because
- **`map`** provides it
- because **`iota`** provides it.

static if in take

An excerpt from **std.range.Take** template:

```
struct Take(Range)
// ... template constraints ...
{
    // ...

    static if (isRandomAccessRange!R)
    {    // ← NOTE: DOES NOT INTRODUCE A SCOPE
         //   (This was one of C++'s objections.)

        auto ref opIndex(size_t index)
        {
            assert(index < length,
                "Attempting to index out of the bounds of a "
                ~ Take.stringof);
            return source[index];
        }

        // ...
    }

    // ...
}
```

User defined attributes (UDA)

```
struct Obfuscated {
    // ...
}

struct User {
    string name;
    @Obfuscated string password;
}
```

User defined attributes (UDA)

```
struct Obfuscated {
    // ...
}

struct User {
    string name;
    @Obfuscated string password;
}
```

```
void serialize(T)(T object) {
    foreach (member; __traits(allMembers, T)) {
        // ...
        static if (hasUDA!(fullName, Obfuscated)) {
            // ... obfuscate this member ...
        }
        // ...
    }
}

void main() {
    auto user = User("Alice", "mypetsname");
    serialize(user);
}
```

```
name: Alice
password: nzqfutobnf
```

Design by introspection (DbI)

Andrei Alexandrescu's DConf 2017 presentation:

DbI prerequisites:

- DbI Input: `tupleof`, `__traits`, etc.
- DbI Processing: CTFE, `static if`, etc.
- DbI Output: template expansion, mixin, etc.

"Each use of static if doubles the design space covered"

DbI example: Checked int

Andrei Alexandrescu's DConf 2017 presentation:

```
struct MyHook {
  alias
    onBadCast = Abort.onBadCast,
    onLowerBound = Saturate.onLowerBound,
    onUpperBound = Saturate.onUpperBound,
    onOverflow = Saturate.onOverflow,
    hookOpEquals = Abort.hookOpEquals,
    hookOpCmp = Abort.hookOpCmp;
}
alias MyInt = Checked!(int, MyHook);
```

Checked int implementation

Andrei Alexandrescu's DConf 2017 presentation:

```
ref Checked opUnary(string op)() return
if (op == "++" || op == "--") {
    static if (hasMember!(Hook, "hookOpUnary"))
        hook.hookOpUnary!op(payload);
    else static if (hasMember!(Hook, "onOverflow")) {
        static if (op == "++") {
            if (payload == max.payload)
                payload = hook.onOverflow!"++"(payload);
            else
                ++payload;
        } else {
            if (payload == min.payload)
                payload = hook.onOverflow!"--"(payload);
            else
                --payload;
        }
    } else
        mixin(op ~ "payload;");
    return this;
}
```

SafeD

Functions defined as **`@safe`** and modules compiled with **`-safe`** cannot corrupt memory.

Examples of what is *not* allowed in SafeD:

- Inline-assembly
- Conversions between values and pointers
- Potentially unsafe pointer uses

  However, **`T1*`** can convert to **`T2*`** in the safe direction. For example, **`T* → void*`** or **`int* → short*`**.
- Removing **`const`**, **`immutable`**, or **`shared`** attribute
- etc.

Returning reference to local variable

We want to be able to return **`ref`** parameters (think **`min()`**):

```
ref int foo(ref int i) {
    return i;
}
```

Returning reference to local variable

We want to be able to return **`ref`** parameters (think **`min()`**):

```
ref int foo(ref int i) {
    return i;
}
```

The problem at the caller:

```
ref int bar() {
    int i;
    return foo(i); // uh-oh!
}
```

Preventing returning reference to local variable

```
ref int foo(return ref int i) {
    return i;
}

ref int bar() {
    int i;
    int j = foo(i); // ok
    return foo(i);  // ← compilation ERROR
}
```

Similarly for pointers

```
int* foo(scope int* p, int** pp) {
    abc(p);     // ← compilation ERROR
    q = p;      // ← compilation ERROR
    *pp = p;    // ← compilation ERROR
    return p;   // ← compilation ERROR
}
```

Software engineering support

Unit testing

```
string repeat(string s, size_t count) {
    // ...
}

///
unittest {
    assert(repeat("abc", 2) == "abcabc");
    assert(repeat("ğ", 5) == "ğğğğğ");
    assert(repeat("a", 0) == "");
}
```

- Enabled when compiled with the **`-unittest`** compiler switch.
- **`unittest`** blocks that have documentation comments (e.g. `///`) are added to the documentation

  More capable unit testing frameworks exist. (e.g. Átila Neves's unit-threaded)

Source documentation

These slides are made with DDOC.

```
/**
  Repeats a string $(D_CODE count) times.

  Params:
    s = The string to repeat
    count = The number of times to repeat
  Returns:
    A new string consisting of $(D_CODE s) repeated $(C count) times
  Throws:
    $(D_CODE OutOfMemoryException)
*/
string repeat(string s, size_t count) {
    // ...
}

///
unittest {
    // ...
}
```

```
$ dmd -D deneme.d
$ ls deneme.html
deneme.html
```

- **`unittest`** blocks that are marked with `///` are added to the documentation
- Use with CSS for documentation style

The `scope` statement

Obviates many cases of RAII.

- **`scope(failure)`**: When the scope is exited due to an exception
- **`scope(success)`**: When the scope is exited normally
- **`scope(exit)`**: When the scope is exited under any condition

```
    a = initialize();
    scope(exit) cleanup(a);
```

```
    scope(failure) if (exists(tmpFile)) remove(tmpFile);
```

Also see Átila Neves's "automem: Hands-Free RAII for D".

Contract programming

`in` for entry conditions, **`out`** for exit conditions:

```
string repeat(string s, size_t count)
in {
    assert(!s.empty);

} out (result) {
    assert(result.length == (s.length * count));

} body {
    string result;
    // ...
    return result;
}
```

Contract programming

`in` for entry conditions, **`out`** for exit conditions:

```
string repeat(string s, size_t count)
in {
    assert(!s.empty);

} out (result) {
    assert(result.length == (s.length * count));

} body {
    string result;
    // ...
    return result;
}
```

`invariant` for object invariants:

```
class WallClock {
    int hour;
    int minute;

    invariant() {
        assert((hour >= 0) && (hour <= 23));
        assert((minute >= 0) && (minute <= 59));
    }

    // ... member functions that mutate 'hour' and 'minute' ...
}
```

Code coverage

```
int foo(int i) {
    if (i % 2) {
        return 100;

    } else {
        return 200;    // this line is never exercised
    }
}

unittest {
    assert(foo(3) == 100);
}

void main() {
}
```

Code coverage output

```
$ dmd -cov -unittest foo.d
$ ./foo
$ cat foo.lst
       |int foo(int i) {
      1|    if (i % 2) {
      1|        return 100;
       |
       |    } else {
0000000|        return 200;
       |    }
       |}
       |
       |unittest {
      1|    assert(foo(3) == 100);
       |}
       |
       |void main() {
       |}
foo.d is 75% covered
```

Profiling

```
void main() {
    foo(10);
}

void foo(int count) {
    foreach (i; 0 .. count) {
        if (i % 2) {
            bar(i);    // bar is called 5 times
        }
    }
}

void bar(int i) {
    if (i % 3) {
        zar(i);
    }
}

void zar(int i) {
}
```

Profiling output

```
$ dmd -profile -g deneme.d
$ ./deneme
$ cat trace.log

[...]
======== Timer Is 3579545 Ticks/Sec, Times are in Microsecs ========

  Num          Tree        Func        Per
  Calls        Time        Time        Call

      1         443         288         288     void deneme.foo(int)
      5         154         148          29     void deneme.bar(int)
      1         522          79          79     _Dmain
      3           5           5           1     void deneme.zar(int)
```

deprecated

```
deprecated("Please use doSomething() instead.")
void do_something() {
    // ...
}
```

```
deprecated("Import core.stdc.math instead")
module std.c.math;
// ... rest of the module ...
```

```
$ dmd --help
[...]
  -d          silently allow deprecated features
  -dw         show use of deprecated features as warnings (default)
  -de         show use of deprecated features as errors (halt compilation)
[...]
```

Conclusion

D gives a competitive advantage by making programmers more productive.

:C

:C++

:D

Thank You!