Attribution/License

accu conference 2024

How DLang Improves My Modern C++ and Vice Versa

Mike Shah

accu conference 2024

How DLang Improves My Modern C++ and Vice Versa

Mike Shah

Throughout the talk there will be interactive prompts:

- Go to slido.com
- Enter meeting #
- OR scan QR Code

Join at
slido.com
#2649 488

accu conference 2024

How DLang Improves My Modern C++ and Vice Versa

Mike Shah

11:00 - 12:30 BST Wed. April 17, 2024

75 minutes + 15 minute Q&A After
Audience: For all and all skill levels!

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube
www.youtube.com/c/MikeShah
http://tinyurl.com/mike-talks

accu conference 2024

How DLang Improves My Modern C++ and Vice Versa

Mike Shah

Throughout the talk there will be interactive prompts:

- Go to slido.com
- Enter meeting #
- OR scan QR Code

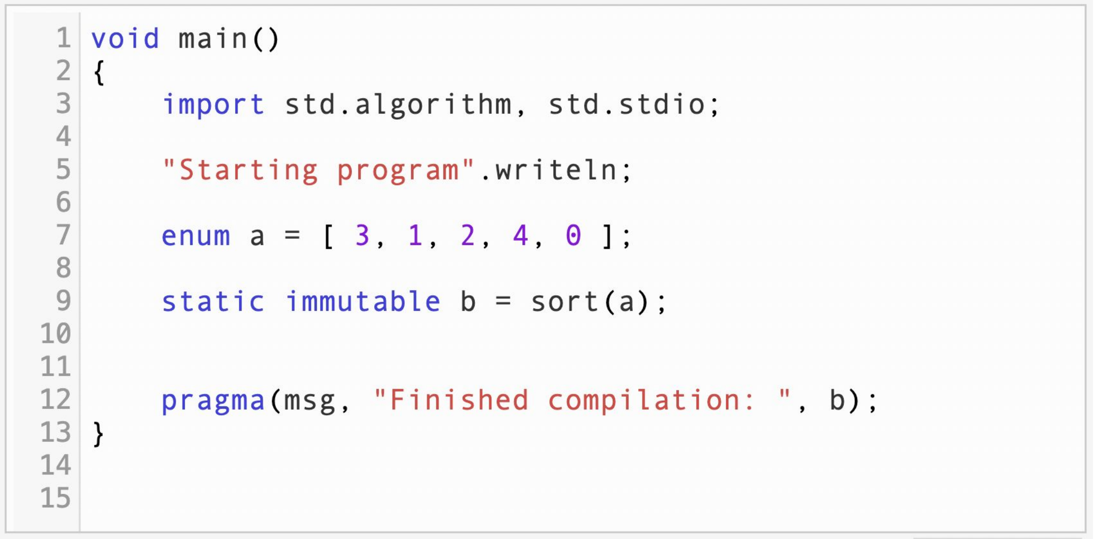
Join at
slido.com
#2649 488

DLang a First Impression

Note: ‘DLang’ often yields better results on search engines versus searching ‘D’ -- thus I’ll usually use ‘Dlang’ to refer to the language

Pop Quiz: (l'examen surprise!) (1/3)

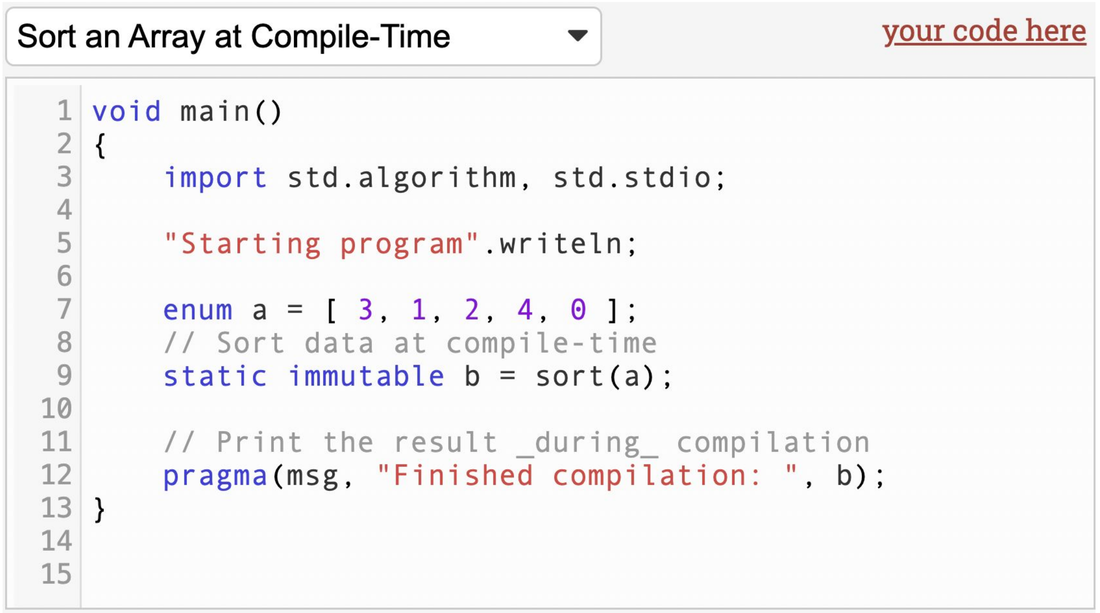
- Let's take a look at an example of D code
  - I'll give everyone a minute to think about it
  - Try to think about what is being done
  - So... what does this program do?



```d
void main()
{
    import std.algorithm, std.stdio;

    "Starting program".writeln;

    enum a = [ 3, 1, 2, 4, 0 ];

    static immutable b = sort(a);


    pragma(msg, "Finished compilation: ", b);
}
```

Pop Quiz: (l'examen surprise!) (2/3)

- One of the first examples on the www.dlang.org webpage
  - An example of sorting an array!
  - Line 3:
    - There's a built-in standard library (named 'Phobos')
  - Line 5:
    - Function call using universal function call syntax (UFCS)
  - Line 7:
    - enum constant -- initializing a fixed-size array
  - Line 9:
    - immutable static data stored in b
  - Line 12:
    - pragma outputs value after compilation
- Amazingly this program does most of its work at compile-time!

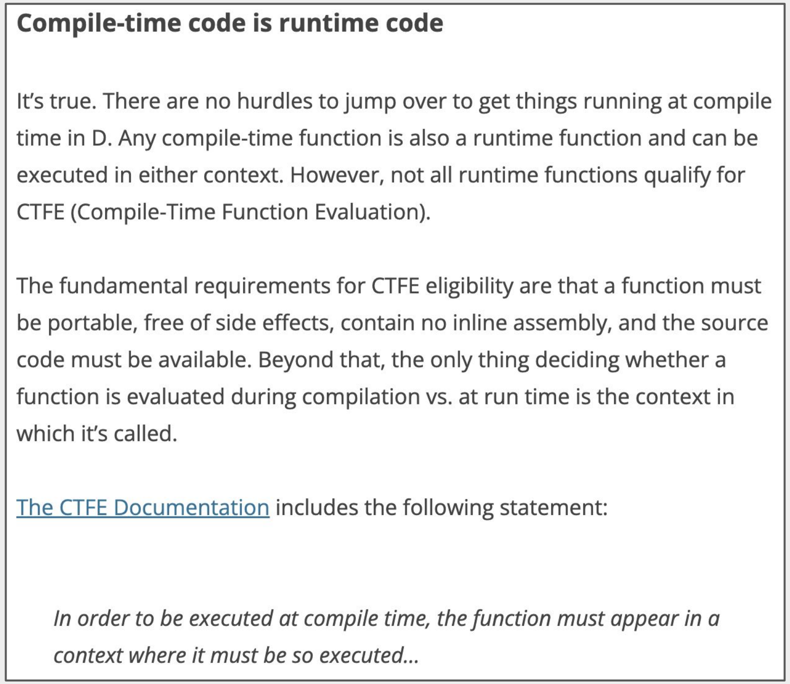


```
 1 void main()
 2 {
 3     import std.algorithm, std.stdio;
 4 
 5     "Starting program".writeln;
 6 
 7     enum a = [ 3, 1, 2, 4, 0 ];
 8     // Sort data at compile-time
 9     static immutable b = sort(a);
10 
11     // Print the result _during_ compilation
12     pragma(msg, "Finished compilation: ", b);
13 }
14 
15 
```

Why you might care to look?

- D tries to **execute as much as possible at compile-time**
  - And the code...just looks like regular code!
- Compile-time execution saves the user time at run-time -- big win!

- https://dlang.org/blog/2017/06/05/compile-time-sort-in-d/
- https://tour.dlang.org/tour/en/gems/compile-time-function-evaluation-ctfe



Compile-time code is runtime code

It's true. There are no hurdles to jump over to get things running at compile time in D. Any compile-time function is also a runtime function and can be executed in either context. However, not all runtime functions qualify for CTFE (Compile-Time Function Evaluation).

The fundamental requirements for CTFE eligibility are that a function must be portable, free of side effects, contain no inline assembly, and the source code must be available. Beyond that, the only thing deciding whether a function is evaluated during compilation vs. at run time is the context in which it's called.

The CTFE Documentation includes the following statement:

> *In order to be executed at compile time, the function must appear in a context where it must be so executed...*

your code here

```
d.stdio;
n;
ing_ compilation
npilation: ", b);
```

- pragma outputs value after compilation
- This program does most of its work (the working) at compile-time!

Your Tour Guide for Today

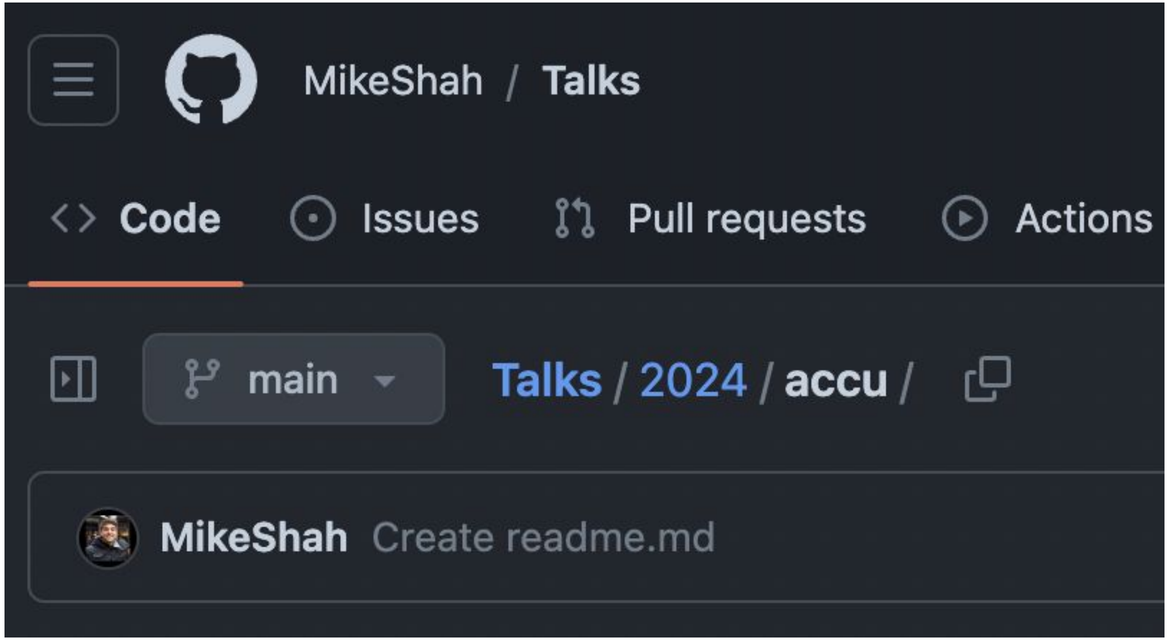
by Mike Shah

- Associate Teaching Professor at Northeastern University in Boston, Massachusetts.
  - I **love** teaching: courses in computer systems, computer graphics, geometry, and game engine development.
  - My research is divided into computer graphics (geometry) and software engineering (software analysis and visualization tools).
- I do actively write code and do consulting and technical training on modern C++, DLang, Concurrency, and Graphics Programming
  - Usually graphics or games related -- e.g. Building 3D application plugins
- Outside of work: guitar, running/weights, traveling and cooking are fun to talk about

Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

Code for the talk

- Located here: https://github.com/MikeShah/Talks/tree/main/2024/accu



Abstract

The abstract that you read and enticed you to join me is here!

The D programming language (DLang) is a multi-paradigm language (like C++) developed to solve real software engineering problems. DLang has a rich history since its inception in 2001, and continues to be an actively evolving memory-safe language used in industry. In this talk, I will discuss how learning and using the D language has directly benefited my use and learning of C++ and vice versa. We'll look at the evolution of both C++ and Dlang, and see how each language has borrowed from each other during their most recent evolution in the past decade. Throughout the talk, I will provide side-by-side code comparisons showing idiomatic ways to complete tasks in D alongside C++ code examples. The goal of this talk however is not to pit one language against the other, but rather to show how to use each language to its strengths and learn how to become a better programmer. Audience members are expected to be familiar with Modern C++, but are not expected to have any prior D programming experience.

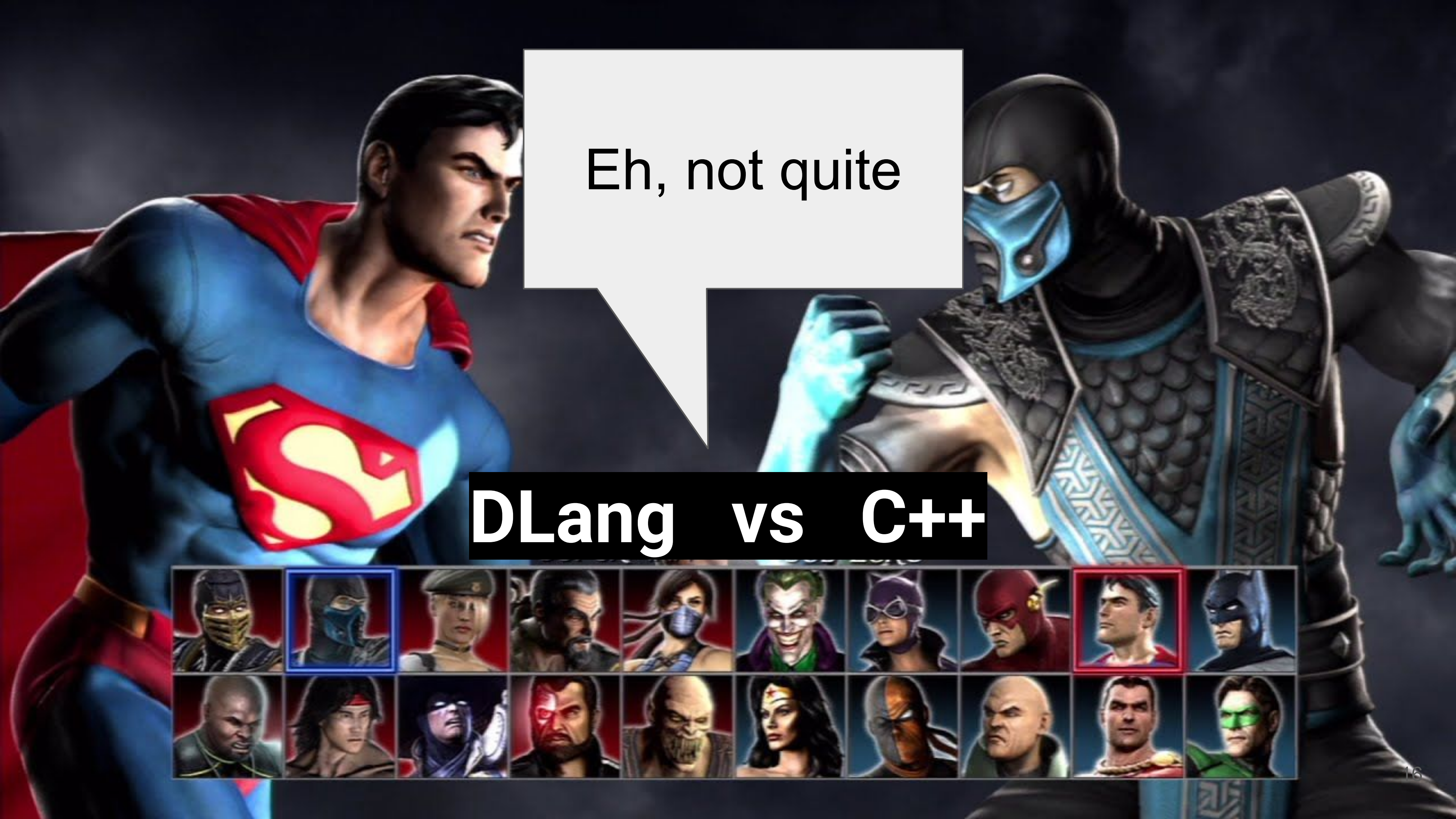
Talk Outline

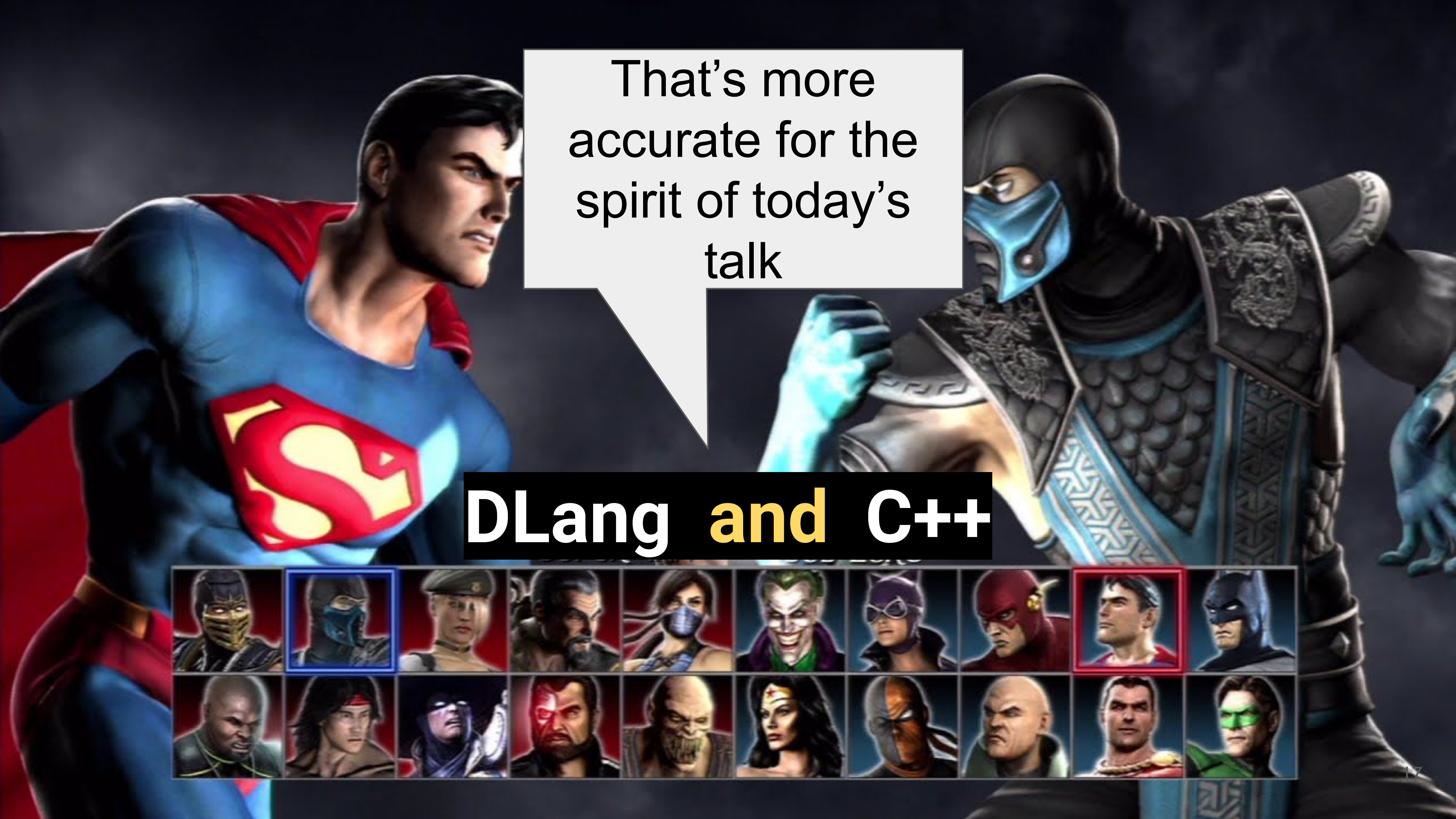
This talk consists of three pieces

1. Some Thoughts on Programming Languages
2. A D Language preview
3. C++ and DLang as they compliment each other

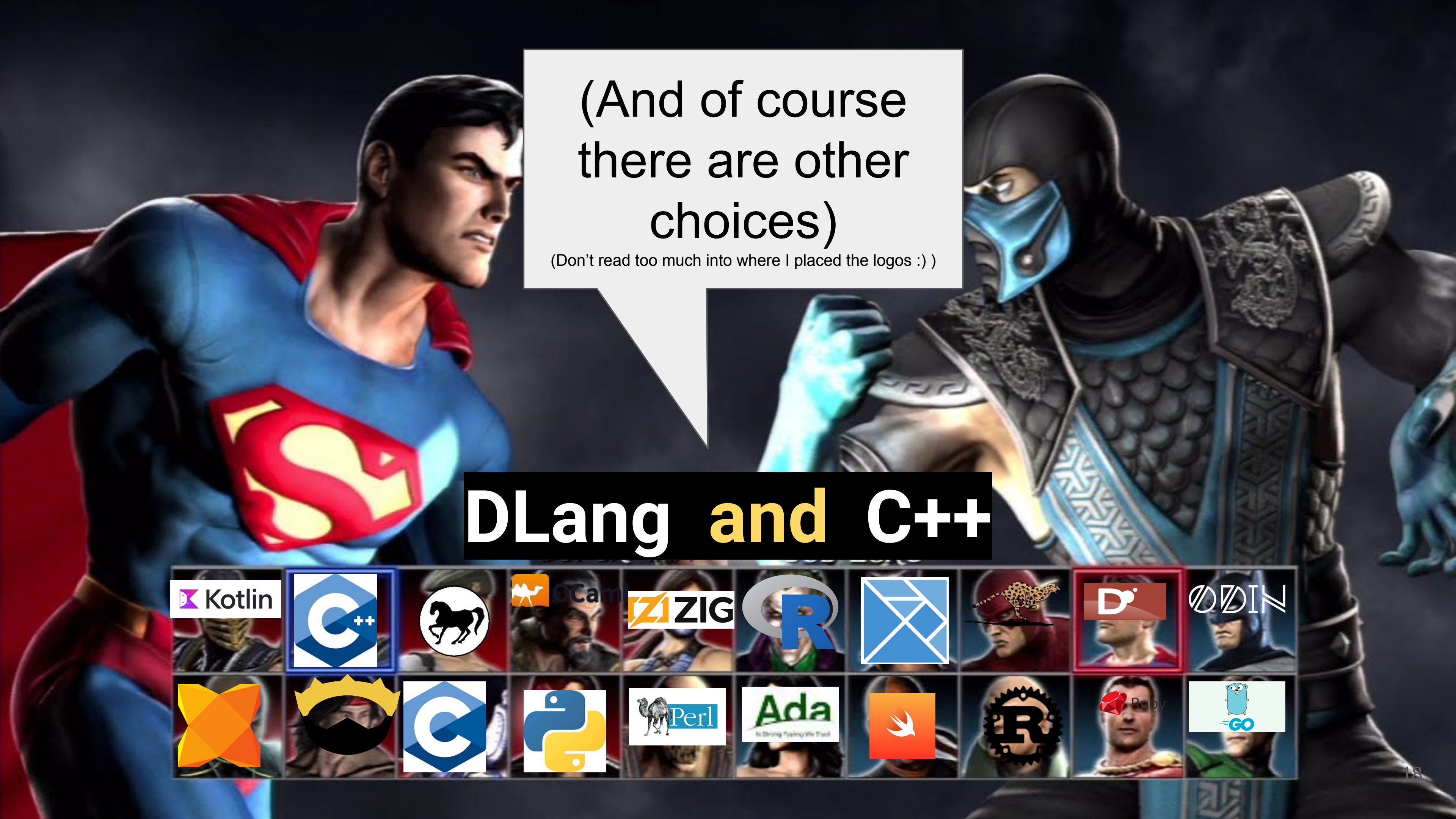
1. **Some Thoughts on Programming Languages**
2. A D Language preview
3. C++ and DLang as they compliment each other

DLang vs C++


Eh, not quite
DLang vs C++



DLang **and** C++


(And of course there are other choices)
(Don't read too much into where I placed the logos :))
DLang and C++
Kotlin
OCaml
ZIG
R
D
ODIN
Perl
Ada
Ruby
GO

ODIN

Kotlin

So I’m a bit of a programming language enthusiast

Perl

Ruby

OCaml

GO

(Free Pascal)

Ada

ZIG

I’ve explored many languages by choice

(Free Pascal)

Many of you use one or more of these programming languages

(Free Pascal)

Multiple languages may be needed for work

Maybe it is required for the platform

Or the domain you work in

(Free Pascal)

Maybe you get to choose

(Free Pascal)

And I bet at some time -- someone said to you....

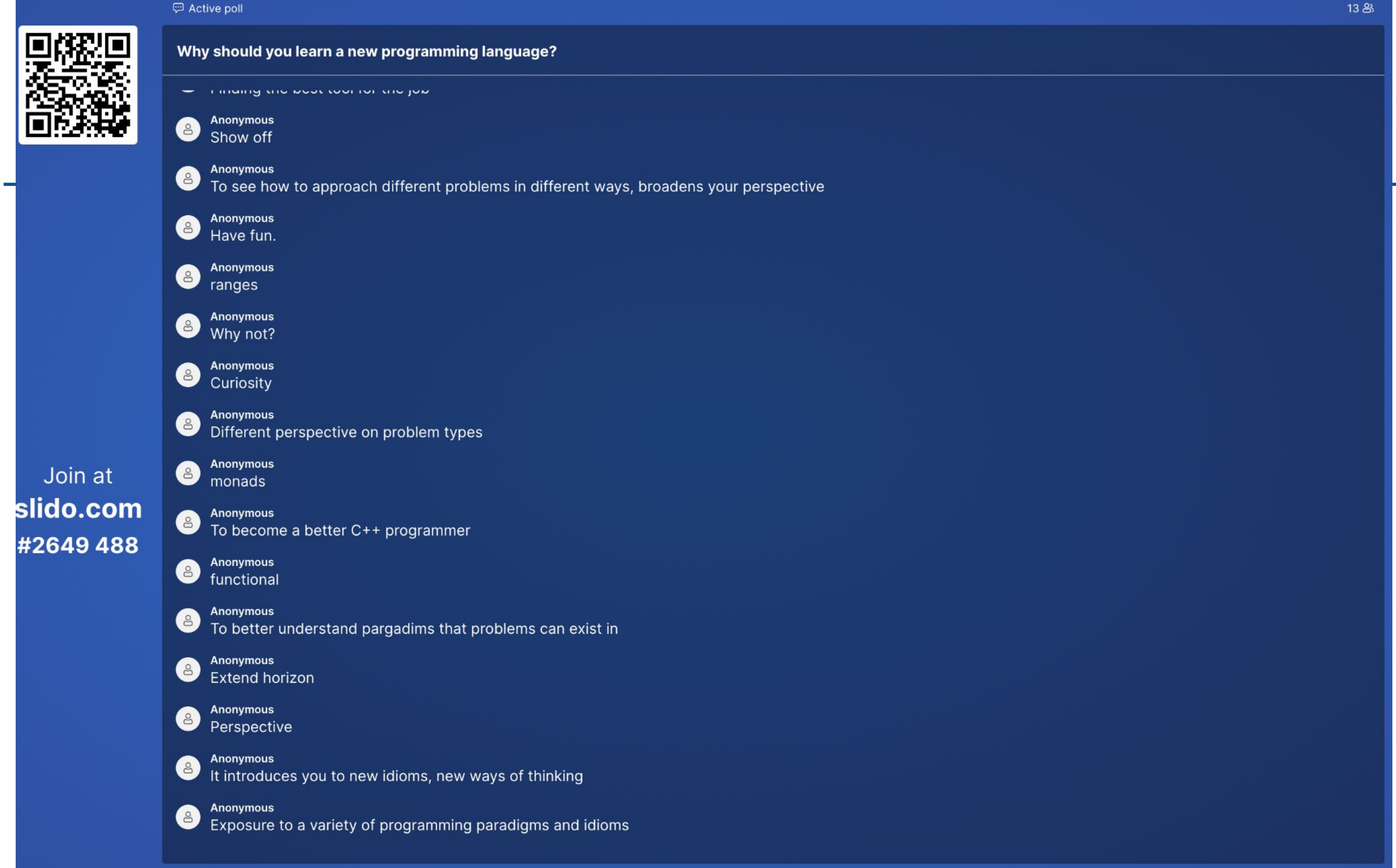
You should just learn Haskell

Haskell

An advanced, purely functional programming language

But why? :)

(Free Pascal)



Active poll 13

Why should you learn a new programming language?

Finding the best tool for the job

Anonymous
Show off

Anonymous
To see how to approach different problems in different ways, broadens your perspective

Anonymous
Have fun.

Anonymous
ranges

Anonymous
Why not?

Anonymous
Curiosity

Anonymous
Different perspective on problem types

Anonymous
monads

Anonymous
To become a better C++ programmer

Anonymous
functional

Anonymous
To better understand pargadims that problems can exist in

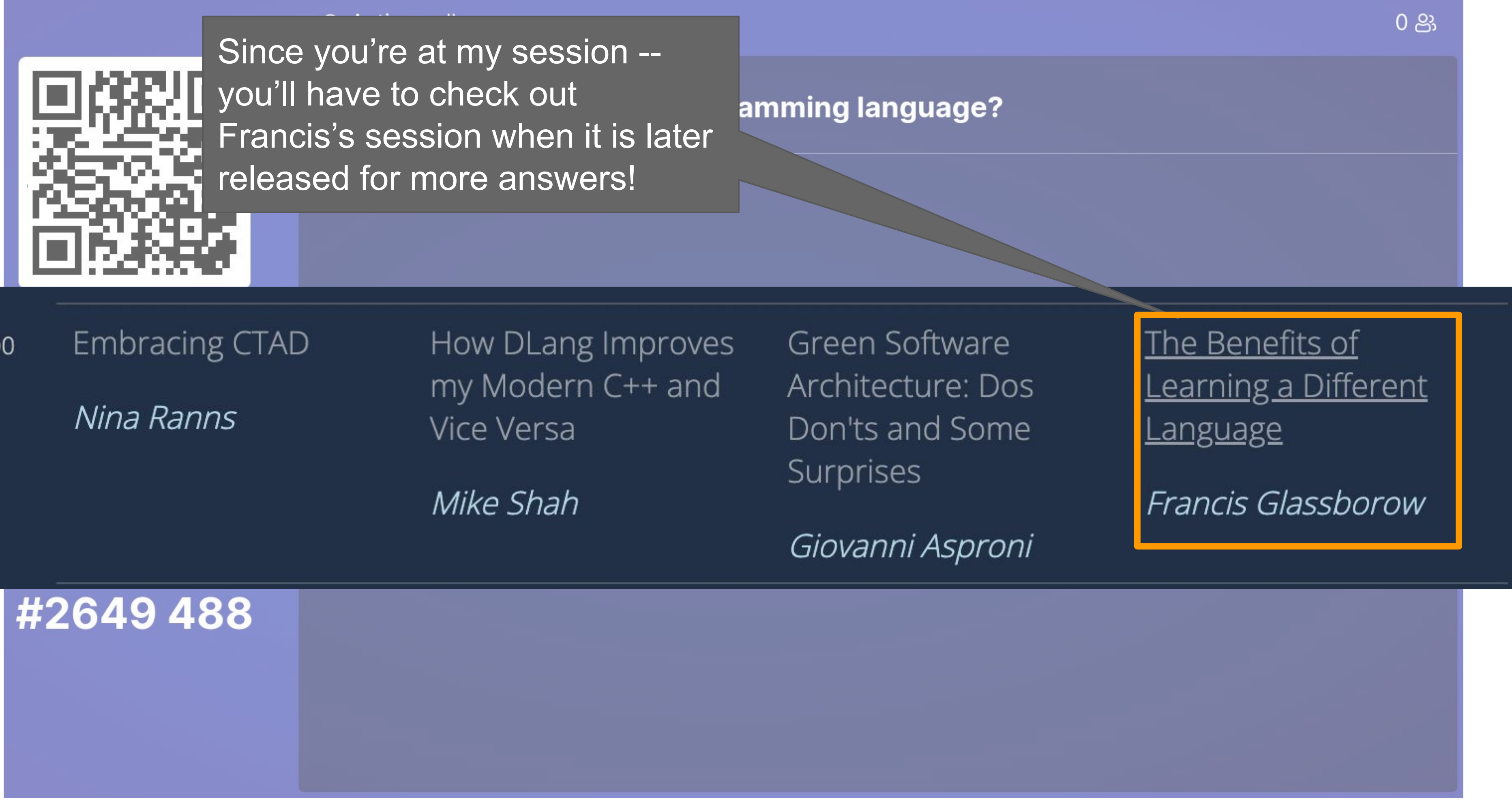
Anonymous
Extend horizon

Anonymous
Perspective

Anonymous
It introduces you to new idioms, new ways of thinking

Anonymous
Exposure to a variety of programming paradigms and idioms

Join at
slido.com
#2649 488



Since you're at my session -- you'll have to check out Francis's session when it is later released for more answers!

amming language?

0

11:00	Embracing CTAD *Nina Ranns*	How DLang Improves my Modern C++ and Vice Versa *Mike Shah*	Green Software Architecture: Dos Don'ts and Some Surprises *Giovanni Asproni*	The Benefits of Learning a Different Language *Francis Glassborow*

#2649 488

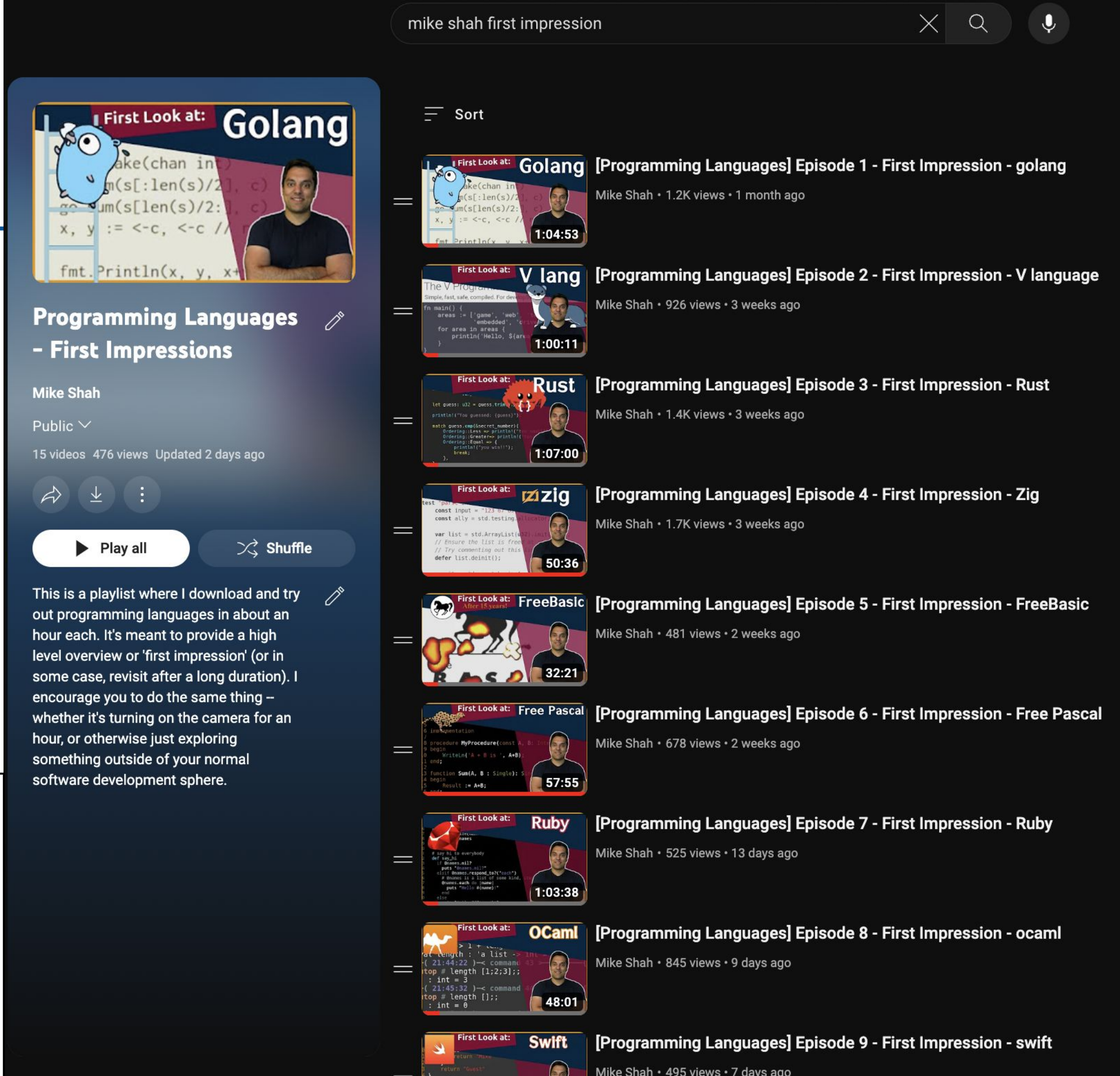
https://wall.sli.do/event/9R85JSdWhaghDHjs6DRMfK?section=ef9c8df6-9115-4740-b00b-8ceb0ee6aab0

The past few months...

- I’ve been documenting myself trying new programming languages for about one hour
  - Most languages are new to me.
  - Some languages are very popular
  - Some languages are less mainstream

My recordings of 23 (and counting) programming languages can be found on the playlist below

Playlist -- *Programming Languages - First Impressions*: https://www.youtube.com/playlist?list=PLvv0ScY6vfd-5hJ47DNAOKKLLIHjz1Tzq


mike shah first impression

Programming Languages - First Impressions
Mike Shah
Public
15 videos 476 views Updated 2 days ago

This is a playlist where I download and try out programming languages in about an hour each. It's meant to provide a high level overview or 'first impression' (or in some case, revisit after a long duration). I encourage you to do the same thing -- whether it's turning on the camera for an hour, or otherwise just exploring something outside of your normal software development sphere.

Sort

[Programming Languages] Episode 1 - First Impression - golang
Mike Shah • 1.2K views • 1 month ago

[Programming Languages] Episode 2 - First Impression - V language
Mike Shah • 926 views • 3 weeks ago

[Programming Languages] Episode 3 - First Impression - Rust
Mike Shah • 1.4K views • 3 weeks ago

[Programming Languages] Episode 4 - First Impression - Zig
Mike Shah • 1.7K views • 3 weeks ago

[Programming Languages] Episode 5 - First Impression - FreeBasic
Mike Shah • 481 views • 2 weeks ago

[Programming Languages] Episode 6 - First Impression - Free Pascal
Mike Shah • 678 views • 2 weeks ago

[Programming Languages] Episode 7 - First Impression - Ruby
Mike Shah • 525 views • 13 days ago

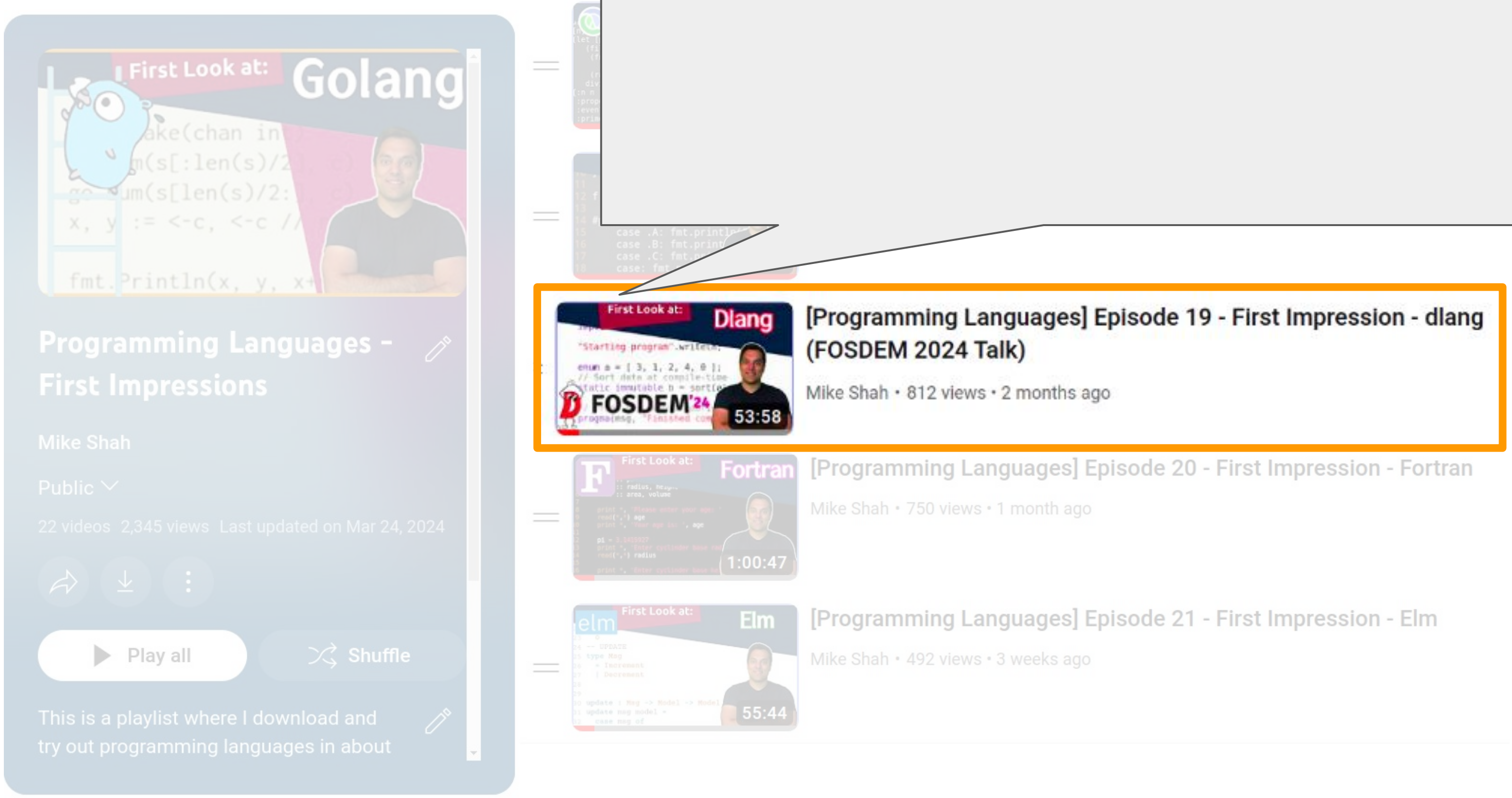
[Programming Languages] Episode 8 - First Impression - ocaml
Mike Shah • 845 views • 9 days ago

[Programming Languages] Episode 9 - First Impression - swift
Mike Shah • 495 views • 7 days ago

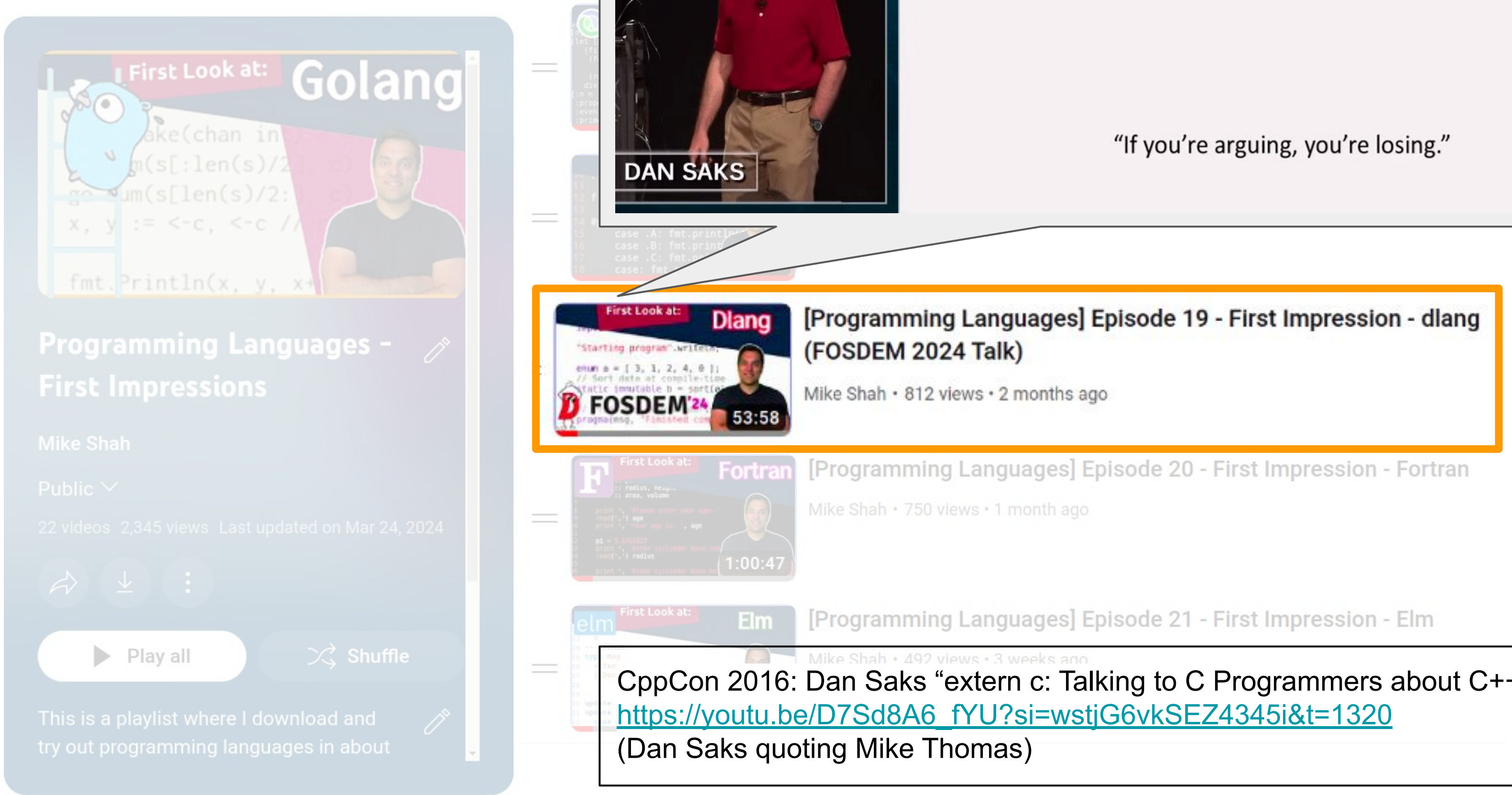
The Art of Persuasion... (1/5)

One important thing to keep in mind in this talk is that:

- I am not going to pick a **better** of two languages


First Look at: Golang
Programming Languages - First Impressions
Mike Shah
Public
22 videos 2,345 views Last updated on Mar 24, 2024
Play all
Shuffle
This is a playlist where I download and try out programming languages in about
[Programming Languages] Episode 19 - First Impression - dlang (FOSDEM 2024 Talk)
Mike Shah • 812 views • 2 months ago
53:58
[Programming Languages] Episode 20 - First Impression - Fortran
Mike Shah • 750 views • 1 month ago
1:00:47
[Programming Languages] Episode 21 - First Impression - Elm
Mike Shah • 492 views • 3 weeks ago
55:44

The Art of Persuasion... (2/5)


Programming Languages - First Impressions

Mike Shah

Public

22 videos 2,345 views Last updated on Mar 24, 2024

This is a playlist where I download and try out programming languages in about

[Programming Languages] Episode 19 - First Impression - dlang (FOSDEM 2024 Talk)

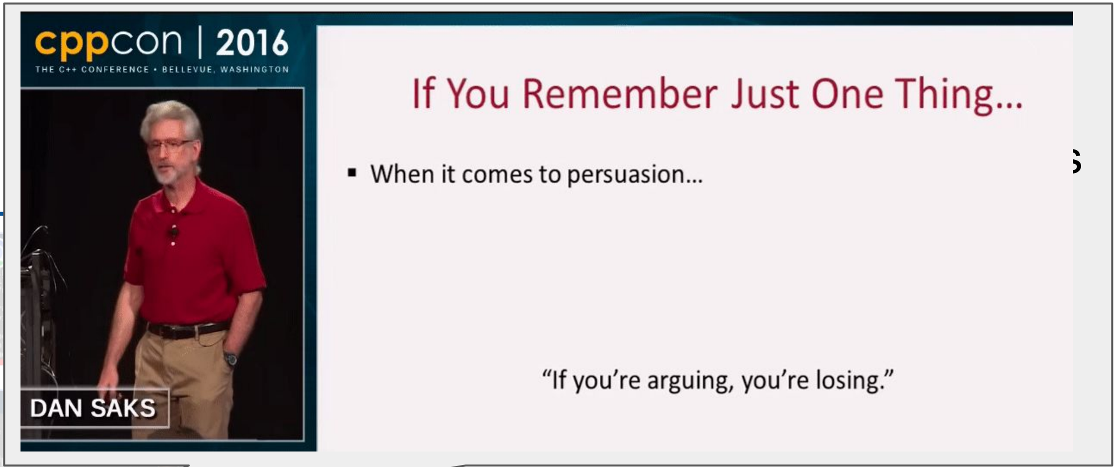
Mike Shah • 812 views • 2 months ago

[Programming Languages] Episode 20 - First Impression - Fortran

Mike Shah • 750 views • 1 month ago

[Programming Languages] Episode 21 - First Impression - Elm

Mike Shah • 492 views • 3 weeks ago



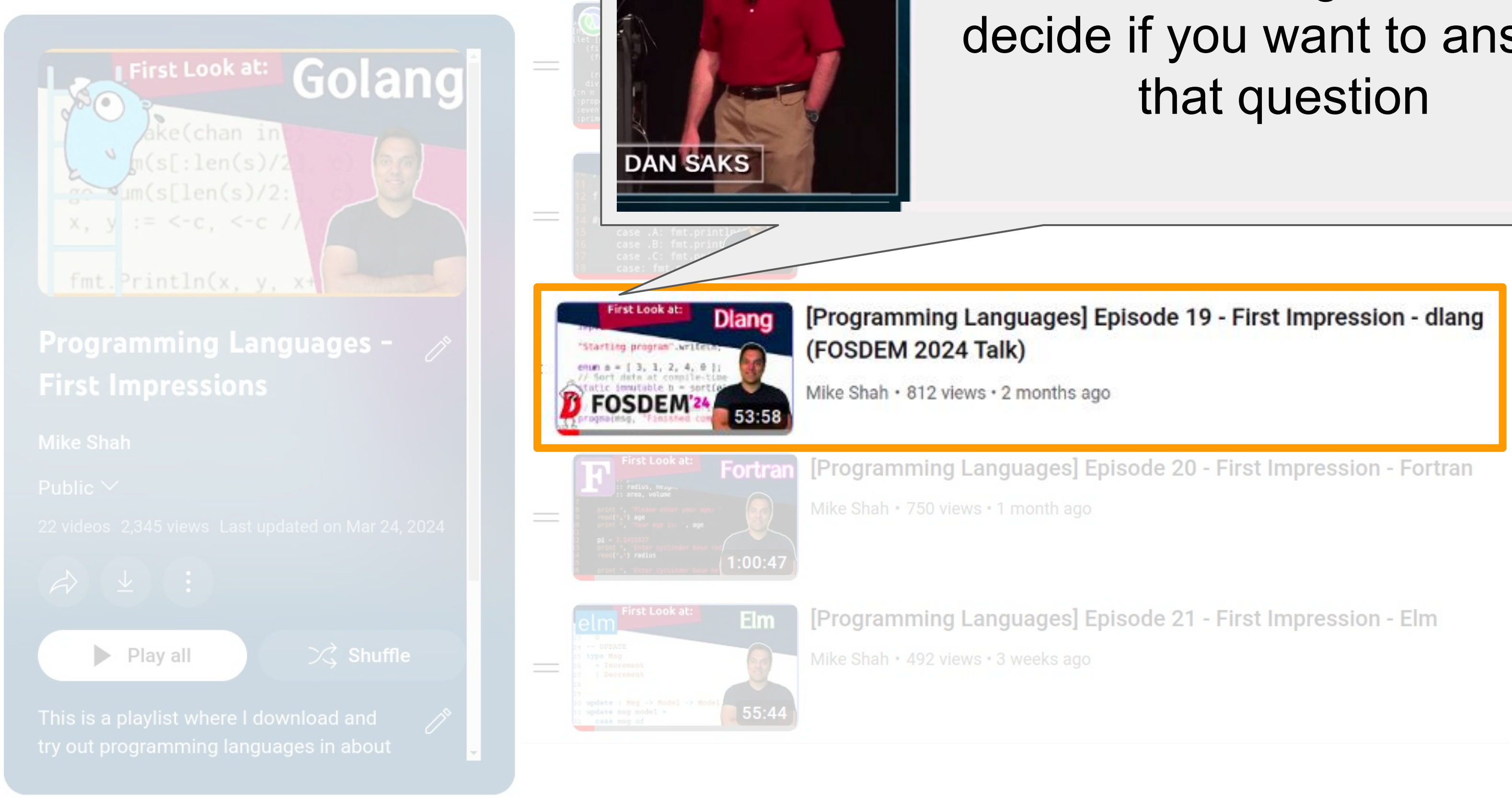
If You Remember Just One Thing...

- When it comes to persuasion...

"If you're arguing, you're losing."

CppCon 2016: Dan Saks "extern c: Talking to C Programmers about C++"
https://youtu.be/D7Sd8A6_fYU?si=wstjG6vkSEZ4345i&t=1320
(Dan Saks quoting Mike Thomas)

The Art of Persuasion... (3/5)


Programming Languages - First Impressions

Mike Shah

Public

22 videos 2,345 views Last updated on Mar 24, 2024

This is a playlist where I download and try out programming languages in about

[Programming Languages] Episode 19 - First Impression - dlang (FOSDEM 2024 Talk)

Mike Shah • 812 views • 2 months ago

[Programming Languages] Episode 20 - First Impression - Fortran

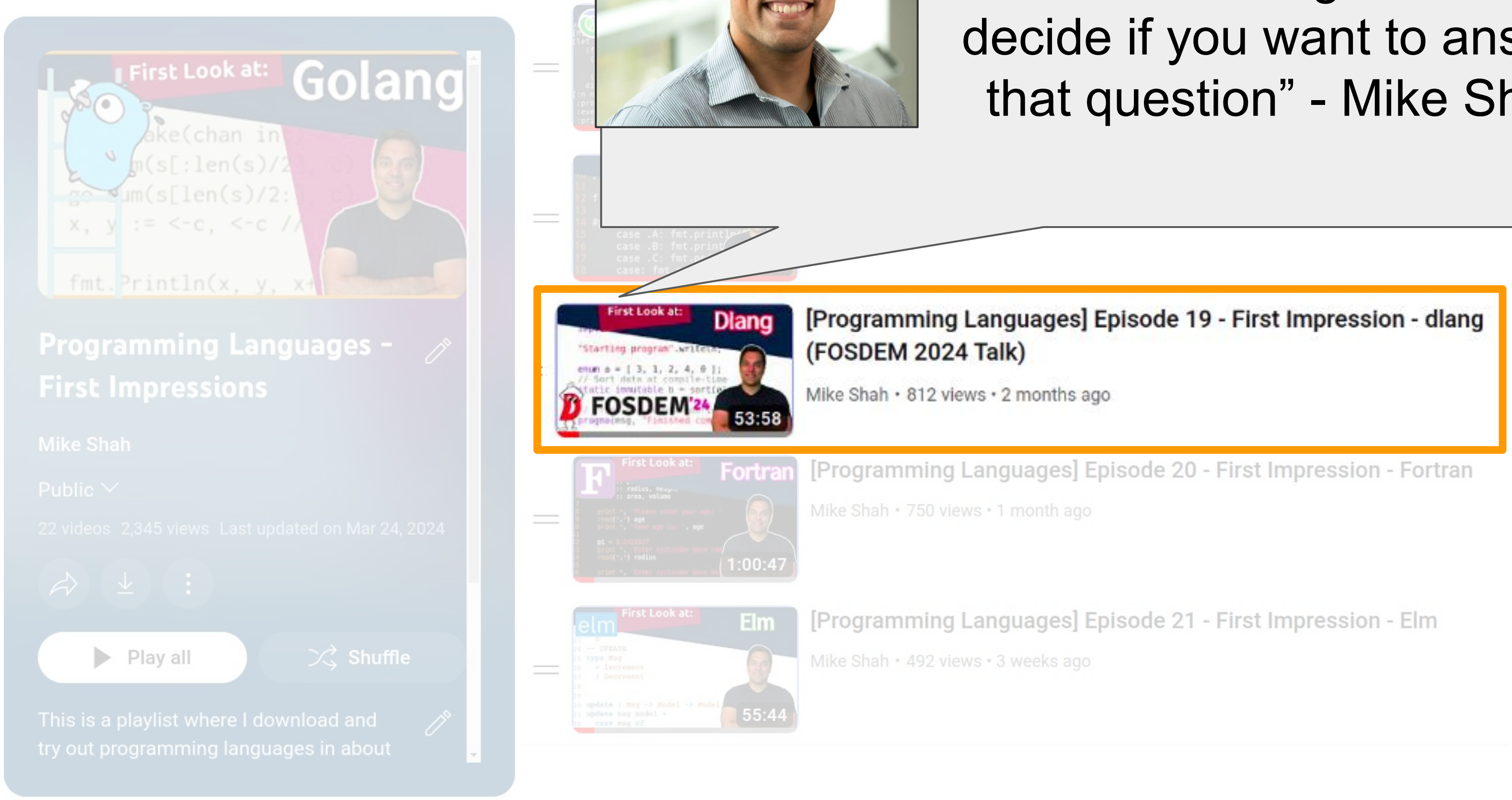
Mike Shah • 750 views • 1 month ago

[Programming Languages] Episode 21 - First Impression - Elm

Mike Shah • 492 views • 3 weeks ago

You're going to have to be motivated enough to want to decide if you want to answer that question

The Art of Persuasion... (4/5)


Programming Languages - First Impressions

Mike Shah

Public

22 videos 2,345 views Last updated on Mar 24, 2024

This is a playlist where I download and try out programming languages in about

[Programming Languages] Episode 19 - First Impression - dlang (FOSDEM 2024 Talk)

Mike Shah • 812 views • 2 months ago

[Programming Languages] Episode 20 - First Impression - Fortran

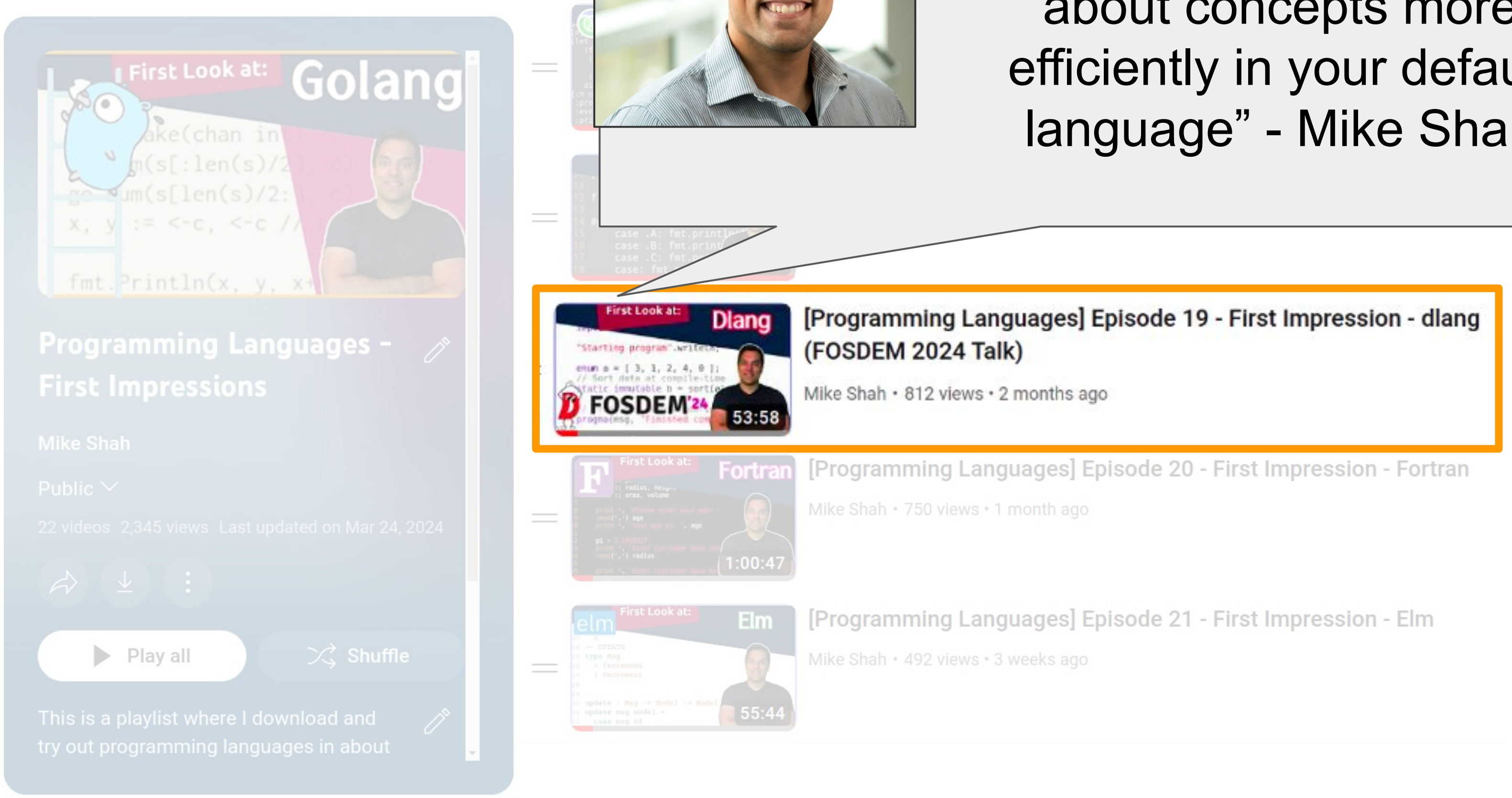
Mike Shah • 750 views • 1 month ago

[Programming Languages] Episode 21 - First Impression - Elm

Mike Shah • 492 views • 3 weeks ago

"You're going to have to be motivated enough to want to decide if you want to answer that question" - Mike Shah

The Art of Persuasion... (5/5)



Programming Languages - First Impressions

Mike Shah

Public

22 videos 2,345 views Last updated on Mar 24, 2024

Play all

Shuffle

This is a playlist where I download and try out programming languages in about

[Programming Languages] Episode 19 - First Impression - dlang (FOSDEM 2024 Talk)

Mike Shah • 812 views • 2 months ago

[Programming Languages] Episode 20 - First Impression - Fortran

Mike Shah • 750 views • 1 month ago

[Programming Languages] Episode 21 - First Impression - Elm

Mike Shah • 492 views • 3 weeks ago

"I personally think learning new languages help you think about concepts more efficiently in your default language" - Mike Shah

Active poll

6

Join at
slido.com
#2649 488

Can you recall one concept you learned in another language that you have applied to another language?

Anonymous
List comprehensions

Anonymous
Async Await

Anonymous
Scope(exit)

Anonymous
ML

Anonymous
The ideas behind functional programming

Anonymous
Functional Programming ideas

Anonymous
Monads

Anonymous
Rust ownership

Goals (1/2)

Programming Languages - First Impressions

Mike Shah

Public

22 videos 2,345 views Last updated on Mar 24, 2024

Play all

Shuffle

This is a playlist where I download and try out programming languages in about

[Programming Languages] Episode 19 - First Impression - dlang (FOSDEM 2024 Talk)

Mike Shah • 812 views • 2 months ago

[Programming Languages] Episode 20 - First Impression - Fortran

Mike Shah • 750 views • 1 month ago

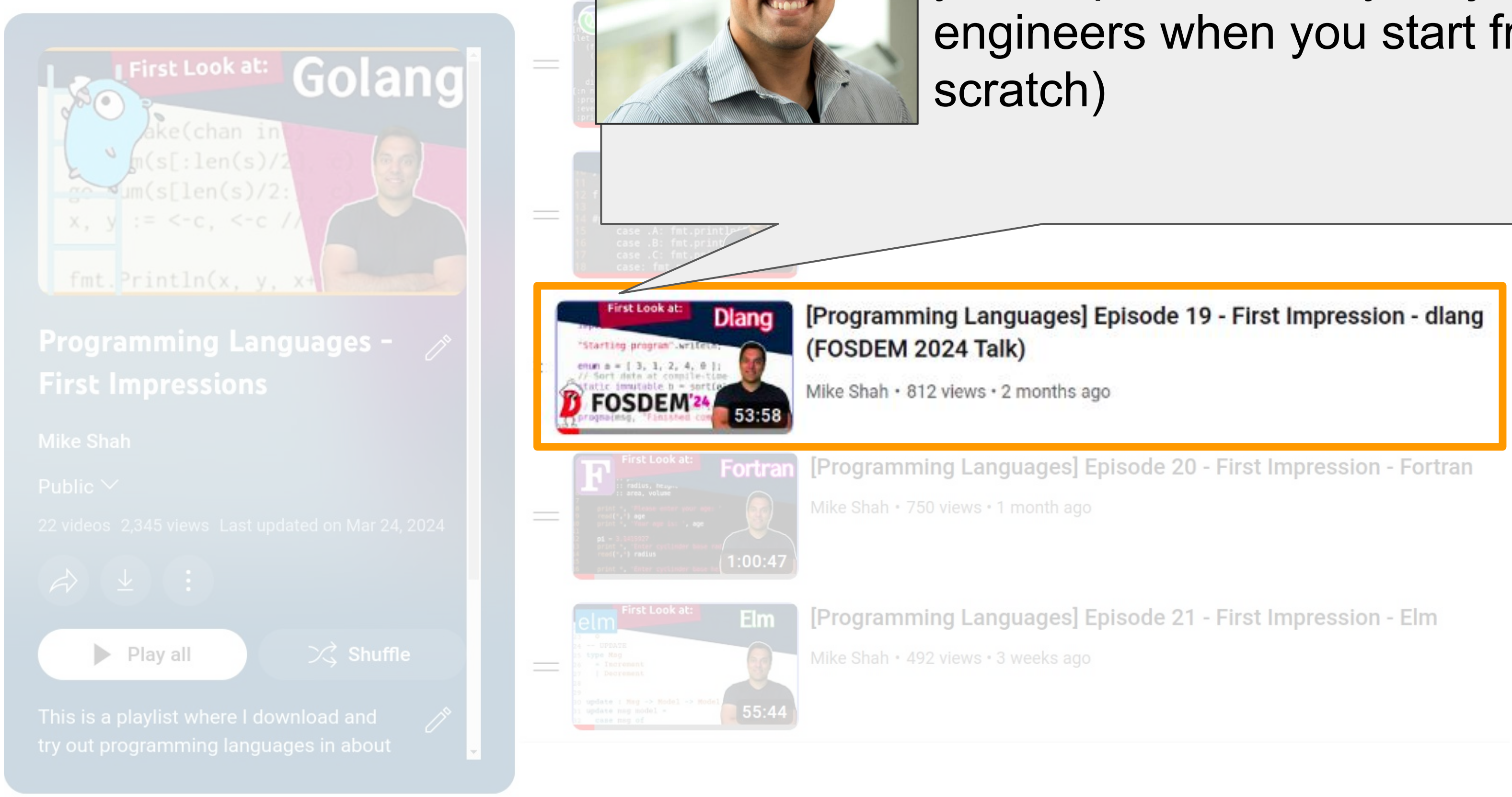
[Programming Languages] Episode 21 - First Impression - Elm

Mike Shah • 492 views • 3 weeks ago

But...If you remember just one thing after this talk:

1. **Set a timer for one hour**
2. **Go to https://tour.dlang.org/**
3. **Try out the D Language**

Goals (2/2)



Programming Languages - First Impressions

Mike Shah

Public

22 videos 2,345 views Last updated on Mar 24, 2024

This is a playlist where I download and try out programming languages in about

[Programming Languages] Episode 19 - First Impression - dlang (FOSDEM 2024 Talk)

Mike Shah • 812 views • 2 months ago

[Programming Languages] Episode 20 - First Impression - Fortran

Mike Shah • 750 views • 1 month ago

[Programming Languages] Episode 21 - First Impression - Elm

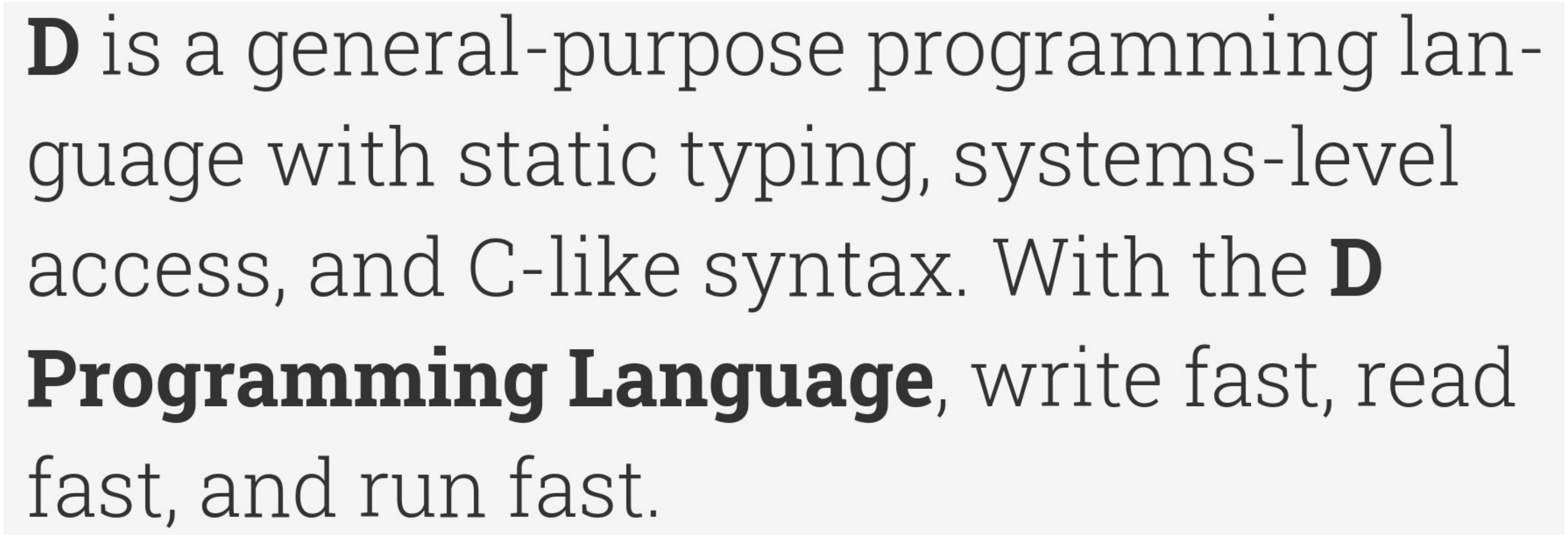
Mike Shah • 492 views • 3 weeks ago

(At the very least, it might help you empathize with your junior engineers when you start from scratch)

The D Programming Language
(Le langage de programmation D)

So what is the D Programming Language? (1/2)

So what is the D Programming Language? (2/2)

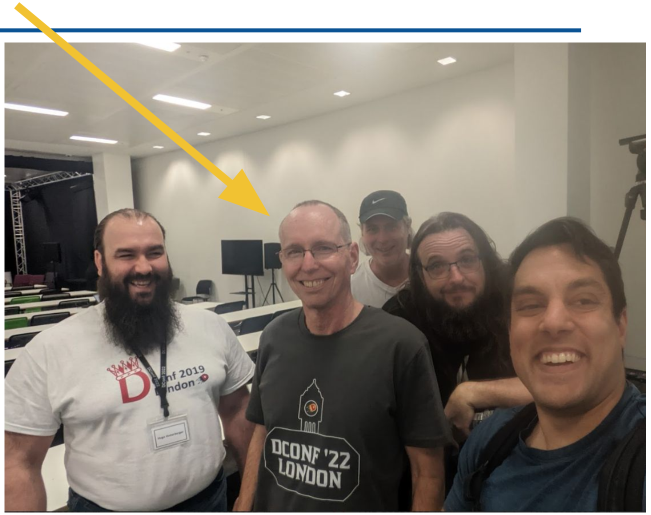


D is a general-purpose programming language with static typing, systems-level access, and C-like syntax. With the **D Programming Language**, write fast, read fast, and run fast.

https://dlang.org/

D Language History - Created by Walter Bright [wiki]

- Walter Bright
  - Wrote a C Compiler (Datalight C compiler)
  - Famously created the Zortech C++ compiler
  - Designed the game Empire
    - (There is even a translation of Empire to D!)
  - Between 1999-2006 worked alone on D version 1 programming language.
    - (Originally it was the Digital Mars Compiler, but everyone colleagues and friends insisted on calling it the next evolution to C++, thus the name 'D')
- Around 2006 or 2007 -- D2 would start being developed with Andrei Alexandrescu and others.
  - Full history here - Origins of the D Programming Language
    - https://dl.acm.org/doi/pdf/10.1145/3386323



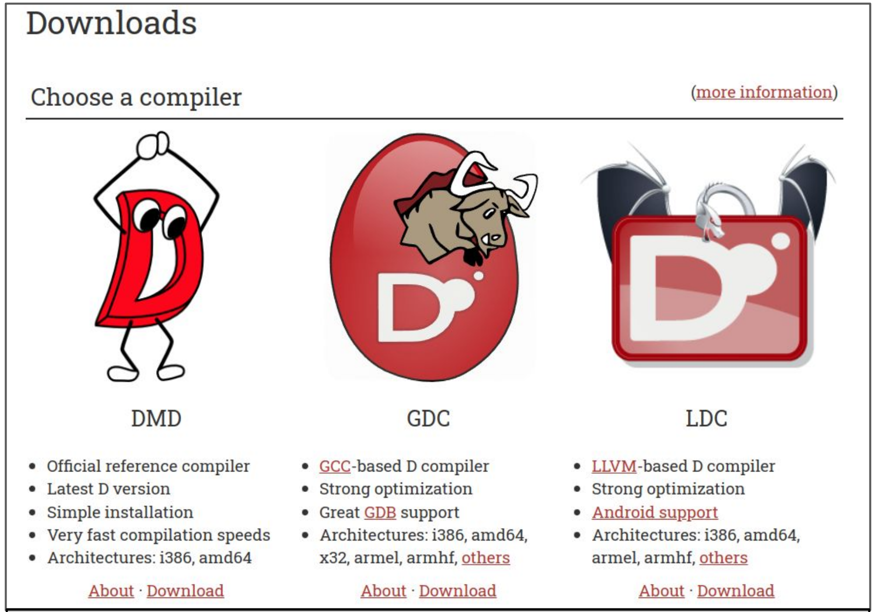
Dconf 2022 in London

D hosts an online and in-person conference every year: https://dconf.org/

So, over the last 25 years -- now three D Compilers!

- DMD is the official reference compiler
  - The compiler **is open-source** and you can fork a copy of it today
  - DMD is a **very fast compiler** (in part because of D's module system)
- GDC
  - GCC-based D Compiler Frontend
  - Good GDB support
- LDC - LLVM based D Compiler
  - Allows you to get LLVM optimizations and target many architectures

Note: Common for D programmers to develop in DMD for quick edit-compile-run cycles, and then deploy using GDC or LDC



Downloads

Choose a compiler (more information)

DMD
- Official reference compiler
- Latest D version
- Simple installation
- Very fast compilation speeds
- Architectures: i386, amd64

About · Download

GDC
- GCC-based D compiler
- Strong optimization
- Great GDB support
- Architectures: i386, amd64, x32, armel, armhf, others

About · Download

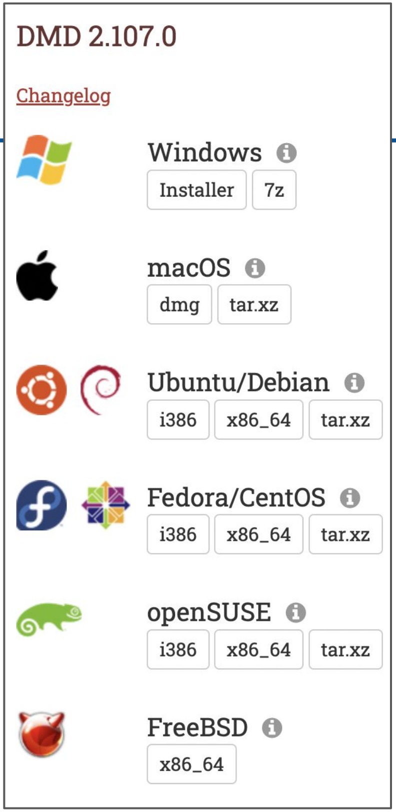
LDC
- LLVM-based D compiler
- Strong optimization
- Android support
- Architectures: i386, amd64, armel, armhf, others

About · Download

https://dlang.org/download.html

Downloading the Tools

- The download of any of the compilers is relatively simple and available for many architectures from the homepage
  - Along with the download, you also get:
    - **Dub** - the package manager for managing dependencies and as a lightweight build tool.
    - Other useful tools like **dfmt** (a code formatter) and **dscanner** (a linter) exist
    - A VSCode extension (**code-d)** is available, as well as some support in IntelliJ for D.


DMD 2.107.0

Changelog

Windows: Installer | 7z

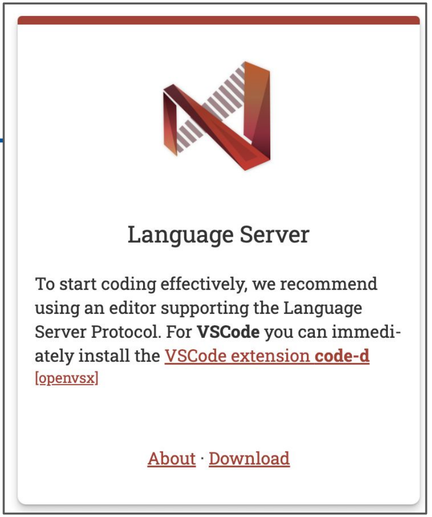
macOS: dmg | tar.xz

Ubuntu/Debian: i386 | x86_64 | tar.xz

Fedora/CentOS: i386 | x86_64 | tar.xz

openSUSE: i386 | x86_64 | tar.xz

FreeBSD: x86_64


Language Server

To start coding effectively, we recommend using an editor supporting the Language Server Protocol. For **VSCode** you can immediately install the VSCode extension **code-d** [openvsx]

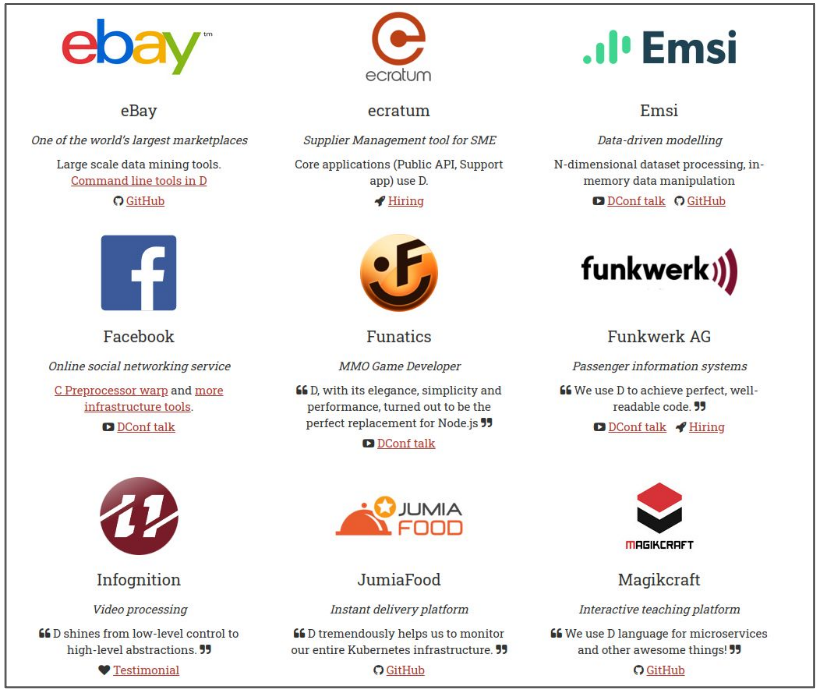
About · Download

https://dlang.org/download.html

Note: Brian Callahan gets a lot of credit for bringing D to OpenBSD https://briancallahan.net/blog/20211013.html

DLang Domains

- DLang is a general purpose systems programming language
  - D can be used in any domain.
- Dlang has found **some niches in performance-based** domains:
  - e.g. image processing, gaming, streaming, finance, and simulation



eBay
One of the world's largest marketplaces
Large scale data mining tools. Command line tools in D
GitHub

ecratum
Supplier Management tool for SME
Core applications (Public API, Support app) use D.
Hiring

Emsi
Data-driven modelling
N-dimensional dataset processing, in-memory data manipulation
DConf talk GitHub

Facebook
Online social networking service
C Preprocessor warp and more infrastructure tools.
DConf talk

Funatics
MMO Game Developer
“D, with its elegance, simplicity and performance, turned out to be the perfect replacement for Node.js”
DConf talk

Funkwerk AG
Passenger information systems
“We use D to achieve perfect, well-readable code.”
DConf talk Hiring

Infognition
Video processing
“D shines from low-level control to high-level abstractions.”
Testimonial

JumiaFood
Instant delivery platform
“D tremendously helps us to monitor our entire Kubernetes infrastructure.”
GitHub

Magikcraft
Interactive teaching platform
“We use D language for microservices and other awesome things!”
GitHub

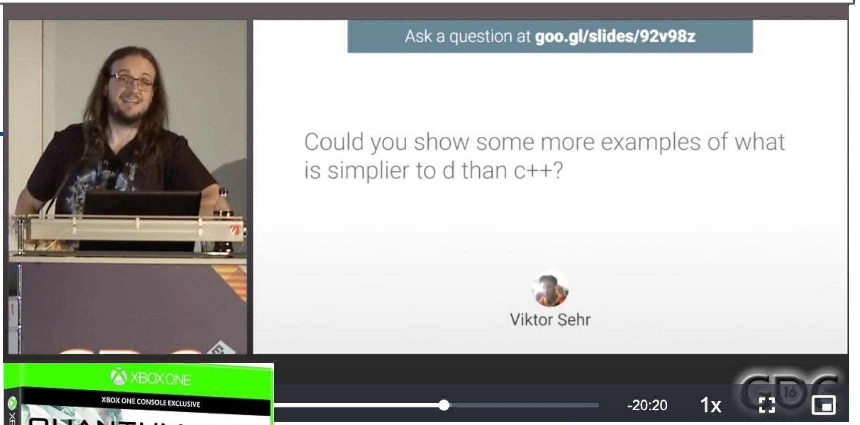
https://dlang.org/orgs-using-d.html

Utilized the **D Programming Language**

Quantum Break -- Game

AAA Game Projects in D

- It’s also worth noting that D has been used in AAA Commercial Games
  - Ethan Watson has a wonderful presentation describing that experience
  - Link to talk: https://www.gdcvault.com/play/1023843/D-Using-an-Emerging-Language
- Talk Abstract: *Can you use D to make games? Yes. Has it been used in a major release? It has now. But what benefits does it have over C++? Is it ready for mass use? Does treating code as data with a traditional C++ engine work? This talk will cover Remedy's usage of the D programming language in Quantum Break and also provide some details on where we want to take usage of it in the future.*


Ask a question at **goo.gl/slides/92v98z**

Could you show some more examples of what is simplier to d than c++?

Viktor Sehr

https://m.media-amazon.com/images/M/MV5BOThjOWRhN2QtYmIxMy00MGE3LTk5ZWMtY2ZkMzI0MGY1ZTM1XkEyXkFqcGdeQXVyMTYxMzY1ODg@._V1_.jpg

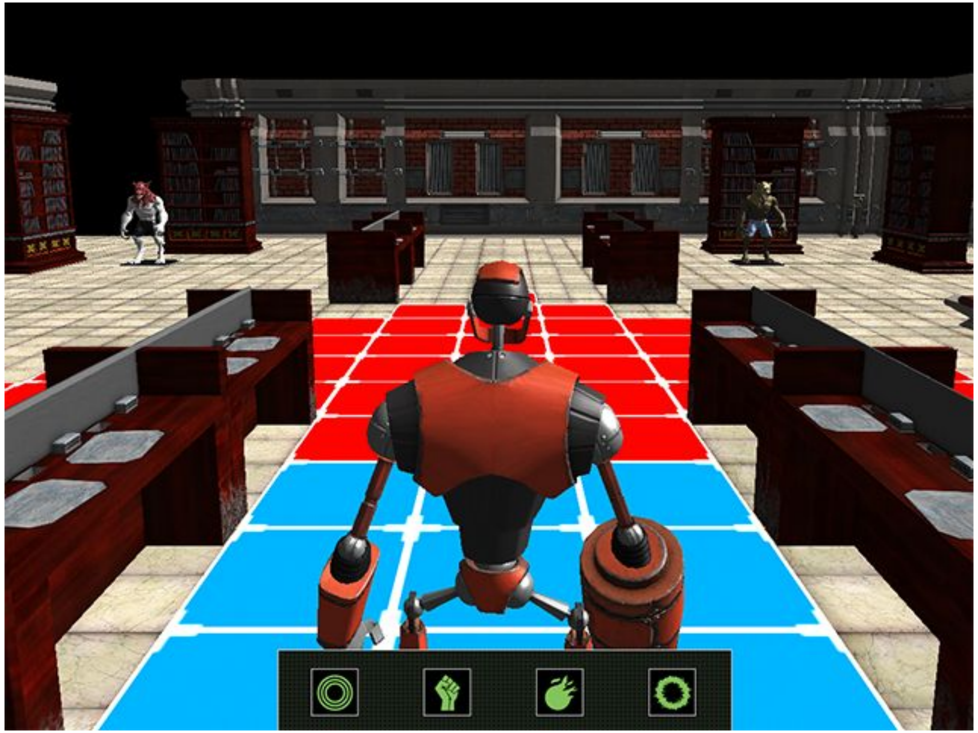
Built in the D Programming Language

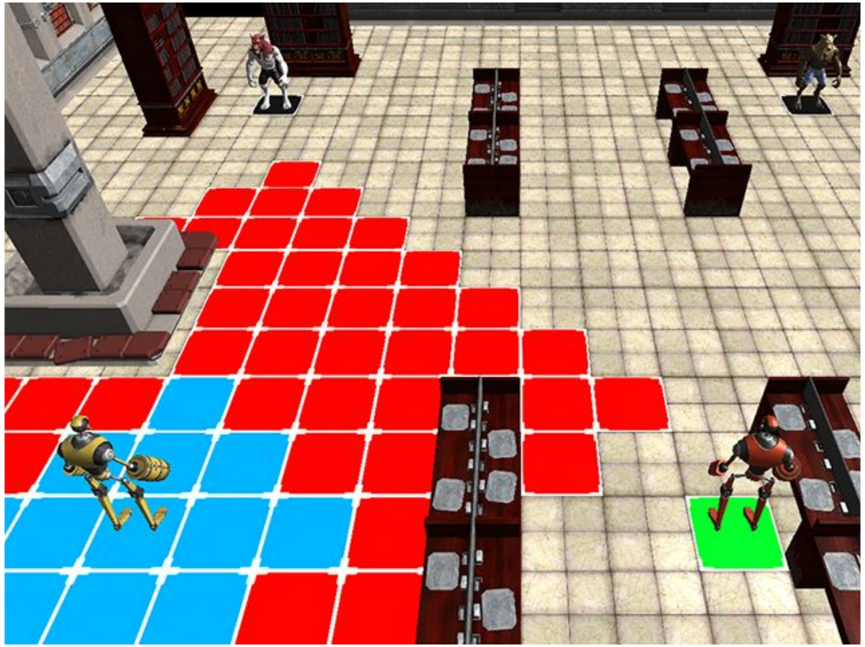
Dagon -- Game Engine

- Website with games and tutorials: https://gecko0307.github.io/dagon/
- Github or Dub Repository: https://github.com/gecko0307/dagon | https://code.dlang.org/packages/dagon

Built in the **D Programming Language**

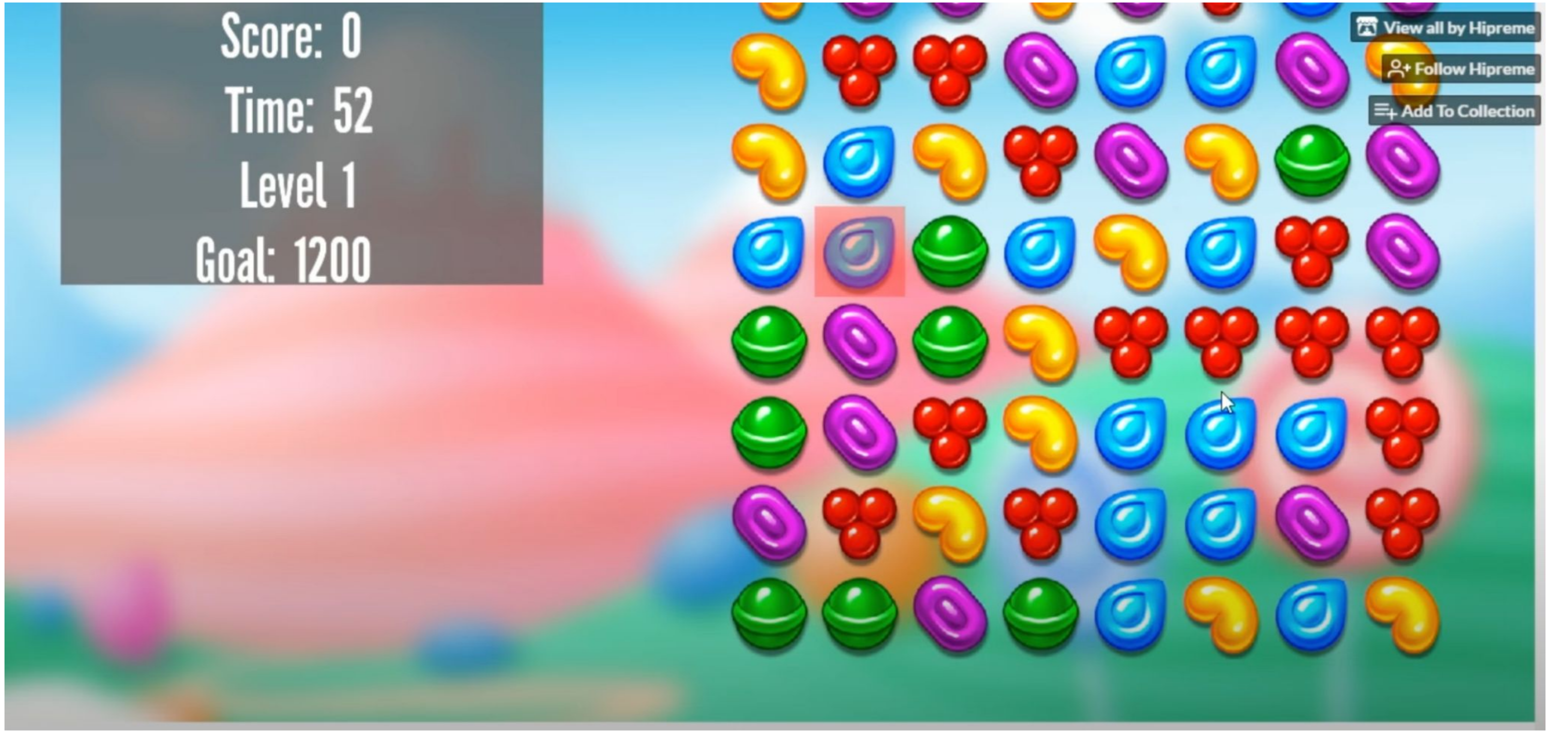
Dash -- Game Engine





- Website with games: https://circularstudios.com/
- Github or Dub Repository: https://github.com/Circular-Studios/Dash
- Forum Post: https://forum.dlang.org/thread/qnaqymkehjvopwxwvwig@forum.dlang.org

Built in the D Programming Language | Hipreme Engine -- Game Engine



- Github or Dub Repository: https://github.com/MrcSnm/HipremeEngine
- DConf 2023 Talk: DConf '23 -- Hipreme Engine: Bringing D Everywhere -- Marcelo Mancini

Built in the D Programming Language | **Eilmer**(/ɛlmə/) Compressible Flow Simulator

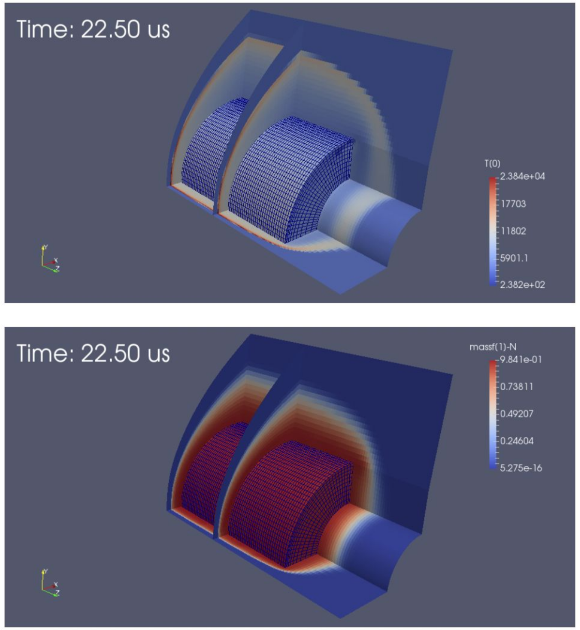
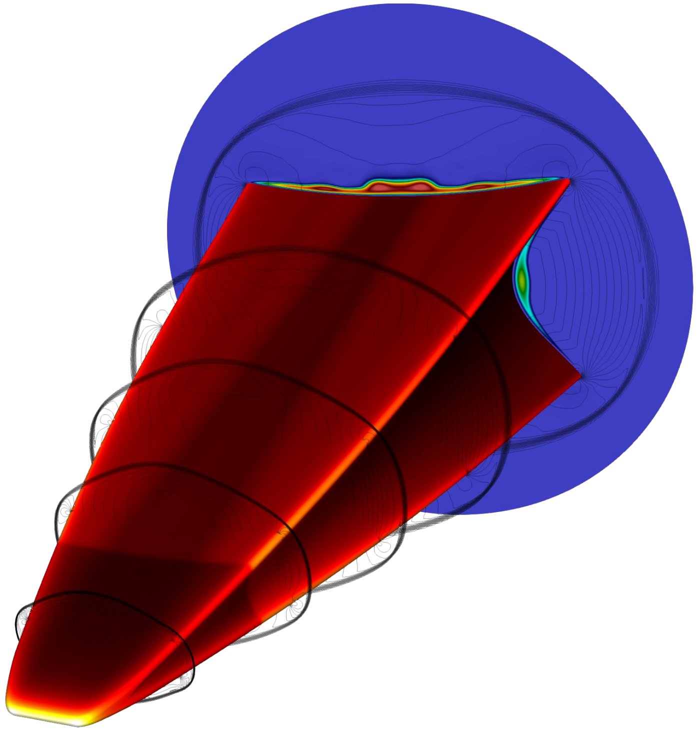


Figure 5.7: Static temperature and mass fraction of nitrogen atoms in the flow field from the chemical nonequilibrium simulation.



- Website: https://gdtk.uqcloud.net/ and https://gdtk.uqcloud.net/pdfs/eilmer-user-guide.pdf
- Github or Dub Repository: https://github.com/gdtk-uq/gdtk

RayTracer

- So here was a Raytracer that I built-in the D programming language
    - An obvious candidate for parallelism from the std.parallelism module

- Talk/Website: https://www.youtube.com/watch?v=nCIB8df7q2g (Dconf 22)
    - Follow on talk: https://www.youtube.com/watch?v=MFhTRiobWfU (Dconf Online 22)
- Github or Dub Repository: https://github.com/MikeShah/Talks/tree/main/2022/2022_dconf_London

(Aside)

- More on the previous open-source projects and the source code
- https://www.youtube.com/watch?v=yLaUsmLr9so

```
"Starting program".writeln;

enum a = [ 3, 1, 2, 4, 0 ];
// Sort data at compile-time
static immutable b = sort(a);

pragma(msg, "Finished com
```

[Programming Languages] Episode 19 - First Impression - dlang (FOSDEM 2024 Talk)

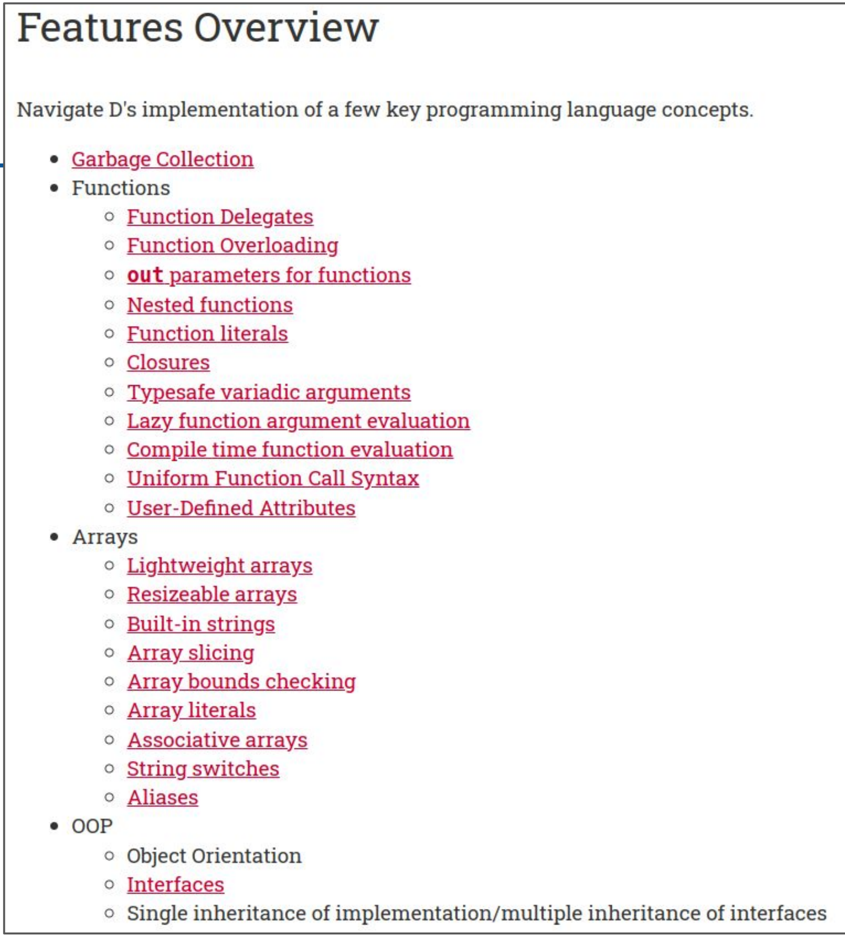
815 views • 2 months ago

Mike Shah

▻Lesson Description: In this lesson I present one of my favorite languages -- in fact I'm breaking the rules a bit -- **dlang**! As many ...

DLang Features

- We've seen **compile-time function execution** (ctfe) as one modern feature of the D language compiler (at the very start of the talk)
- The language itself supports many nice quality of life features for safety and productivity -- for example:
  - Built-in dynamic arrays
  - Built-in Associative arrays (i.e. map/hashtable/dictionary)
  - Bounds checked arrays
    - (With ability to disable if needed)
  - lambda's and delegates
  - Uniform Function Call Syntax (UFCS)
  - Object-Oriented Programming Paradigm
  - Functional paradigms (lazy evaluation, pure functions)
  - Concurrency
  - Garbage Collection or manual memory management options
    - i.e. You can just use malloc/free if you really want!
  - and more!



Features Overview

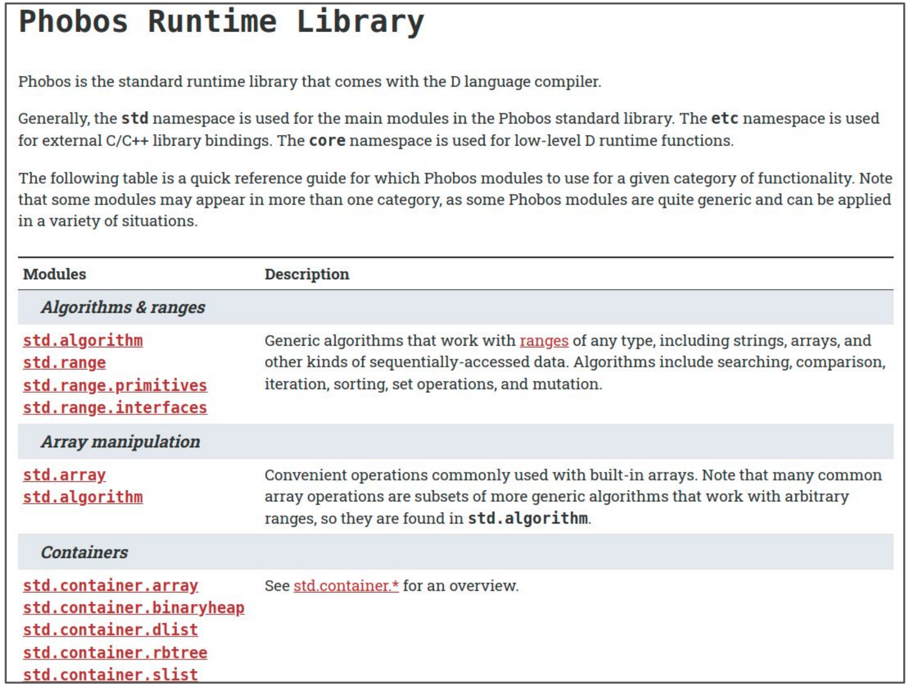
Navigate D's implementation of a few key programming language concepts.

- Garbage Collection
- Functions
  - Function Delegates
  - Function Overloading
  - **out** parameters for functions
  - Nested functions
  - Function literals
  - Closures
  - Typesafe variadic arguments
  - Lazy function argument evaluation
  - Compile time function evaluation
  - Uniform Function Call Syntax
  - User-Defined Attributes
- Arrays
  - Lightweight arrays
  - Resizeable arrays
  - Built-in strings
  - Array slicing
  - Array bounds checking
  - Array literals
  - Associative arrays
  - String switches
  - Aliases
- OOP
  - Object Orientation
  - Interfaces
  - Single inheritance of implementation/multiple inheritance of interfaces

https://dlang.org/comparison.html

Phobos The Standard Runtime Library

- Phobos is the **standard runtime library** that comes with D.
  - Thus, I like to think of D as a ‘batteries included’ language
  - You can get started immediately and be productive and writing software to solve problems.
    - Phobos comes ready with a rich set of algorithms, containers (data structures), and other common libraries for solving problems.
      - “Containers” are the standard libraries **data structures** (beyond the built-in types) that describe how we access and store data.
      - And the “**algorithms**” and “**ranges**” and are building blocks for computation
- The Standard Library (std) has common data structures and ability to work with data (json, csv, xml), compression (zip), networking (sockets, curl), etc.



Phobos Runtime Library

Phobos is the standard runtime library that comes with the D language compiler.

Generally, the `std` namespace is used for the main modules in the Phobos standard library. The `etc` namespace is used for external C/C++ library bindings. The `core` namespace is used for low-level D runtime functions.

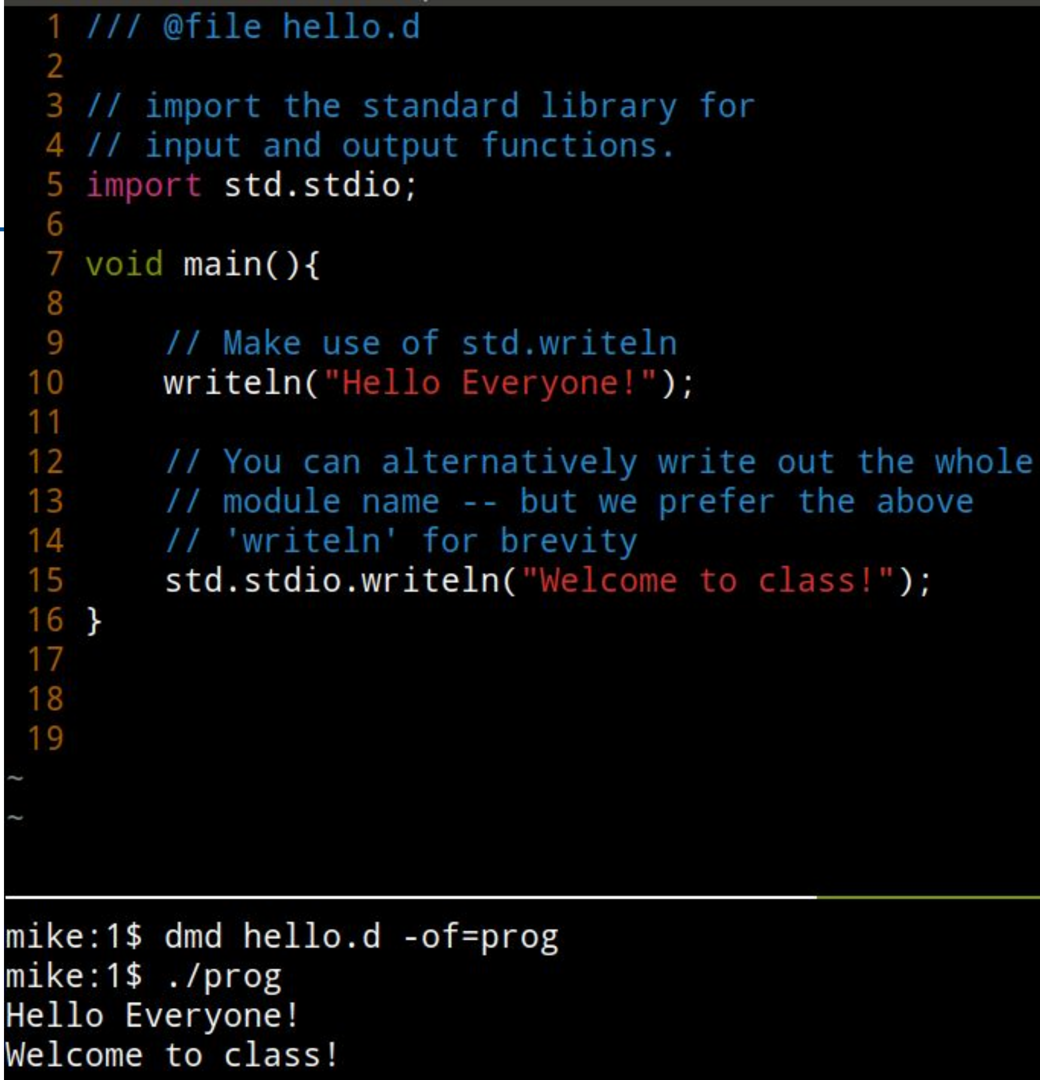
The following table is a quick reference guide for which Phobos modules to use for a given category of functionality. Note that some modules may appear in more than one category, as some Phobos modules are quite generic and can be applied in a variety of situations.

Modules	Description
Algorithms & ranges	
`std.algorithm` `std.range` `std.range.primitives` `std.range.interfaces`	Generic algorithms that work with ranges of any type, including strings, arrays, and other kinds of sequentially-accessed data. Algorithms include searching, comparison, iteration, sorting, set operations, and mutation.
Array manipulation	
`std.array` `std.algorithm`	Convenient operations commonly used with built-in arrays. Note that many common array operations are subsets of more generic algorithms that work with arbitrary ranges, so they are found in `std.algorithm`.
Containers	
`std.container.array` `std.container.binaryheap` `std.container.dlist` `std.container.rbtree` `std.container.slist`	See std.container.* for an overview.

https://dlang.org/phobos/index.html

Hello DLang (1/2)

- Here it is, the “Hello World” program in D
    - You’ll see some familiar constructs in other languages
    - **import** brings in a ‘module’ a library of code.
        - prefixed with the ‘std’ means this comes from the standard library
        - The standard library is called ‘phobos’ in D.
    - You’ll see ‘writeln’ for writing out text (or we can use the fully qualified name std.stdio.writeln)

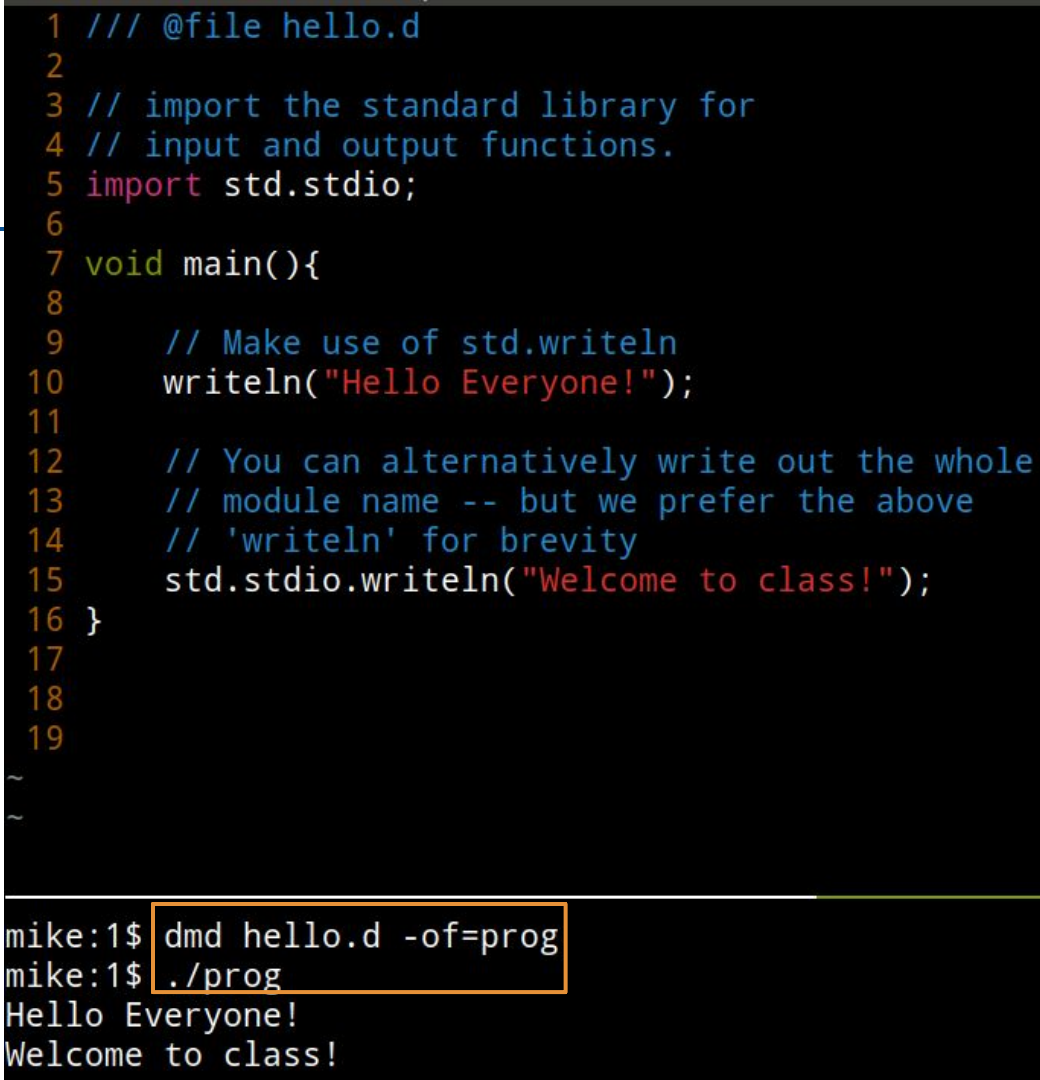


```d
/// @file hello.d

// import the standard library for
// input and output functions.
import std.stdio;

void main(){

    // Make use of std.writeln
    writeln("Hello Everyone!");

    // You can alternatively write out the whole
    // module name -- but we prefer the above
    // 'writeln' for brevity
    std.stdio.writeln("Welcome to class!");
}



~
~
```

```
mike:1$ dmd hello.d -of=prog
mike:1$ ./prog
Hello Everyone!
Welcome to class!
```

Hello DLang (2/2)

- Observe at the bottom, that D is a compiled language
  - This means we need to invoke the dmd compiler
    - **dmd** - the compiler
    - **hello.d** - source file we want to compile
    - **-of=prog** - Tells us that we want the output binary to be named ‘prog’
    - **./prog** - runs our executable (note this may be prog.exe on windows)

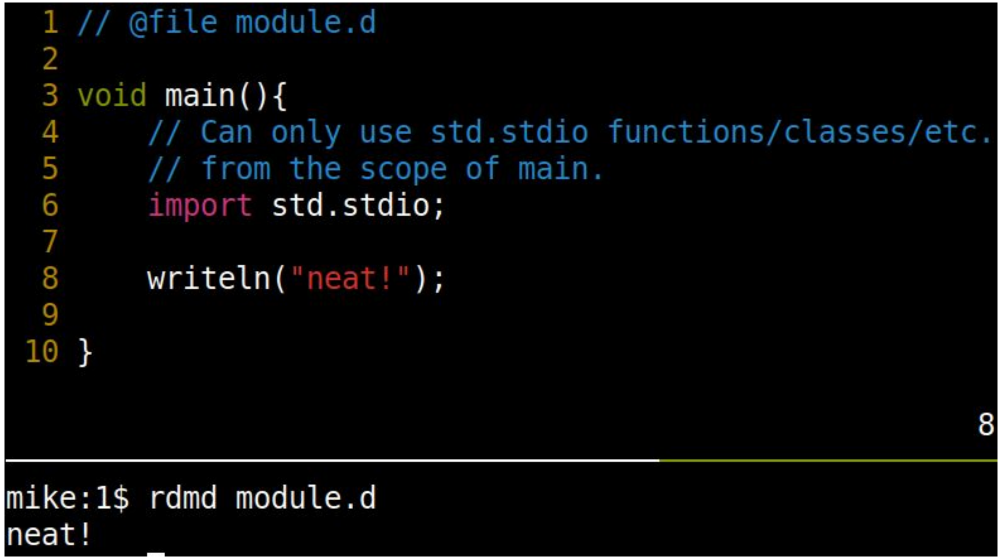


```d
/// @file hello.d

// import the standard library for
// input and output functions.
import std.stdio;

void main(){

    // Make use of std.writeln
    writeln("Hello Everyone!");

    // You can alternatively write out the whole
    // module name -- but we prefer the above
    // 'writeln' for brevity
    std.stdio.writeln("Welcome to class!");
}



~
~
```

```
mike:1$ dmd hello.d -of=prog
mike:1$ ./prog
Hello Everyone!
Welcome to class!
```

Note: on modules

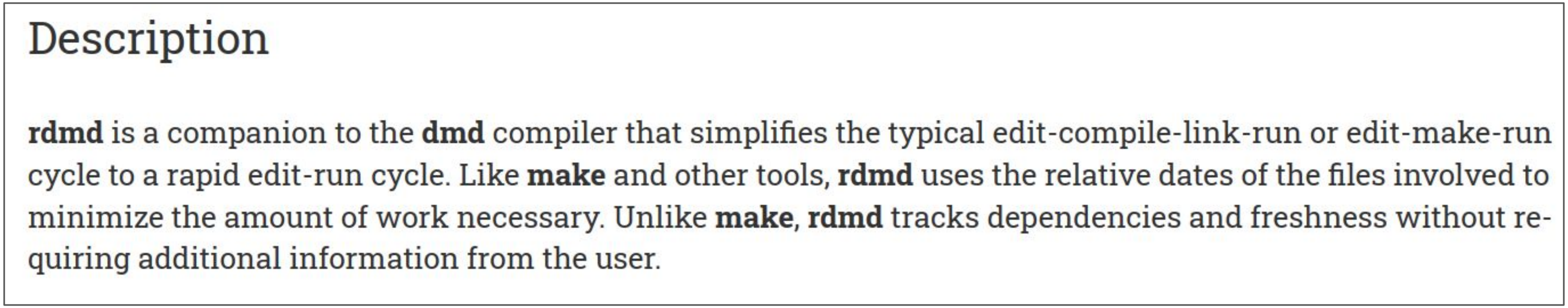
- imports do not have to have a global scope, often times we'll prefer to have a scope local to a function.
- More info:
  - https://tour.dlang.org/tour/en/basics/imports-and-modules



```
// @file module.d

void main(){
    // Can only use std.stdio functions/classes/etc.
    // from the scope of main.
    import std.stdio;

    writeln("neat!");

}
                                                              8
```

```
mike:1$ rdmd module.d
neat!
```

rdmd

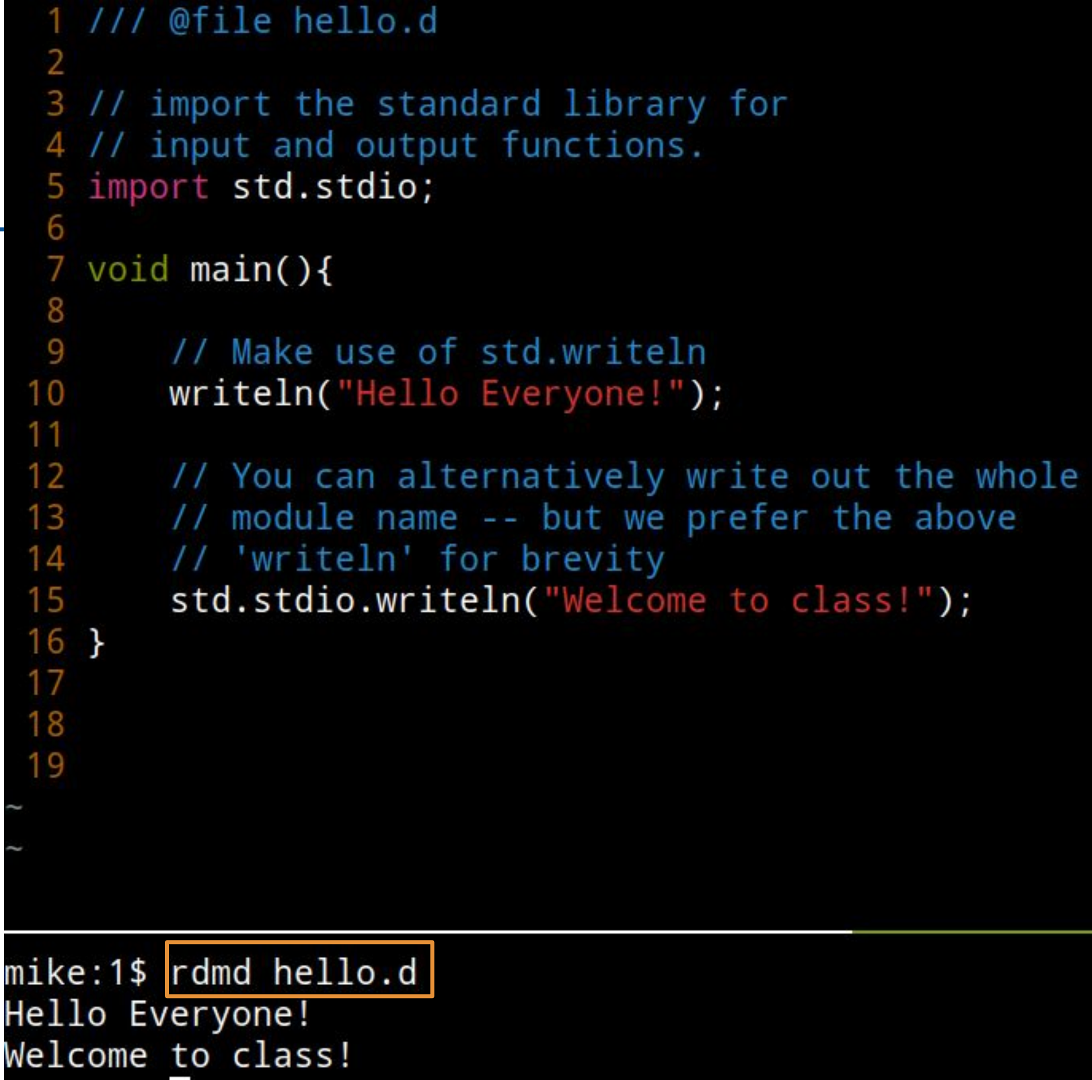


Description

rdmd is a companion to the **dmd** compiler that simplifies the typical edit-compile-link-run or edit-make-run cycle to a rapid edit-run cycle. Like **make** and other tools, **rdmd** uses the relative dates of the files involved to minimize the amount of work necessary. Unlike **make**, **rdmd** tracks dependencies and freshness without requiring additional information from the user.

rdmd introduction

- Now I’m going to re-run the hello.d program again
  - This time with a ‘shortcut’, the rdmd
  - This allows me to speed up my edit-compile-run cycle
    - rdmd is a smart tool to help us iterate more quickly when writing D code
- (Note: You can also use: ‘dmd -run hello.d)

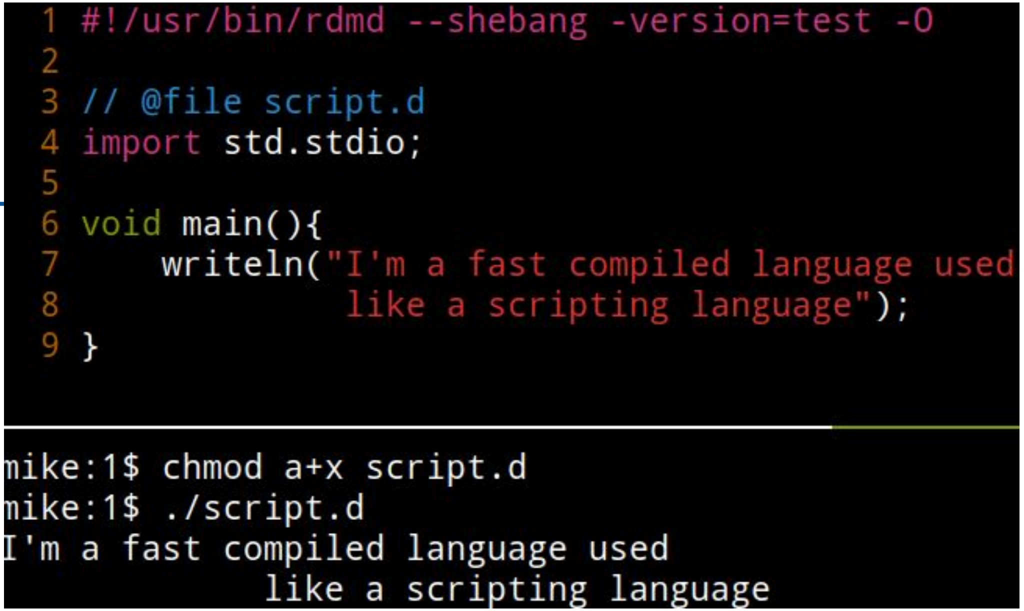


```d
/// @file hello.d

// import the standard library for
// input and output functions.
import std.stdio;

void main(){

    // Make use of std.writeln
    writeln("Hello Everyone!");

    // You can alternatively write out the whole
    // module name -- but we prefer the above
    // 'writeln' for brevity
    std.stdio.writeln("Welcome to class!");
}
```

```
mike:1$ rdmd hello.d
Hello Everyone!
Welcome to class!
```

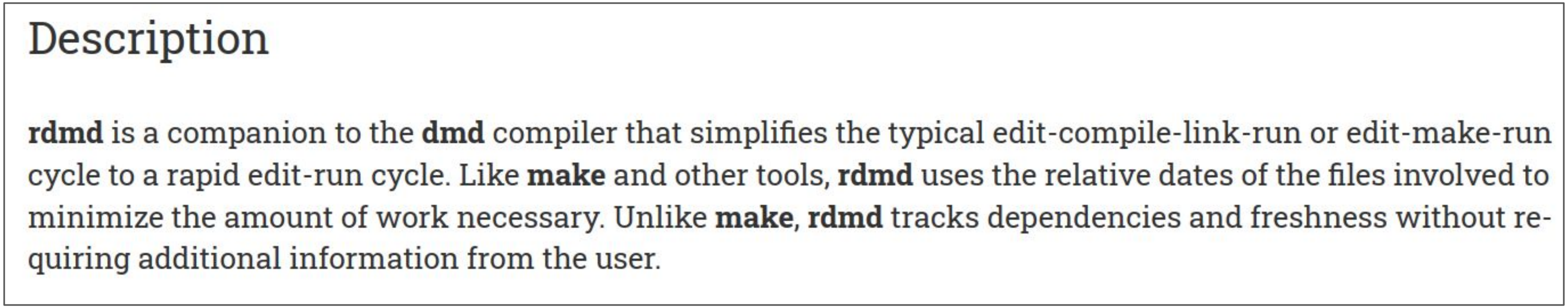
rdmd scripts

- You can check out more here: https://dlang.org/rdmd.html
  - Having the rdmd tool allows us to essentially use the D compiler like a scripting language
    - See example to the right



```
#!/usr/bin/rdmd --shebang -version=test -O

// @file script.d
import std.stdio;

void main(){
    writeln("I'm a fast compiled language used
                like a scripting language");
}
```

```
mike:1$ chmod a+x script.d
mike:1$ ./script.d
I'm a fast compiled language used
            like a scripting language
```

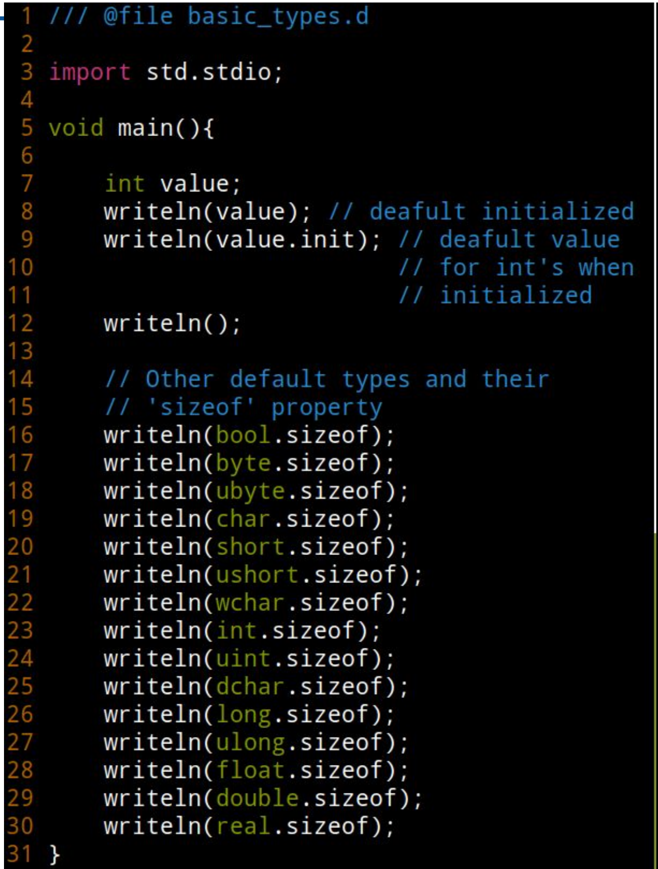


Description

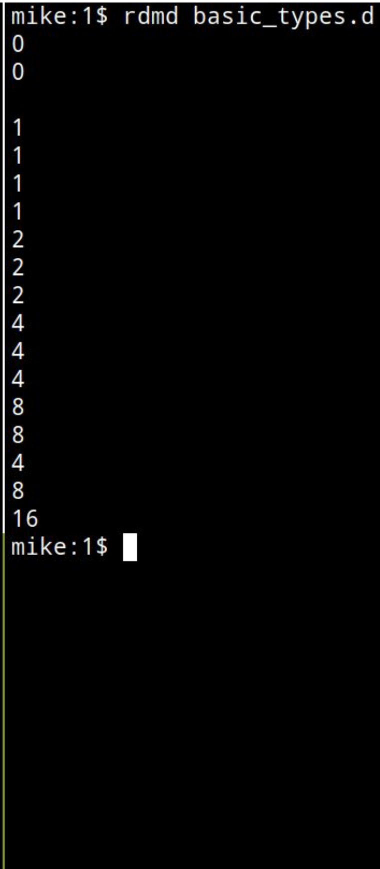
rdmd is a companion to the **dmd** compiler that simplifies the typical edit-compile-link-run or edit-make-run cycle to a rapid edit-run cycle. Like **make** and other tools, **rdmd** uses the relative dates of the files involved to minimize the amount of work necessary. Unlike **make**, **rdmd** tracks dependencies and freshness without requiring additional information from the user.

Basic Types - D gets the Defaults Right (1/2)

- The size of variables is fixed regardless of platform.
  - e.g. An int is always 4 bytes
- Variables are default initialized
  - In some languages (e.g. C or C++) variables must be explicitly initialized
  - D initializes everything
    - (You can explicitly leave something uninitialized with =void however if you truly do not want say a large buffer of data to be zero initialized)



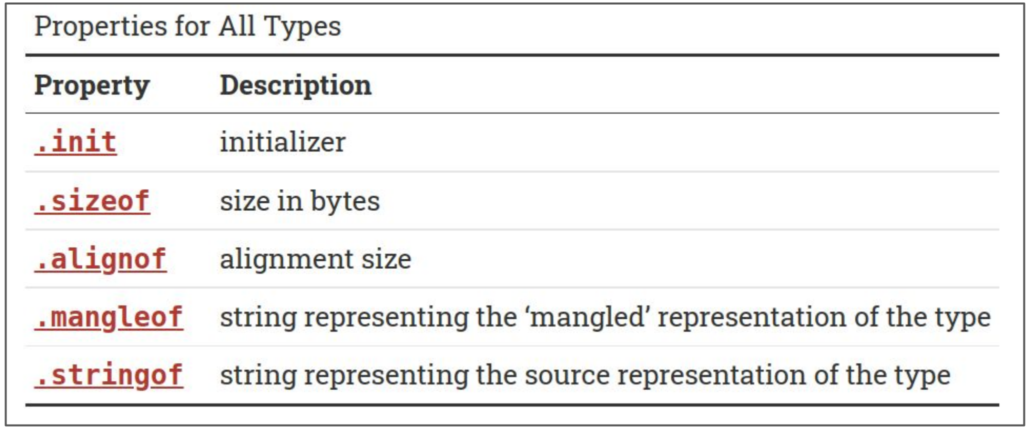
```d
/// @file basic_types.d

import std.stdio;

void main(){

    int value;
    writeln(value); // deafult initialized
    writeln(value.init); // deafult value
                         // for int's when
                         // initialized
    writeln();

    // Other default types and their
    // 'sizeof' property
    writeln(bool.sizeof);
    writeln(byte.sizeof);
    writeln(ubyte.sizeof);
    writeln(char.sizeof);
    writeln(short.sizeof);
    writeln(ushort.sizeof);
    writeln(wchar.sizeof);
    writeln(int.sizeof);
    writeln(uint.sizeof);
    writeln(dchar.sizeof);
    writeln(long.sizeof);
    writeln(ulong.sizeof);
    writeln(float.sizeof);
    writeln(double.sizeof);
    writeln(real.sizeof);
}
```



```
mike:1$ rdmd basic_types.d
0
0

1
1
1
1
2
2
2
4
4
4
8
8
4
8
16
mike:1$
```

Basic Types - D gets the Defaults Right (2/2)

- You can find the defaults and properties here:
  - https://dlang.org/spec/property.html
  - To the right are some examples of some of the things you can query -- see the documentation for more.

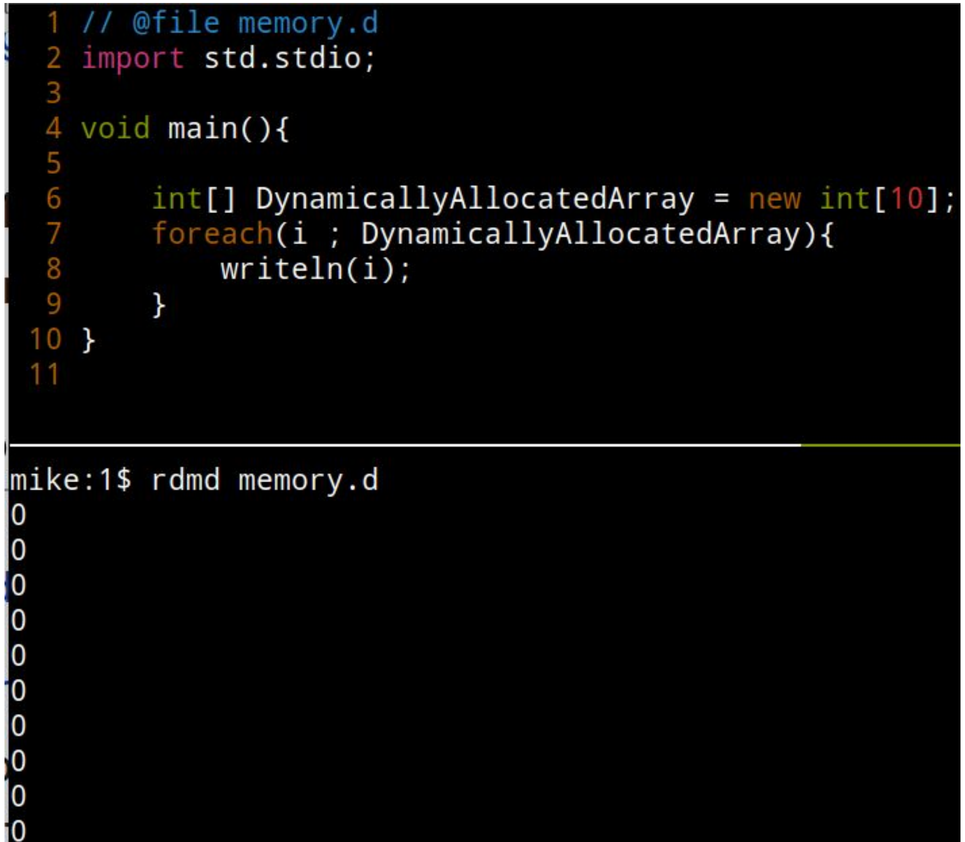


Properties for All Types

Property	Description
`.init`	initializer
`.sizeof`	size in bytes
`.alignof`	alignment size
`.mangleof`	string representing the 'mangled' representation of the type
`.stringof`	string representing the source representation of the type

Memory - D is a systems language (1/3)

- D has a garbage collector (gc) that is on by default (it can be turned off)
  - This means that we don't have to explicitly delete memory that we have allocated.
  - In the example on the right, we dynamically allocate an array of 10 integers
  - Then I use a 'foreach' loop to display them all.
  - The garbage collector will periodically run, and remove any memory that cannot be reached for us.

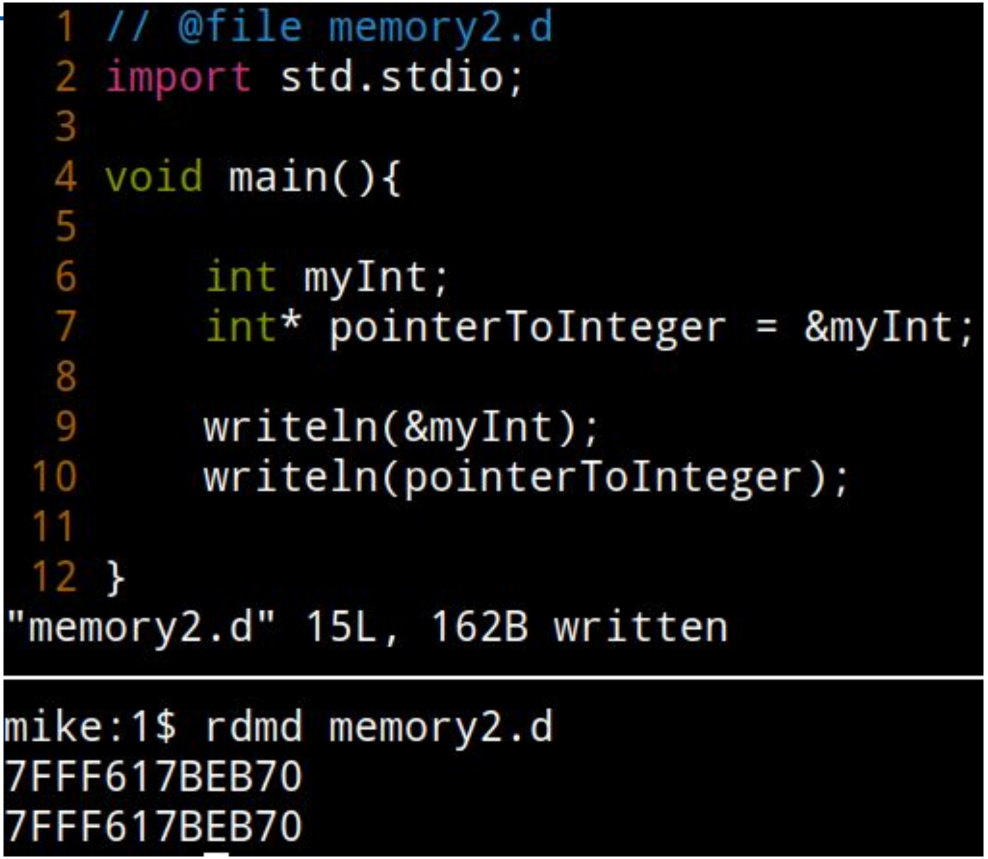


```d
// @file memory.d
import std.stdio;

void main(){

    int[] DynamicallyAllocatedArray = new int[10];
    foreach(i ; DynamicallyAllocatedArray){
        writeln(i);
    }
}

```

```
mike:1$ rdmd memory.d
0
0
0
0
0
0
0
0
0
0
```

Memory - D is a systems language (2/3)

- D does allow us to use pointers as shown on line 7
- We can use the '&' operator to get the **address of** a variable.
    - Observe the address printed out below.

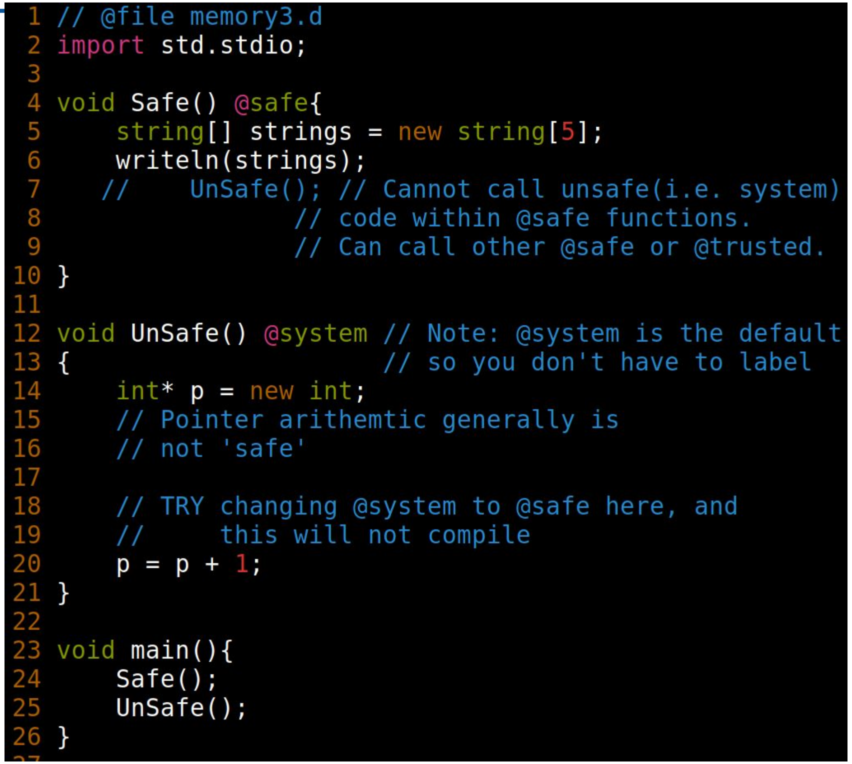


```
 1 // @file memory2.d
 2 import std.stdio;
 3
 4 void main(){
 5
 6     int myInt;
 7     int* pointerToInteger = &myInt;
 8
 9     writeln(&myInt);
10     writeln(pointerToInteger);
11
12 }
"memory2.d" 15L, 162B written
```

```
mike:1$ rdmd memory2.d
7FFF617BEB70
7FFF617BEB70
```

Memory - D is a systems language (3/3)

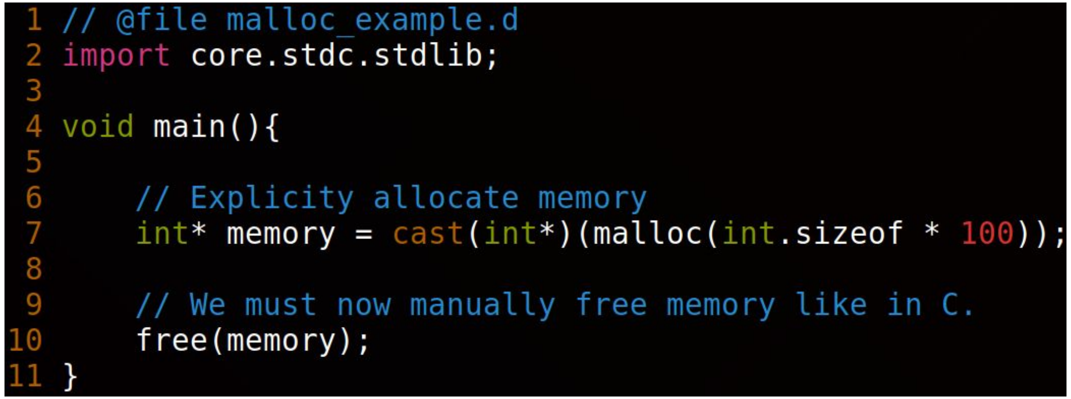
- D pays extra attention to memory safety.
  - You can add an @safe attribute after a function, and this will ensure that memory safety bugs are avoided.
  - @system is the ‘default’ however -- so observe on line 9 we can manipulate memory.
    - While this is the default,
    - **try changing** @system to @safe on line 9, you’ll see the compiler give you an error that this is not verified to be safe code.



```
 1 // @file memory3.d
 2 import std.stdio;
 3 
 4 void Safe() @safe{
 5     string[] strings = new string[5];
 6     writeln(strings);
 7    //    UnSafe(); // Cannot call unsafe(i.e. system)
 8                 // code within @safe functions.
 9                 // Can call other @safe or @trusted.
10 }
11 
12 void UnSafe() @system // Note: @system is the default
13 {                     // so you don't have to label
14     int* p = new int;
15     // Pointer arithemtic generally is
16     // not 'safe'
17 
18     // TRY changing @system to @safe here, and
19     //     this will not compile
20     p = p + 1;
21 }
22 
23 void main(){
24     Safe();
25     UnSafe();
26 }
```

(Aside)Explicit Memory Allocation (1/3)

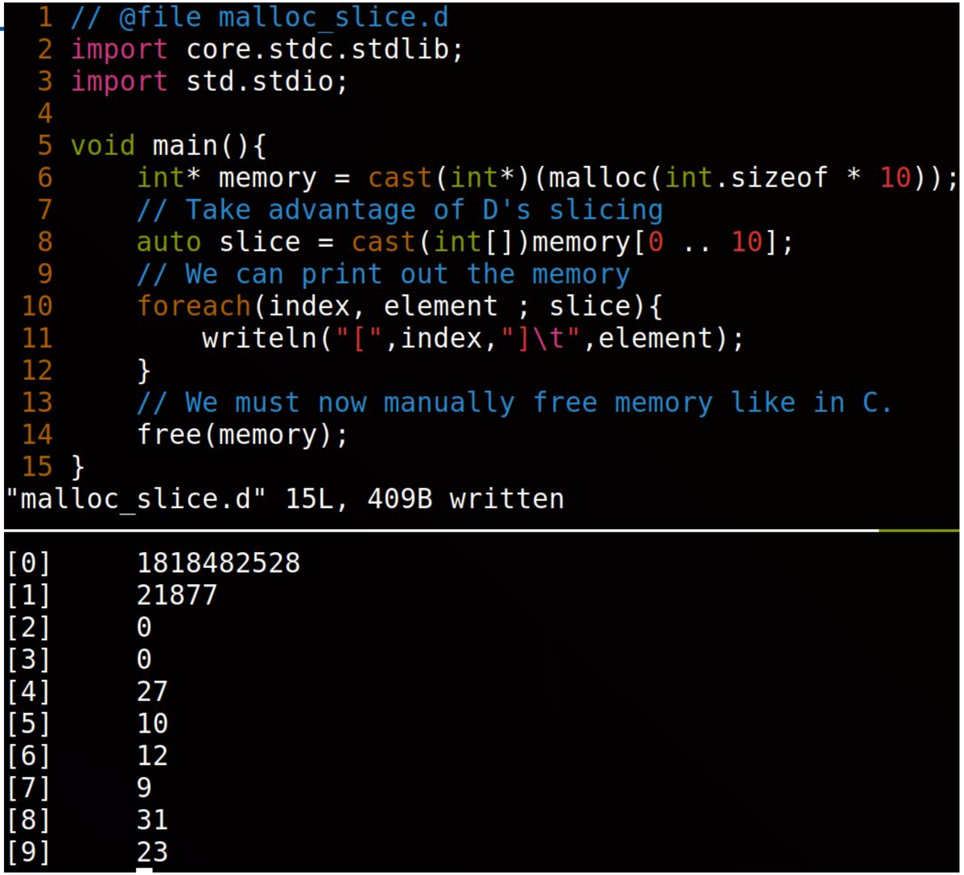
- We can also use the standard C libraries (libc) malloc and free functionality to allocate our own memory.
  - If you want to completely disable the garbage collector, that is also an option
- The point is that D gives you several options for how to handle memory.



```d
// @file malloc_example.d
import core.stdc.stdlib;

void main(){

    // Explicity allocate memory
    int* memory = cast(int*)(malloc(int.sizeof * 100));

    // We must now manually free memory like in C.
    free(memory);
}
```

(Aside)Explicit Memory Allocation (2/3)

- Observe in this example, we can create a ‘slice’ (line 8) and access the memory more conveniently.
    - Note: When we write out the malloc’d memory that our slice also points to, observe that we have garbage values.
        - So when using C libraries, we play by C’s rules (memory is not initialized).
        - In this class, prefer to just use D’s garbage collector unless otherwise stated.
        - (Note: In D we can do: auto[50] memory= void; if we want uninitialized memory)

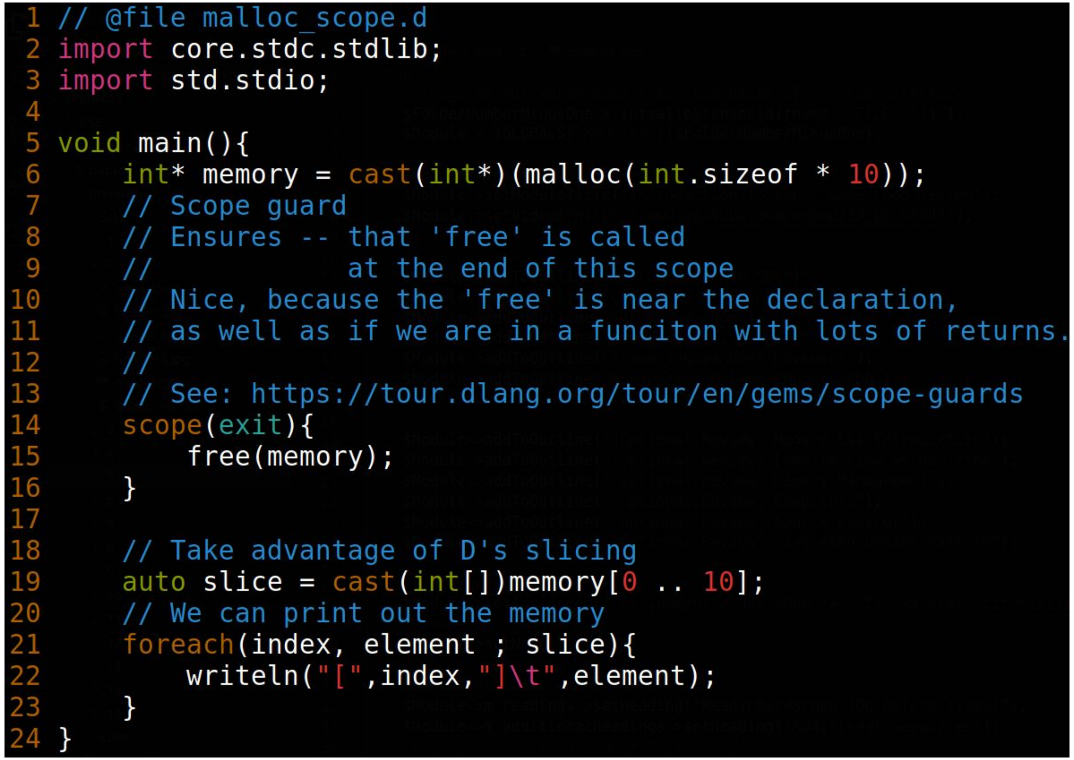


```
 1 // @file malloc_slice.d
 2 import core.stdc.stdlib;
 3 import std.stdio;
 4
 5 void main(){
 6     int* memory = cast(int*)(malloc(int.sizeof * 10));
 7     // Take advantage of D's slicing
 8     auto slice = cast(int[])memory[0 .. 10];
 9     // We can print out the memory
10     foreach(index, element ; slice){
11         writeln("[",index,"]\t",element);
12     }
13     // We must now manually free memory like in C.
14     free(memory);
15 }
"malloc_slice.d" 15L, 409B written
```

```
[0]     1818482528
[1]     21877
[2]     0
[3]     0
[4]     27
[5]     10
[6]     12
[7]     9
[8]     31
[9]     23
```

(Aside)Explicit Memory Allocation (3/3)

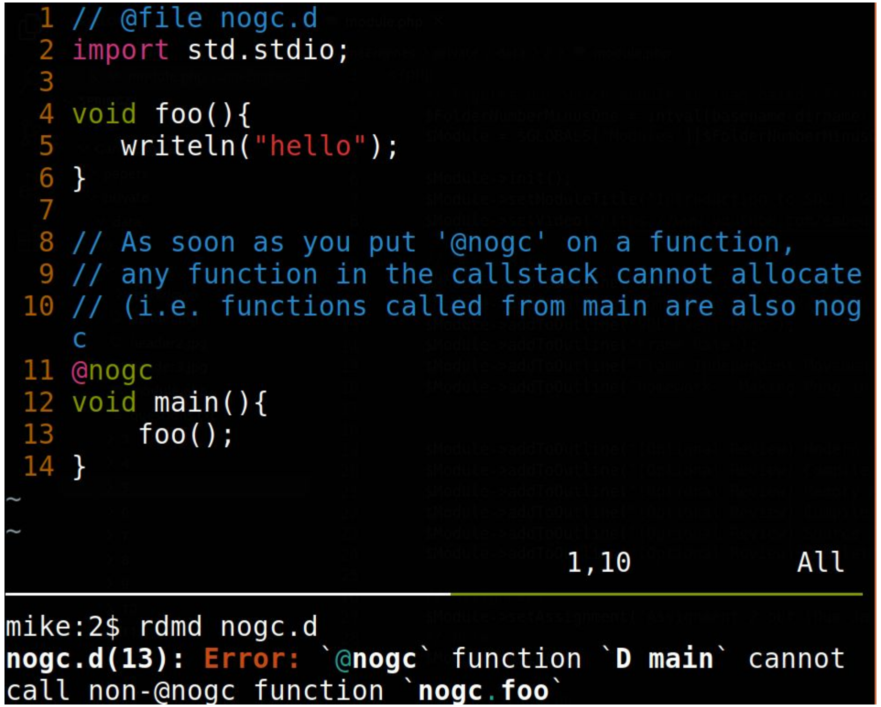
- Here’s another idiomatic D language improvement with what’s called a **scope guard**.
  - Notice at line 14-16, I can create what is equivalent to a try-catch-finally block.
  - Scope guards however (with ‘exit’) will always execute, and is a bit cleaner in my opinion with complicated control flow.



```
 1 // @file malloc_scope.d
 2 import core.stdc.stdlib;
 3 import std.stdio;
 4 
 5 void main(){
 6     int* memory = cast(int*)(malloc(int.sizeof * 10));
 7     // Scope guard
 8     // Ensures -- that 'free' is called
 9     //             at the end of this scope
10     // Nice, because the 'free' is near the declaration,
11     // as well as if we are in a funciton with lots of returns.
12     //
13     // See: https://tour.dlang.org/tour/en/gems/scope-guards
14     scope(exit){
15         free(memory);
16     }
17 
18     // Take advantage of D's slicing
19     auto slice = cast(int[])memory[0 .. 10];
20     // We can print out the memory
21     foreach(index, element ; slice){
22         writeln("[",index,"]\t",element);
23     }
24 }
```

Avoiding Garbage Collection - @nogc

- The @nogc attribute can mark a function as something that will not collect.
    - You can effectively disable garbage collection for your entire program, but the attribute is transitive
    - Meaning if you allocate (which writeln does -- we need space for a string), then you cannot use those functions.
- With care @nogc can help give you performance for allocations when needed.

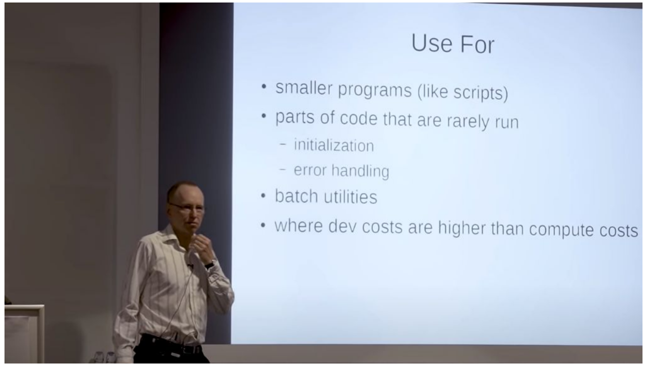


```
// @file nogc.d
import std.stdio;

void foo(){
    writeln("hello");
}

// As soon as you put '@nogc' on a function,
// any function in the callstack cannot allocate
// (i.e. functions called from main are also nog
c
@nogc
void main(){
     foo();
}
~
~
                                    1,10          All
```

```
mike:2$ rdmd nogc.d
nogc.d(13): Error: `@nogc` function `D main` cannot
call non-@nogc function `nogc.foo`
```

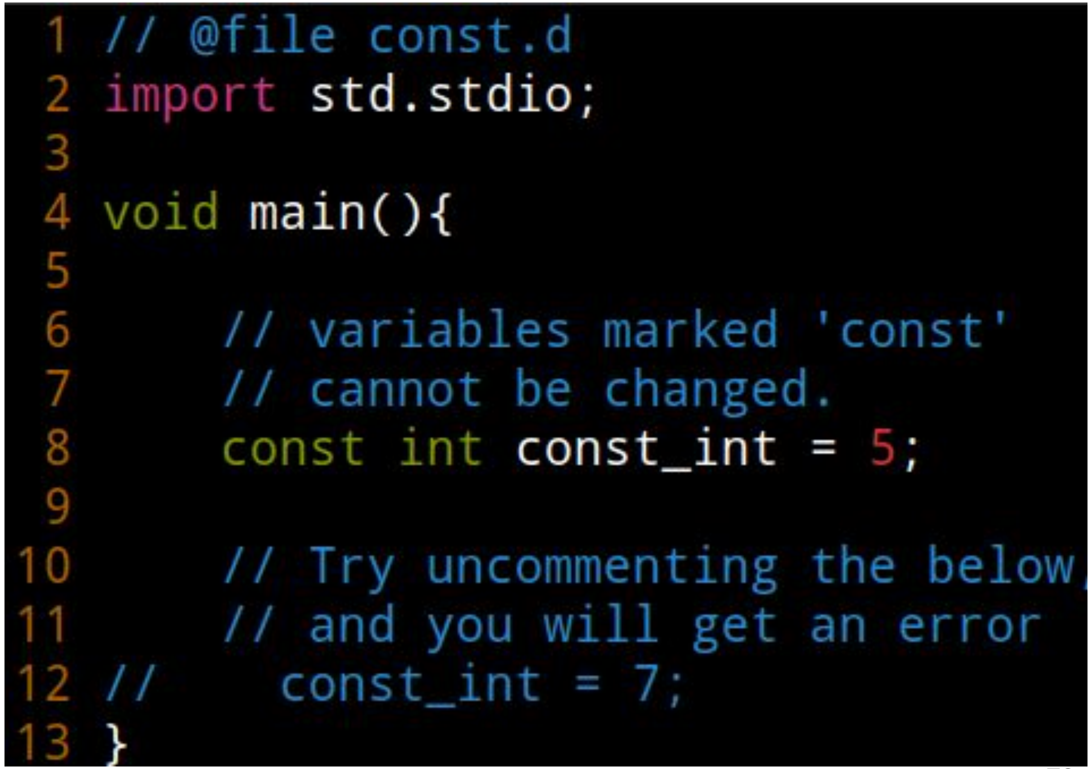
(Aside) More on D’s Memory Allocation

- Here’s a list of articles for more on memory allocation, in an order that would be reasonable to read them.
- https://dlang.org/blog/the-gc-series/
    - A series of several articles on the garbage collector (gc), stack, heap, profiling, and more
    - We’ll talk about some of these topics throughout the course.
- Garbage Collection in the D Programming Language
    - https://dlang.org/spec/garbage.html
- https://dlang.org/blog/2017/06/16/life-in-the-fast-lane/
- DConf 2019 Day 1 Keynote: Allocating Memory with the D Programming Language -- Walter Bright
    - https://www.youtube.com/watch?v=_PB6Hdi4R7M


Use For
• smaller programs (like scripts)
• parts of code that are rarely run
- initialization
- error handling
• batch utilities
• where dev costs are higher than compute costs

const

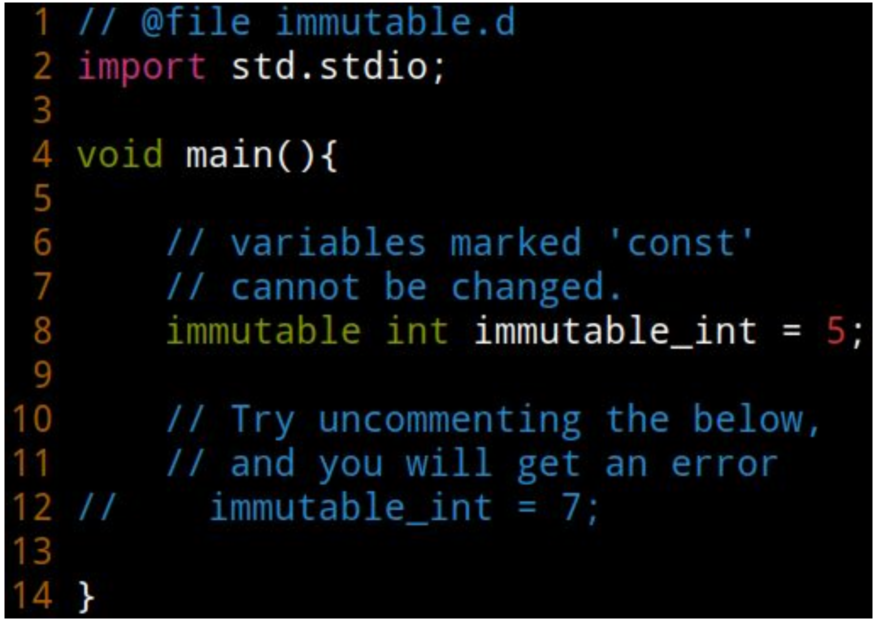
- D supports ‘const’ qualifier on variables.
    - This means that you cannot in the current scope change the value
- In general -- we like making data ‘const’ to minimize state in our program if data is read-only



```
// @file const.d
import std.stdio;

void main(){

    // variables marked 'const'
    // cannot be changed.
    const int const_int = 5;

    // Try uncommenting the below
    // and you will get an error
//    const_int = 7;
}
```

immutable - Data never ever changes

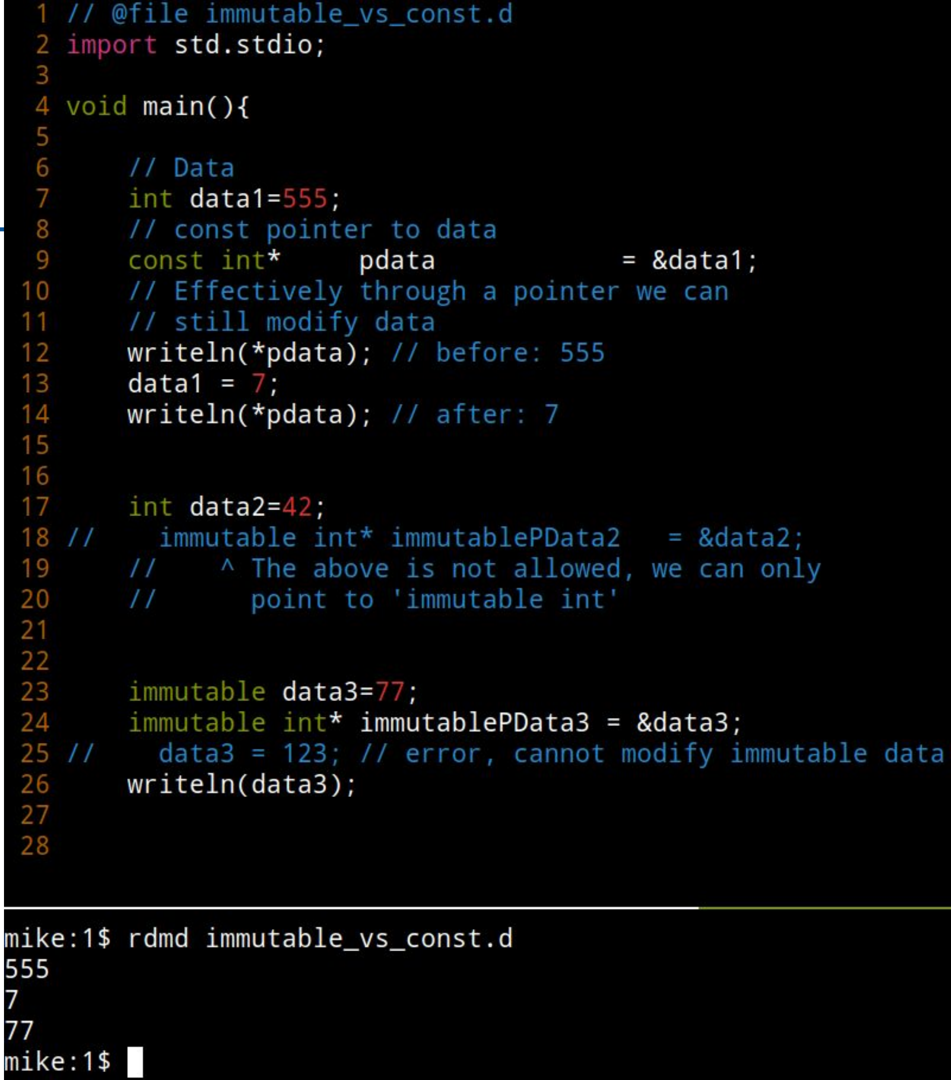
- immutable data in D is truly *read-only*
  - This is even safer -- and important for parallel programming.
    - i.e. we like a guarantee that data cannot change.
- This becomes very important when working with pointers
  - (next slide)



```
// @file immutable.d
import std.stdio;

void main(){

    // variables marked 'const'
    // cannot be changed.
    immutable int immutable_int = 5;

    // Try uncommenting the below,
    // and you will get an error
//    immutable_int = 7;

}
```

Immutable data is safer

- Here’s an example showing that with immutable data, we get an even stronger guarantee
- const is still good to use -- just means we cannot reassign our pointer
  - The underlying data may change however.

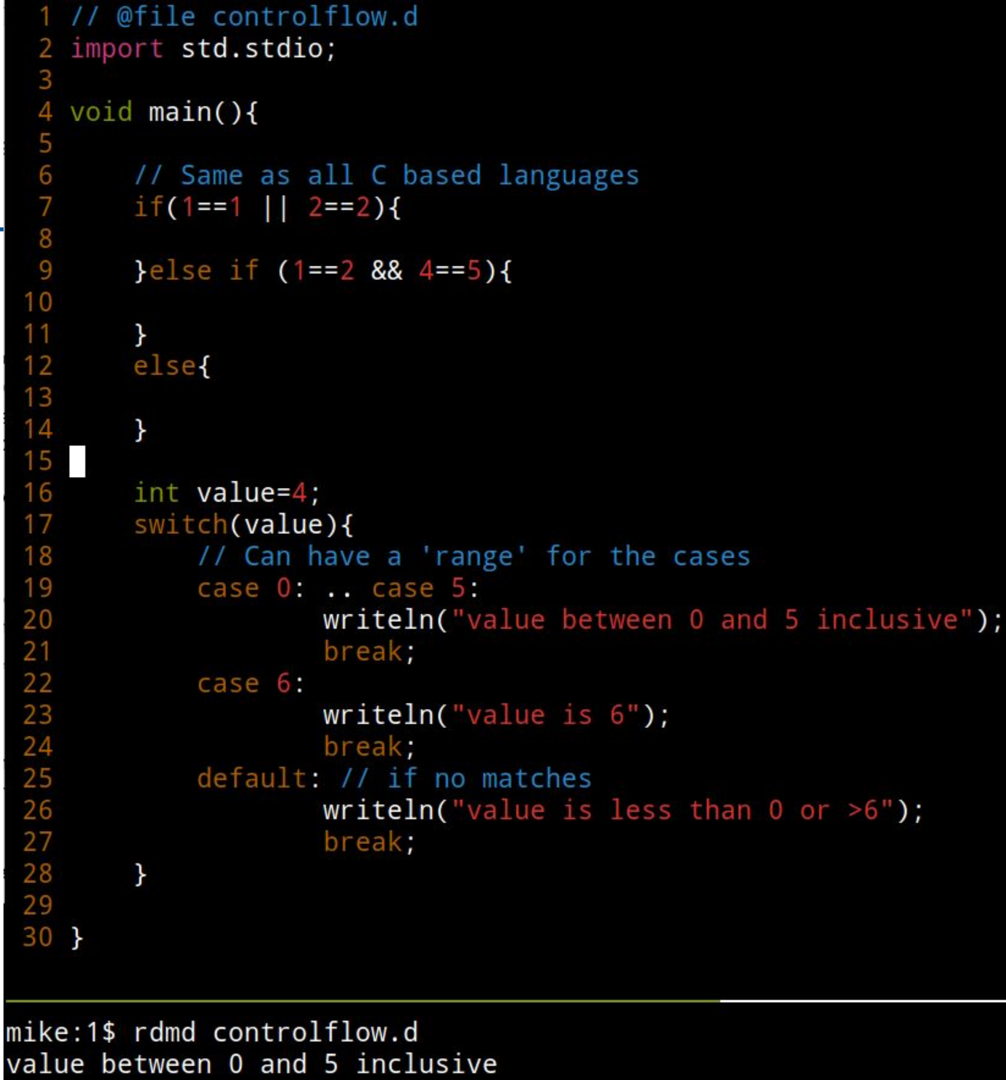


```d
// @file immutable_vs_const.d
import std.stdio;

void main(){

    // Data
    int data1=555;
    // const pointer to data
    const int*     pdata            = &data1;
    // Effectively through a pointer we can
    // still modify data
    writeln(*pdata); // before: 555
    data1 = 7;
    writeln(*pdata); // after: 7


    int data2=42;
//    immutable int* immutablePData2   = &data2;
    //    ^ The above is not allowed, we can only
    //      point to 'immutable int'


    immutable data3=77;
    immutable int* immutablePData3 = &data3;
//    data3 = 123; // error, cannot modify immutable data
    writeln(data3);


```

```
mike:1$ rdmd immutable_vs_const.d
555
7
77
mike:1$ 
```

Control Flow

- if/else/elseif supported in other languages
- switch statement more powerful than C and C++
  - Can support ranges
    - e.g. case 0: .. case 5:
  - Can also switch on 'strings' as well.
  - Can also switch on enums

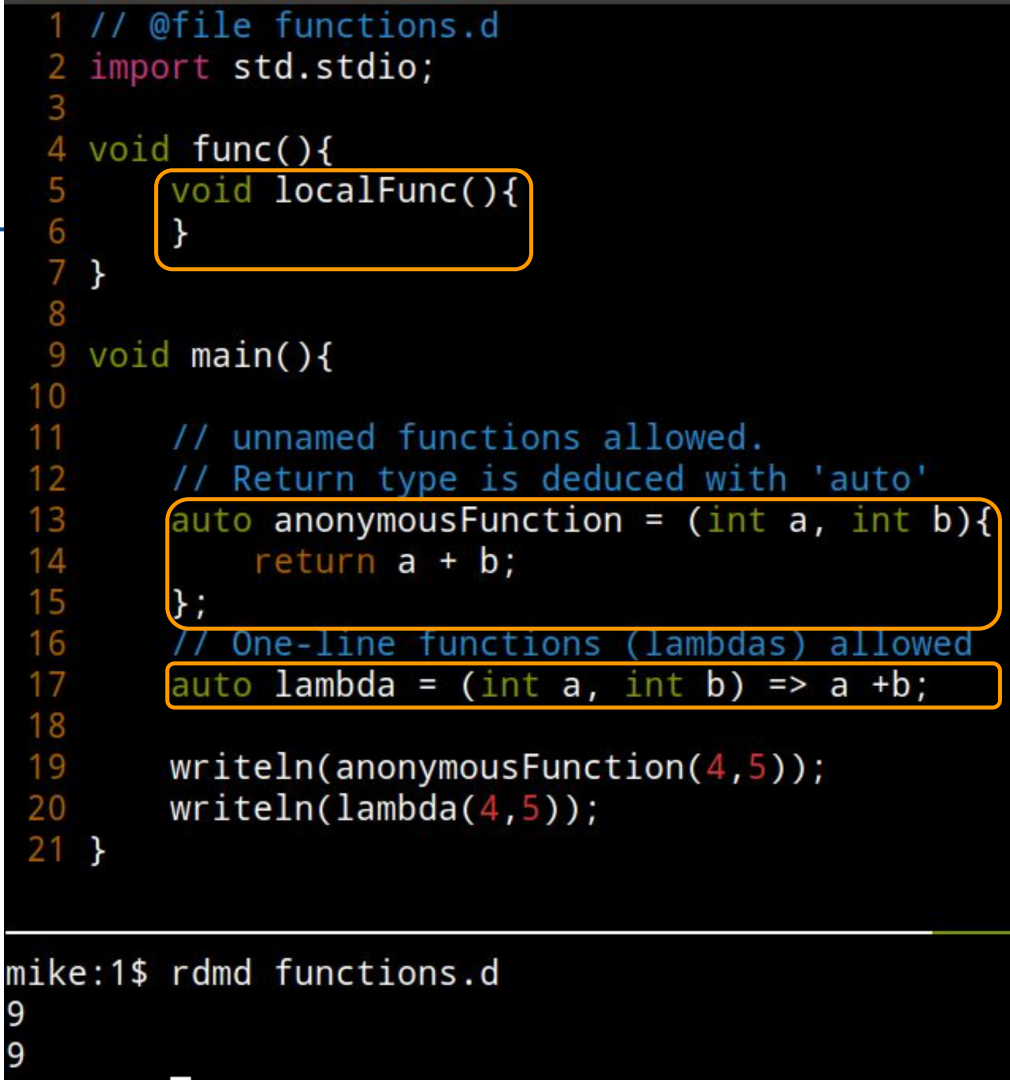


```d
// @file controlflow.d
import std.stdio;

void main(){

	// Same as all C based languages
	if(1==1 || 2==2){

	}else if (1==2 && 4==5){

	}
	else{

	}

	int value=4;
	switch(value){
		// Can have a 'range' for the cases
		case 0: .. case 5:
			writeln("value between 0 and 5 inclusive");
			break;
		case 6:
			writeln("value is 6");
			break;
		default: // if no matches
			writeln("value is less than 0 or >6");
			break;
	}

}
```

```
mike:1$ rdmd controlflow.d
value between 0 and 5 inclusive
```

Functions

- D allows local functions (line 5 and 6) for further encapsulation
- At lines 13 you can also create anonymous (unnamed) functions
- Line 17 shows another way to create a one line function (lambda)

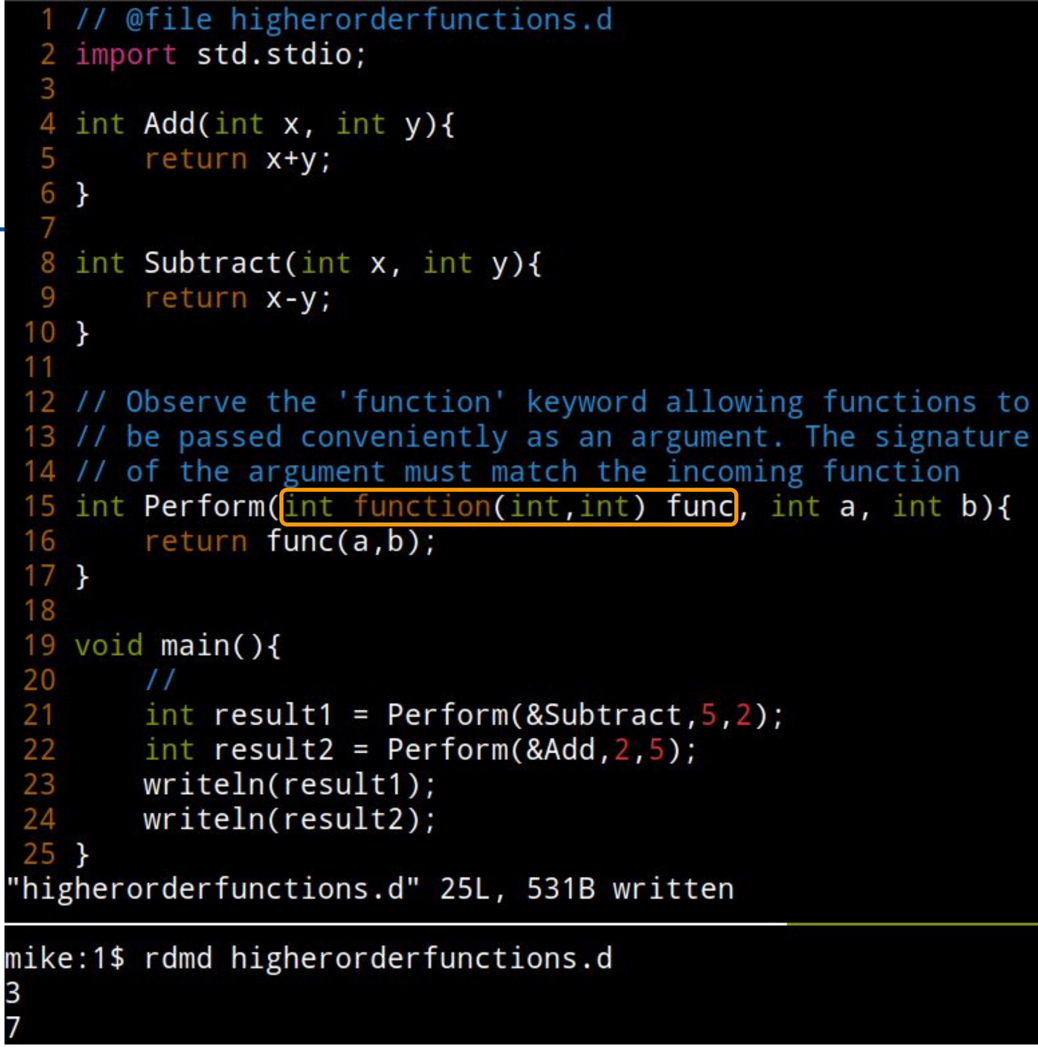


```d
 1 // @file functions.d
 2 import std.stdio;
 3
 4 void func(){
 5     void localFunc(){
 6     }
 7 }
 8
 9 void main(){
10
11     // unnamed functions allowed.
12     // Return type is deduced with 'auto'
13     auto anonymousFunction = (int a, int b){
14         return a + b;
15     };
16     // One-line functions (lambdas) allowed
17     auto lambda = (int a, int b) => a +b;
18
19     writeln(anonymousFunction(4,5));
20     writeln(lambda(4,5));
21 }
```

```
mike:1$ rdmd functions.d
9
9
```

Higher Order Functions

- D supports the passing of functions using a nice syntax, the **function** keyword
  - Note: When working with classes/structs, if we want a function pointer to a member function we use **delegates** to capture state.

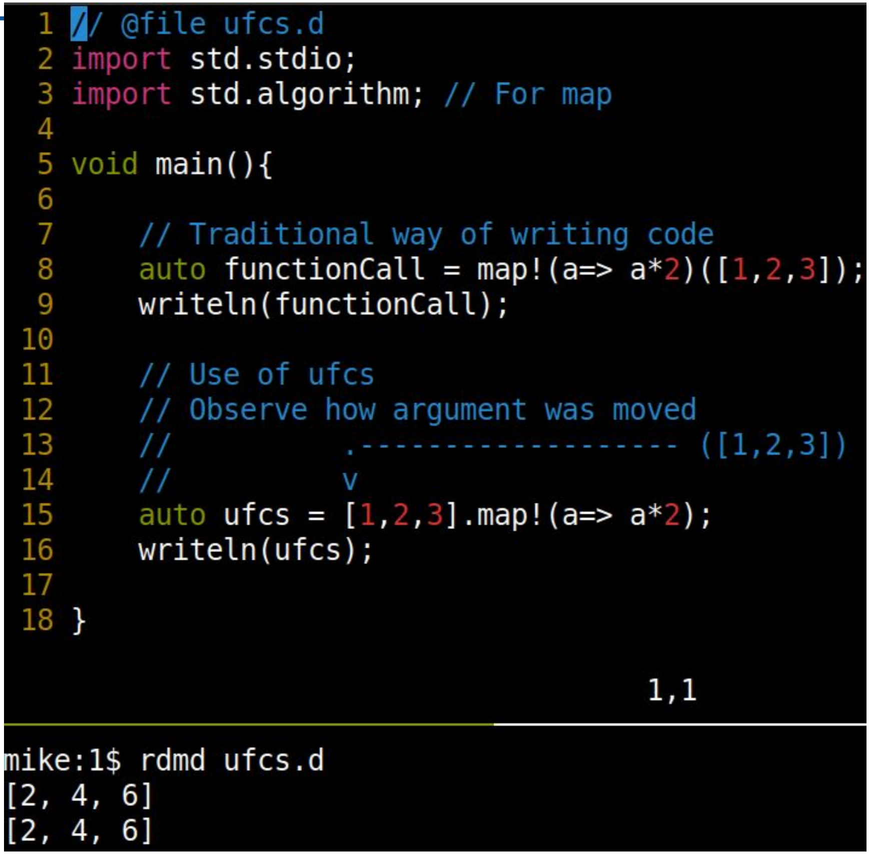


```d
// @file higherorderfunctions.d
import std.stdio;

int Add(int x, int y){
    return x+y;
}

int Subtract(int x, int y){
    return x-y;
}

// Observe the 'function' keyword allowing functions to
// be passed conveniently as an argument. The signature
// of the argument must match the incoming function
int Perform(int function(int,int) func, int a, int b){
    return func(a,b);
}

void main(){
    //
    int result1 = Perform(&Subtract,5,2);
    int result2 = Perform(&Add,2,5);
    writeln(result1);
    writeln(result2);
}
"higherorderfunctions.d" 25L, 531B written
```

```
mike:1$ rdmd higherorderfunctions.d
3
7
```

Universal Function Call Syntax and Chaining (1/2)

- Allows you to call free functions with the '.' syntax
    - e.g.
        - func(param)) is called as
        - param.func.
    - d tour - uniform-function-call-syntax-ufcs
- Article by Walter Bright
    - [archived link]

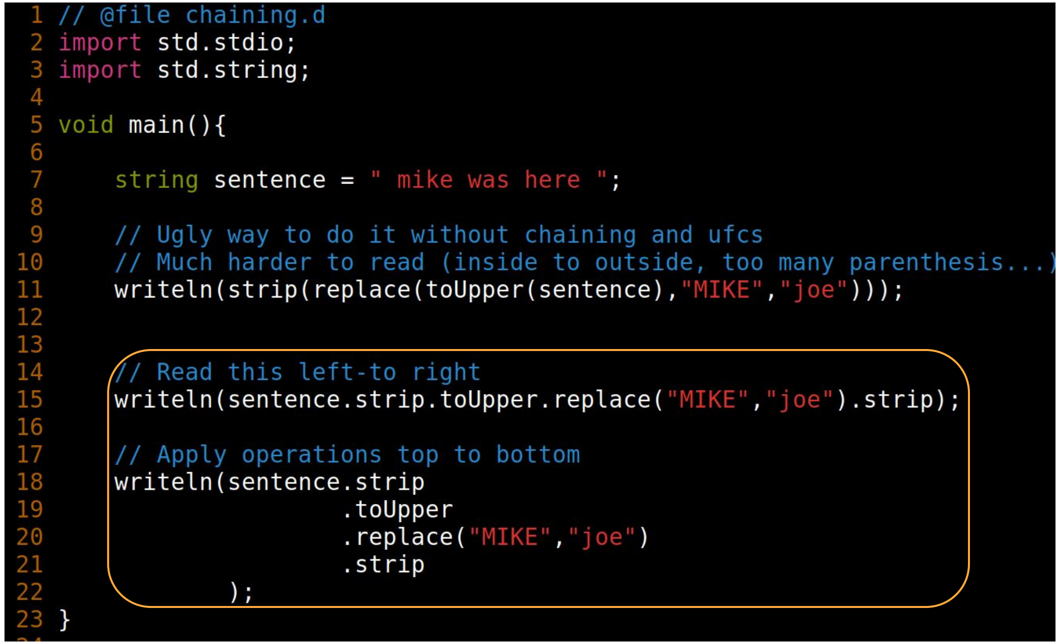


```
// @file ufcs.d
import std.stdio;
import std.algorithm; // For map

void main(){

    // Traditional way of writing code
    auto functionCall = map!(a=> a*2)([1,2,3]);
    writeln(functionCall);

    // Use of ufcs
    // Observe how argument was moved
    //           .-------------------- ([1,2,3])
    //           v
    auto ufcs = [1,2,3].map!(a=> a*2);
    writeln(ufcs);

}
                                                1,1
```

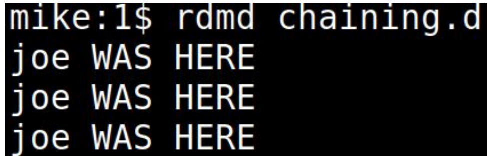
```
mike:1$ rdmd ufcs.d
[2, 4, 6]
[2, 4, 6]
```

Universal Function Call Syntax and Chaining (2/2)

- UFCS allows you to more conveniently chain together function calls
  - Here's an example of chaining together several calls
- Note: It can be useful to space out the calls.



```d
// @file chaining.d
import std.stdio;
import std.string;

void main(){

    string sentence = " mike was here ";

    // Ugly way to do it without chaining and ufcs
    // Much harder to read (inside to outside, too many parenthesis...)
    writeln(strip(replace(toUpper(sentence),"MIKE","joe")));


    // Read this left-to right
    writeln(sentence.strip.toUpper.replace("MIKE","joe").strip);

    // Apply operations top to bottom
    writeln(sentence.strip
                    .toUpper
                    .replace("MIKE","joe")
                    .strip
            );
}
```



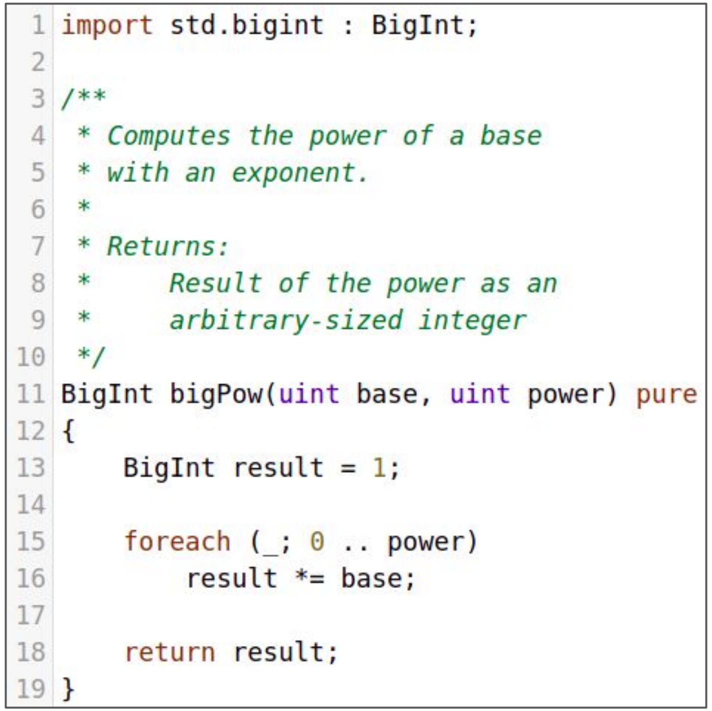
```
mike:1$ rdmd chaining.d
joe WAS HERE
joe WAS HERE
joe WAS HERE
```

More on Functions

- D Supports more with functions:
  - pure
  - lazy
  - memoization

pure functions [dlang tour on pure]

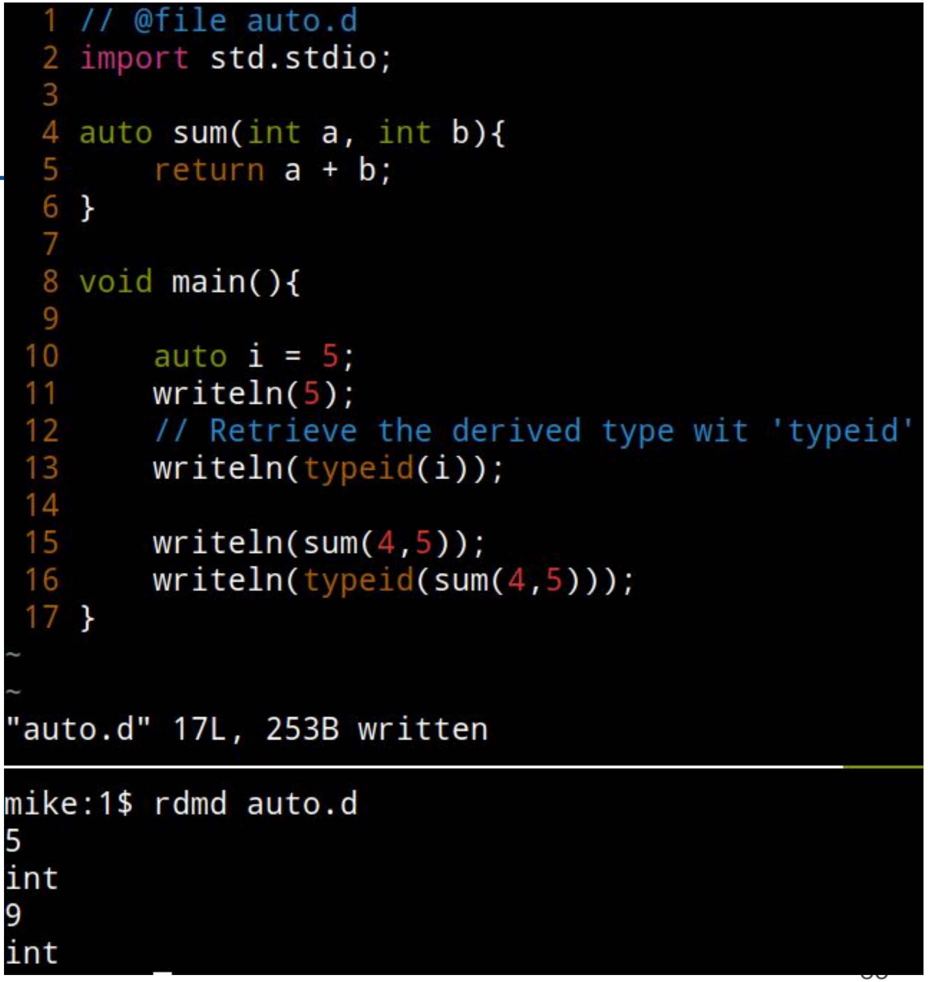
- Function purity is also an important part of functional programming
  - A strongly pure function is one which has no side effects (i.e. parameters are not modified.)
    - The same input provides the same output
  - We can also have ‘weakly pure’ functions which have mutable parameters (parameters passed by reference whether explicitly (with the ref parameter) or implicity (e.g. a class or pointer)
- More on Pure: [Dlang Episode 68] D Language - Functions - Part 15 of n - pure functions



```
import std.bigint : BigInt;

/**
 * Computes the power of a base
 * with an exponent.
 *
 * Returns:
 *      Result of the power as an
 *      arbitrary-sized integer
 */
BigInt bigPow(uint base, uint power) pure
{
    BigInt result = 1;

    foreach (_; 0 .. power)
        result *= base;

    return result;
}
```

Type Deduction with auto

- D allows for type deduction with the ‘auto’ keyword
- For functions the return type can be deduced using ‘auto’
    - (arguments of functions however cannot be auto, unless they are ‘auto ref’)
- “almost always auto” - is the general rule (is fine if the type is obvious, but I usually prefer explicit types stil)

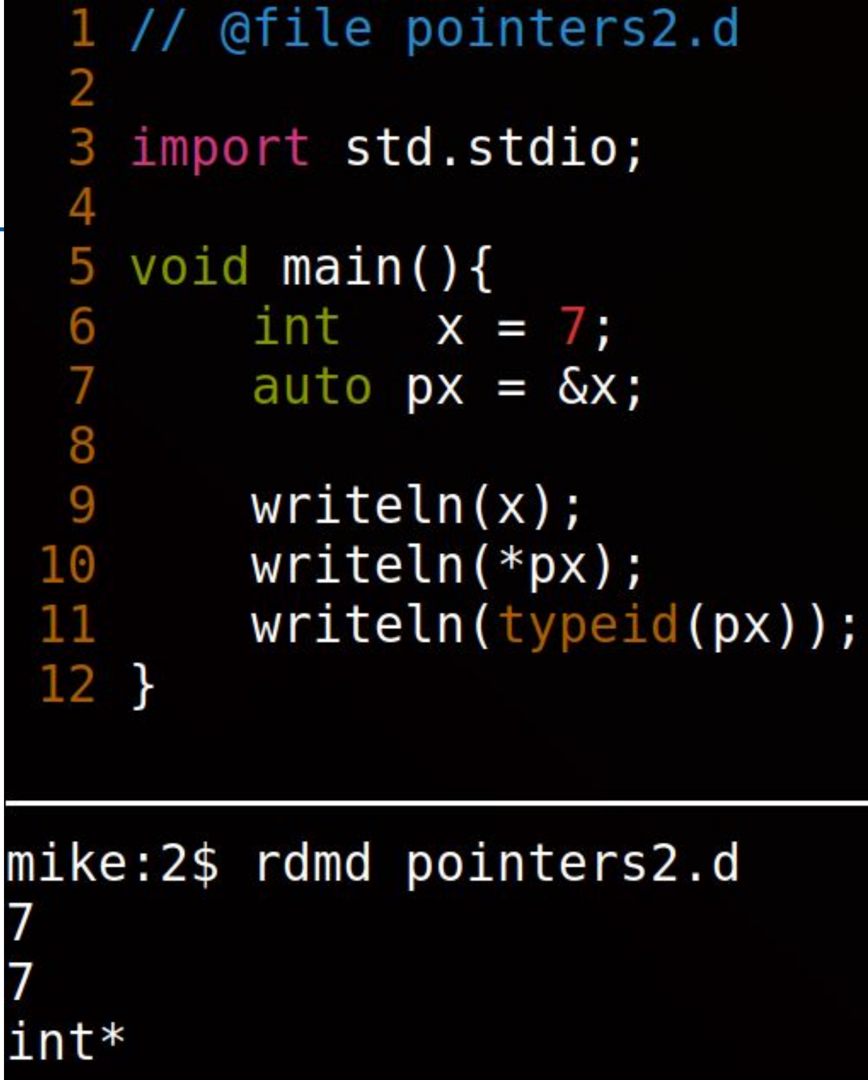


```
// @file auto.d
import std.stdio;

auto sum(int a, int b){
    return a + b;
}

void main(){

    auto i = 5;
    writeln(5);
    // Retrieve the derived type wit 'typeid'
    writeln(typeid(i));

    writeln(sum(4,5));
    writeln(typeid(sum(4,5)));
}
~
~
"auto.d" 17L, 253B written
```

```
mike:1$ rdmd auto.d
5
int
9
int
```

(Aside) ‘auto’

- On line 7 I have used ‘auto’ in D to declare the type.
- ‘auto’ is smart enough to deduce that we’re storing in address on the right hand side, and that the type of ‘x’ is an int.
    - Thus: typeid(px) is a int*
    - (This code is equivalent to the previous slide)

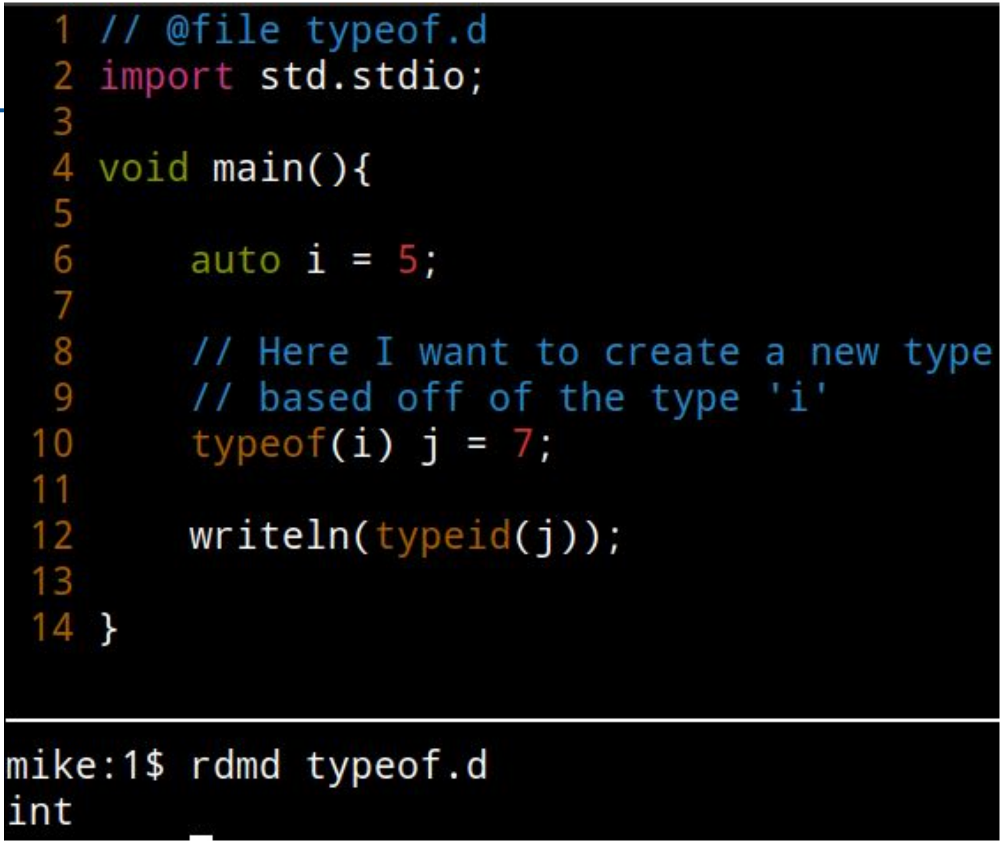


```
 1 // @file pointers2.d
 2
 3 import std.stdio;
 4
 5 void main(){
 6         int    x = 7;
 7         auto px = &x;
 8
 9         writeln(x);
10         writeln(*px);
11         writeln(typeid(px));
12 }
```

```
mike:2$ rdmd pointers2.d
7
7
int*
```

Type Creation - typeof

- Making use of type deduction to create new types with **typeof**
  - This example ensures that whatever type is deduced from i, the variable ‘j’ will also be that type.

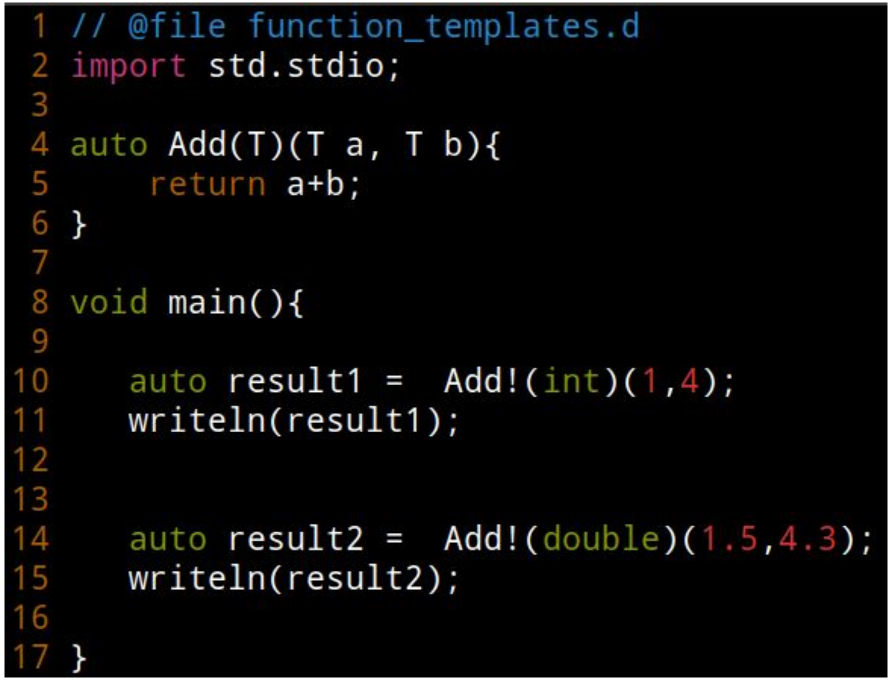


```d
// @file typeof.d
import std.stdio;

void main(){

    auto i = 5;

    // Here I want to create a new type
    // based off of the type 'i'
    typeof(i) j = 7;

    writeln(typeid(j));

}
```

```
mike:1$ rdmd typeof.d
int
```

Function Templates

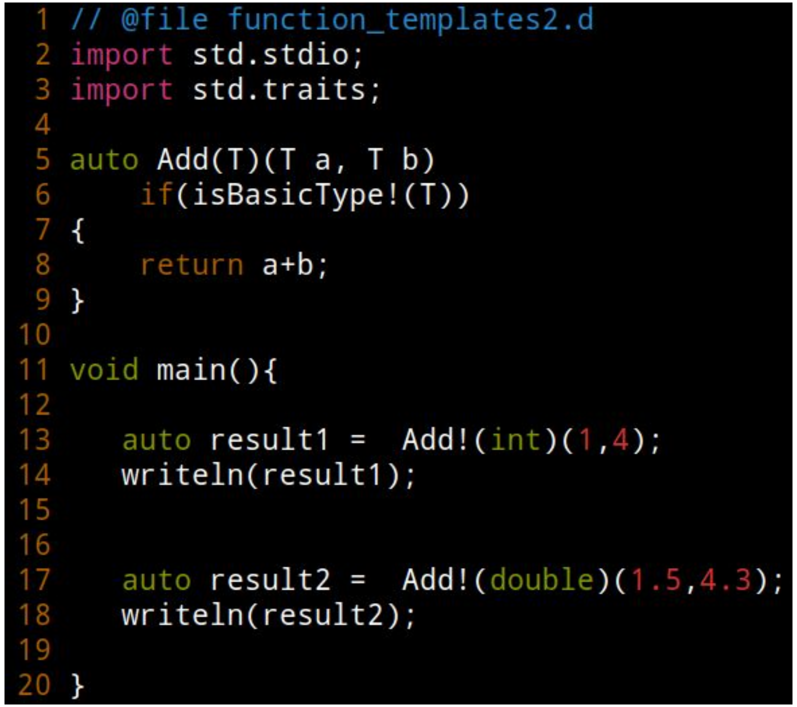
- Function template syntax allows you to parametrize your functions.
  - Line 4, ‘T’ is substituted for the type
  - At line 10 and line 14 the type ‘int’ and ‘double’ respectively is substituted in.
- The notation again for choosing the type is with the ‘!’
  - Other languages use a set of <>
    - e.g. C++: std::vector<int>
    - e.g. Java: List<String>



```
 1 // @file function_templates.d
 2 import std.stdio;
 3
 4 auto Add(T)(T a, T b){
 5      return a+b;
 6 }
 7
 8 void main(){
 9
10     auto result1 =  Add!(int)(1,4);
11     writeln(result1);
12
13
14     auto result2 =  Add!(double)(1.5,4.3);
15     writeln(result2);
16
17 }
```

Template Constraints

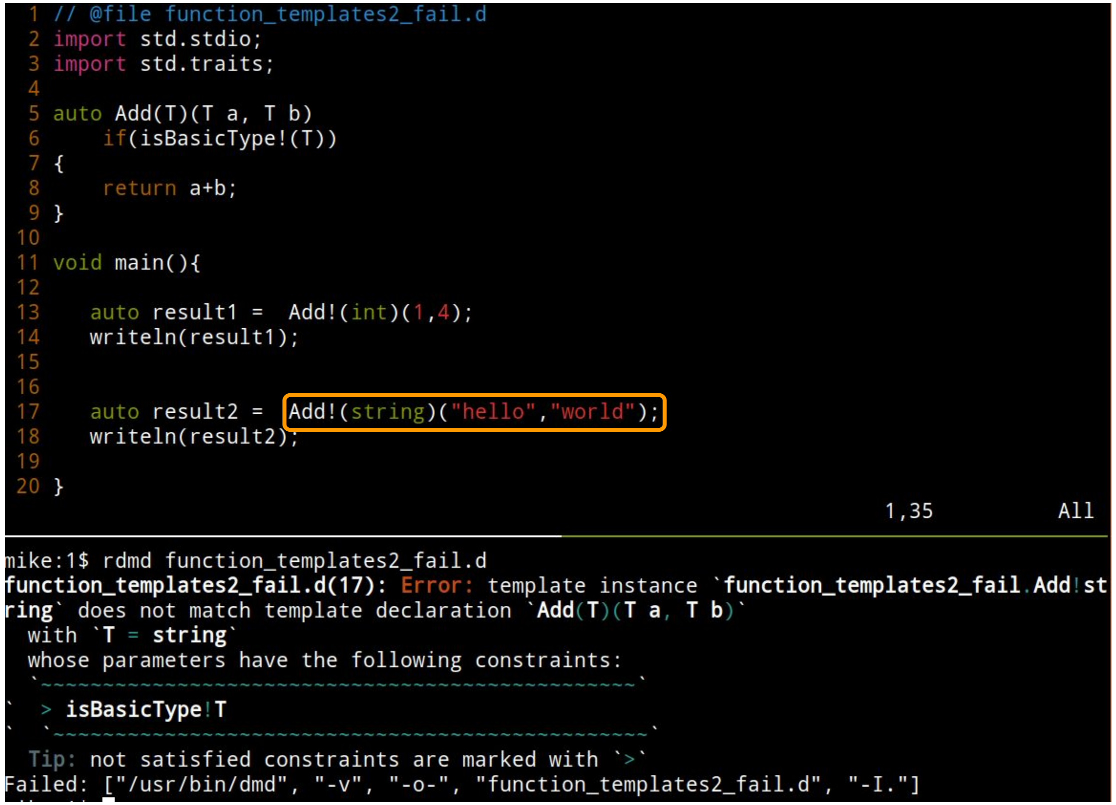
- At line 6 observe that we can further add *template constraints* for what is allowed
  - Note: We've added the std.traits library which lets us at compile-time check that the types are basic types
- Template constraints are probably something new, but D allows you to write them to ensure code meets requirements
  - https://dlang.org/articles/constraints.html
  - Template Constraints are like 'concepts in c++'



```
 1 // @file function_templates2.d
 2 import std.stdio;
 3 import std.traits;
 4
 5 auto Add(T)(T a, T b)
 6     if(isBasicType!(T))
 7 {
 8     return a+b;
 9 }
10
11 void main(){
12
13    auto result1 =  Add!(int)(1,4);
14    writeln(result1);
15
16
17    auto result2 =  Add!(double)(1.5,4.3);
18    writeln(result2);
19
20 }
```

Template Constraints - Fail

- This example fails as strings are not a basic type
  - Strings are an immutable array of characters
    - i.e. immutable char[]
- Note: `isBasicType` actually isn't the best way to check here, we'd rather have 'isAddable'
  - See: https://dlang.org/articles/constraints.html

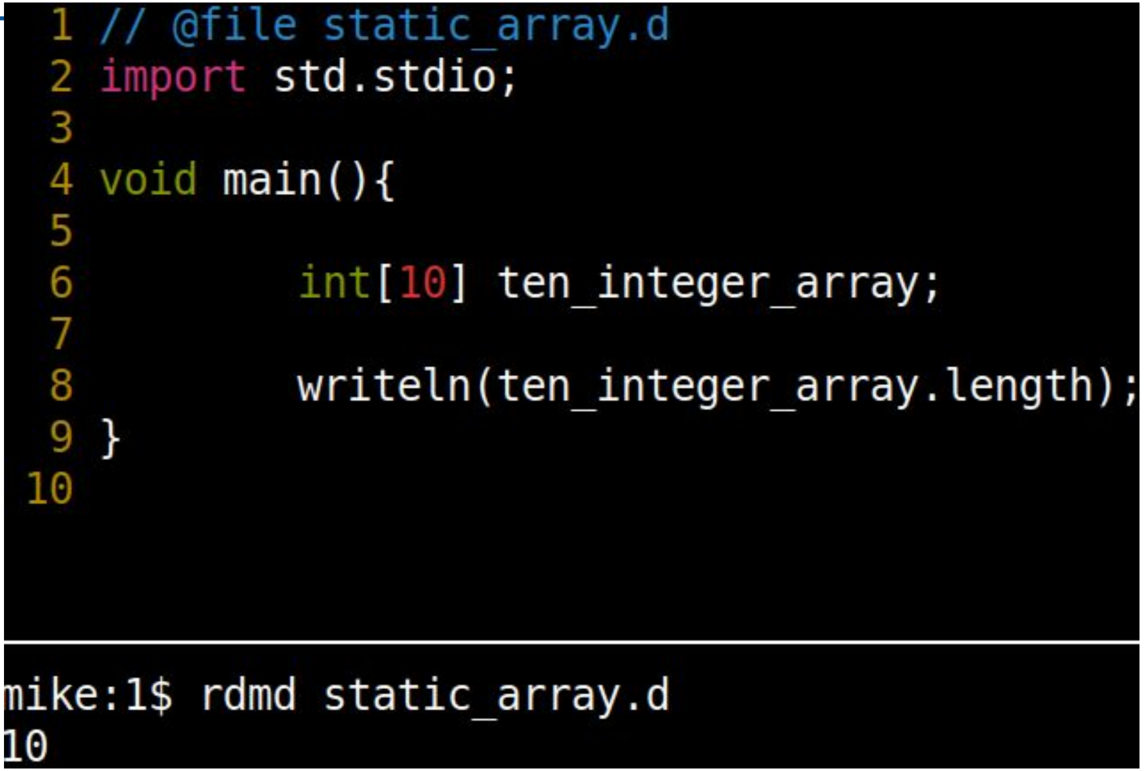


```
 1 // @file function_templates2_fail.d
 2 import std.stdio;
 3 import std.traits;
 4
 5 auto Add(T)(T a, T b)
 6     if(isBasicType!(T))
 7 {
 8     return a+b;
 9 }
10
11 void main(){
12
13    auto result1 =  Add!(int)(1,4);
14    writeln(result1);
15
16
17    auto result2 =  Add!(string)("hello","world");
18    writeln(result2);
19
20 }
                                                              1,35          All
```

```
mike:1$ rdmd function_templates2_fail.d
function_templates2_fail.d(17): Error: template instance `function_templates2_fail.Add!st
ring` does not match template declaration `Add(T)(T a, T b)`
  with `T = string`
  whose parameters have the following constraints:
  `~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`
`   > isBasicType!T
`  `~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`
  Tip: not satisfied constraints are marked with `>`
Failed: ["/usr/bin/dmd", "-v", "-o-", "function_templates2_fail.d", "-I."]
```

Static Arrays (Fixed-sized arrays)

- Static Arrays are declared with the type and the size.
- Sometimes these are also called ‘fixed-sized’ arrays.
  - These arrays are **stack allocated**.
  - Static arrays size cannot be changed -- they are fixed-size...

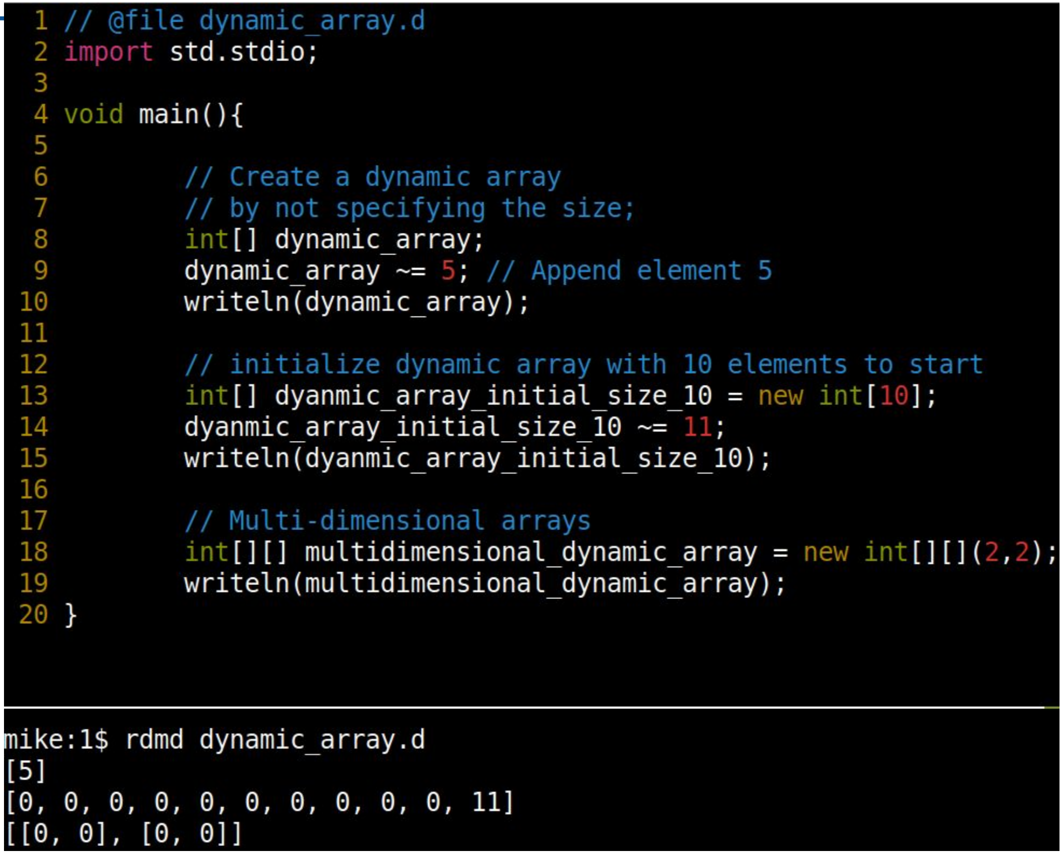


```d
// @file static_array.d
import std.stdio;

void main(){

		int[10] ten_integer_array;

		writeln(ten_integer_array.length);
}

```

```
mike:1$ rdmd static_array.d
10
```

Dynamic Arrays

- Dynamic Arrays
  - Heap allocated with ‘new’
  - Size can be queried with .length as well
  - Can be **concatenated** with ~ operator
    - (We do not overload the ‘+’ operator)
- https://tour.dlang.org/tour/en/basics/arrays
  - More: https://tastyminerals.github.io/tasty-blog/dlang/2020/03/22/multidimensional_arrays_in_d.html

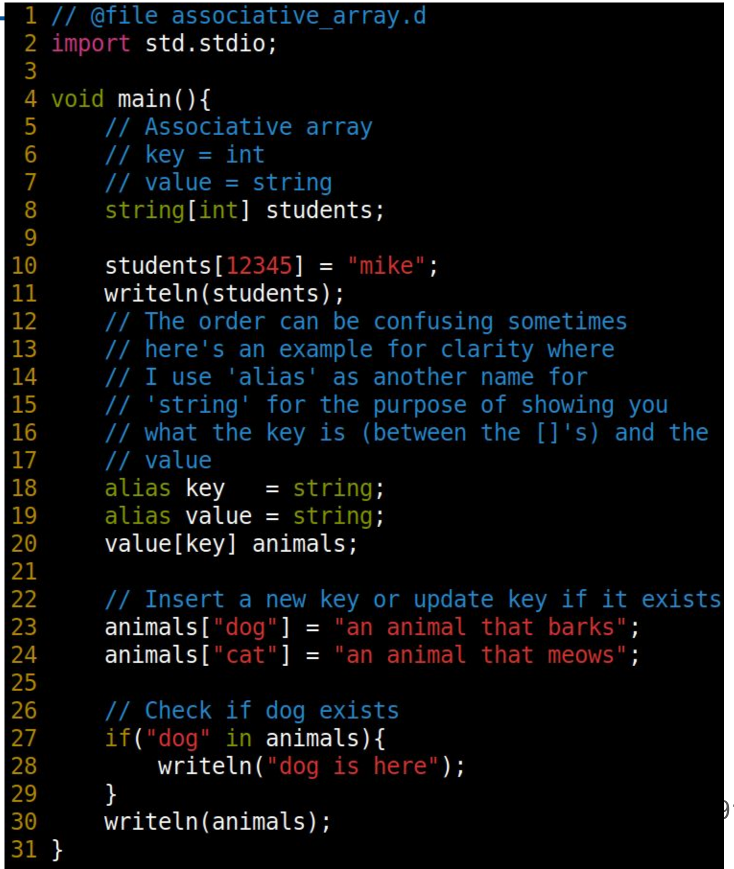


```d
// @file dynamic_array.d
import std.stdio;

void main(){

        // Create a dynamic array
        // by not specifying the size;
        int[] dynamic_array;
        dynamic_array ~= 5; // Append element 5
        writeln(dynamic_array);

        // initialize dynamic array with 10 elements to start
        int[] dyanmic_array_initial_size_10 = new int[10];
        dyanmic_array_initial_size_10 ~= 11;
        writeln(dyanmic_array_initial_size_10);

        // Multi-dimensional arrays
        int[][] multidimensional_dynamic_array = new int[][](2,2);
        writeln(multidimensional_dynamic_array);
}
```

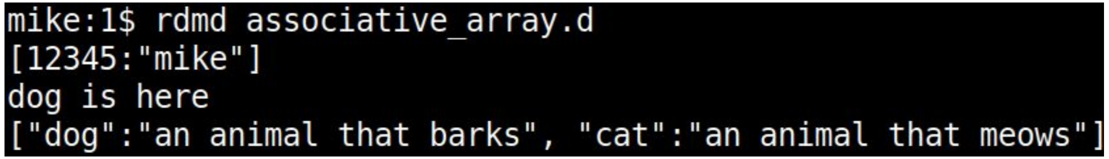
```
mike:1$ rdmd dynamic_array.d
[5]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 11]
[[0, 0], [0, 0]]
```

Associative Arrays (and sneak peak at alias)

- Associative Arrays
  - a.k.a dictionaries, hashmaps, hash tables
  - array
- https://tour.dlang.org/tour/en/basics/arrays



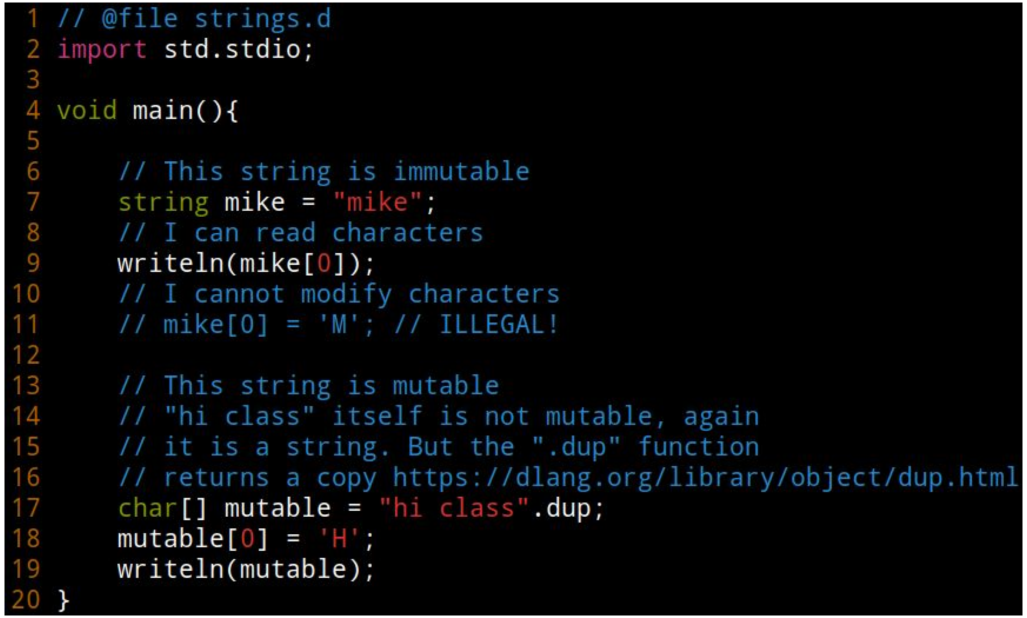
```d
// @file associative_array.d
import std.stdio;

void main(){
    // Associative array
    // key = int
    // value = string
    string[int] students;

    students[12345] = "mike";
    writeln(students);
    // The order can be confusing sometimes
    // here's an example for clarity where
    // I use 'alias' as another name for
    // 'string' for the purpose of showing you
    // what the key is (between the []'s) and the
    // value
    alias key   = string;
    alias value = string;
    value[key] animals;

    // Insert a new key or update key if it exists
    animals["dog"] = "an animal that barks";
    animals["cat"] = "an animal that meows";

    // Check if dog exists
    if("dog" in animals){
        writeln("dog is here");
    }
    writeln(animals);
}
```



```
mike:1$ rdmd associative_array.d
[12345:"mike"]
dog is here
["dog":"an animal that barks", "cat":"an animal that meows"]
```

DLang strings and char[]

- In DLang, strings are again
  - alias string = immutable(char)[];
    - That means we cannot change strings
  - If we want a string that we can modify, just make it an array of characters
    - i.e. char[] mutable_string = “Hello friends”;
  - See https://tour.dlang.org/tour/en/basics/alias-strings

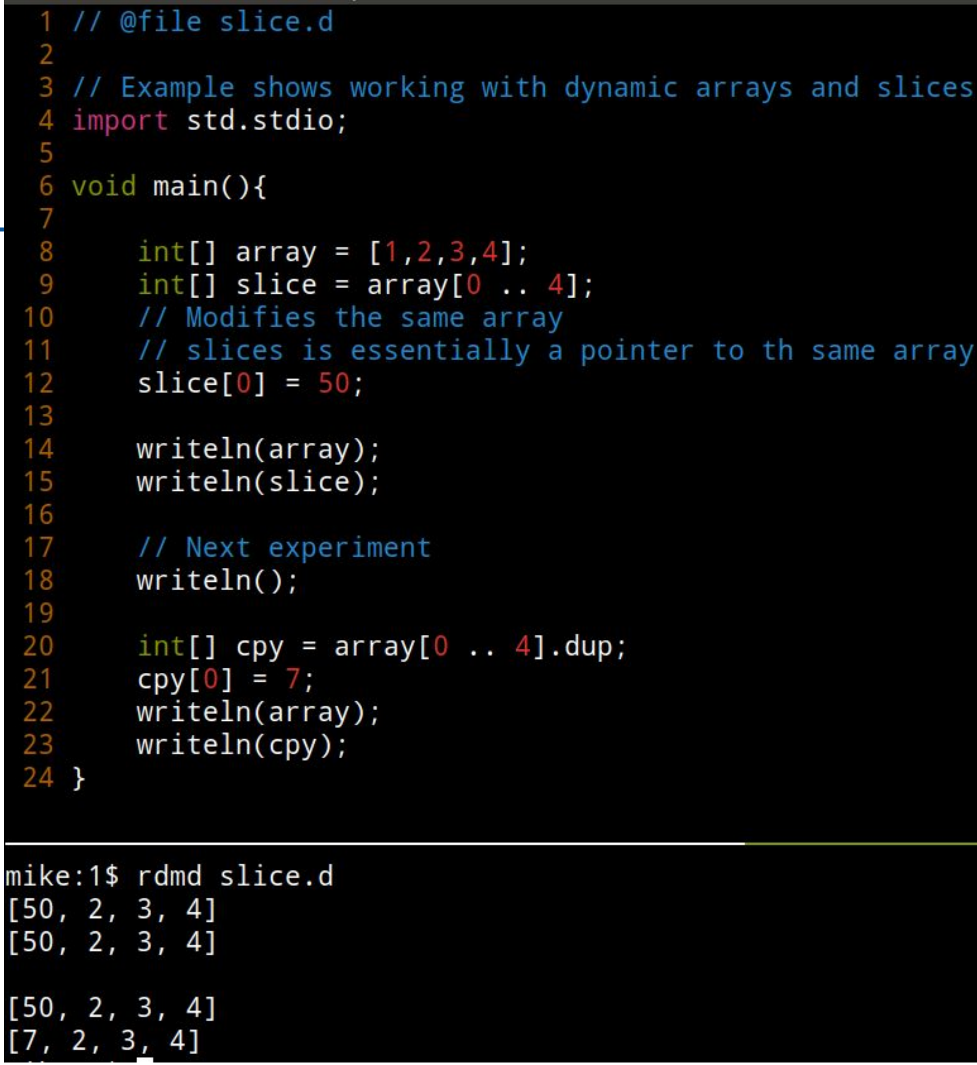


```d
// @file strings.d
import std.stdio;

void main(){

    // This string is immutable
    string mike = "mike";
    // I can read characters
    writeln(mike[0]);
    // I cannot modify characters
    // mike[0] = 'M'; // ILLEGAL!

    // This string is mutable
    // "hi class" itself is not mutable, again
    // it is a string. But the ".dup" function
    // returns a copy https://dlang.org/library/object/dup.html
    char[] mutable = "hi class".dup;
    mutable[0] = 'H';
    writeln(mutable);
}
```

dup example https://dlang.org/library/object/dup.html

Slices

- Slices themselves point to already existing memory
  - "a **view** into memory"
- Very quick way to get a few into data
  - Again -- use .dup if you want to initialize a new array with its own copy of previous data
  - Can use $ as a shortcut for end of collection
  - https://tour.dlang.org/tour/en/basics/slices
  - Great article on slices
    - https://dlang.org/articles/d-array-article.html

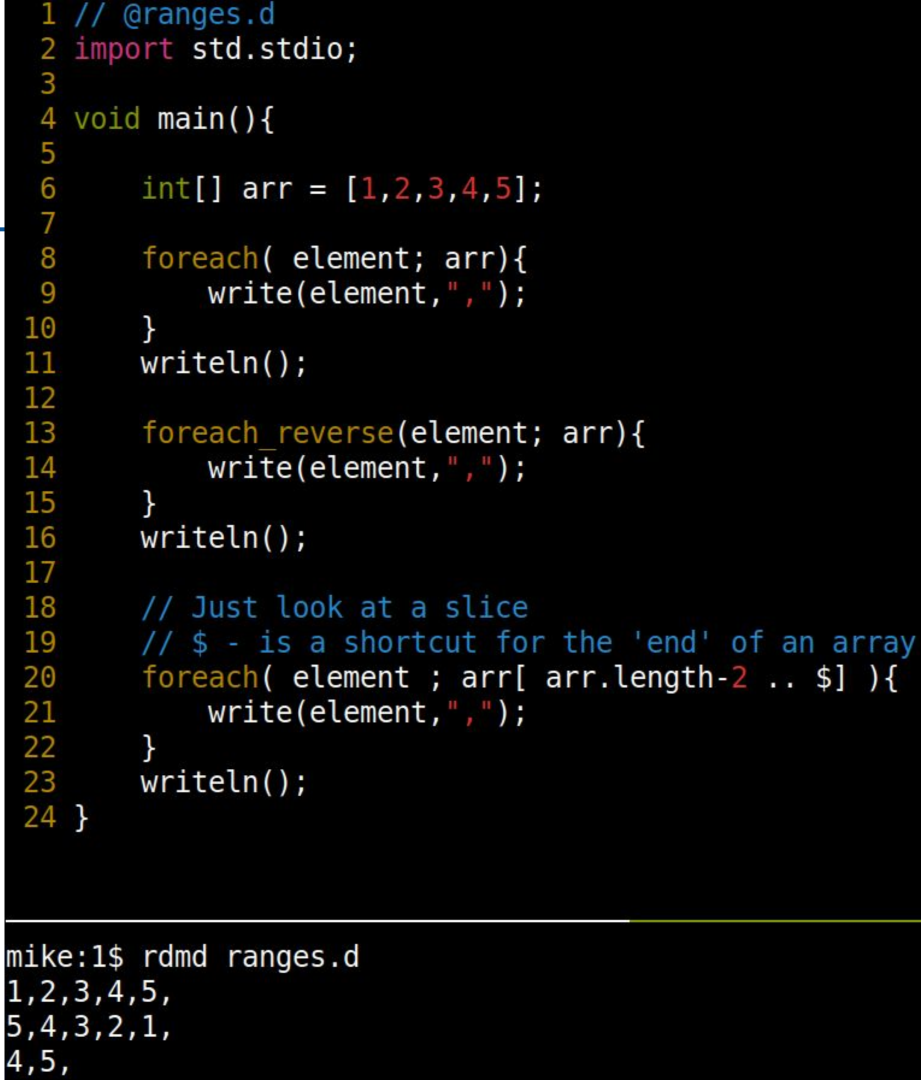


```
// @file slice.d

// Example shows working with dynamic arrays and slices
import std.stdio;

void main(){

    int[] array = [1,2,3,4];
    int[] slice = array[0 .. 4];
    // Modifies the same array
    // slices is essentially a pointer to th same array
    slice[0] = 50;

    writeln(array);
    writeln(slice);

    // Next experiment
    writeln();

    int[] cpy = array[0 .. 4].dup;
    cpy[0] = 7;
    writeln(array);
    writeln(cpy);
}
```

```
mike:1$ rdmd slice.d
[50, 2, 3, 4]
[50, 2, 3, 4]

[50, 2, 3, 4]
[7, 2, 3, 4]
```

Ranges

- A very brief introduction into the idea of ‘ranges’ in the D language.
  - https://tour.dlang.org/tour/en/basics/ranges
  - C++ 20 similarly has ranges
    - D however leans fully into ranges, meaning you do not see begin/end iterators in the standard library



```d
// @ranges.d
import std.stdio;

void main(){

    int[] arr = [1,2,3,4,5];

    foreach( element; arr){
        write(element,",");
    }
    writeln();

    foreach_reverse(element; arr){
        write(element,",");
    }
    writeln();

    // Just look at a slice
    // $ - is a shortcut for the 'end' of an array
    foreach( element ; arr[ arr.length-2 .. $] ){
        write(element,",");
    }
    writeln();
}
```

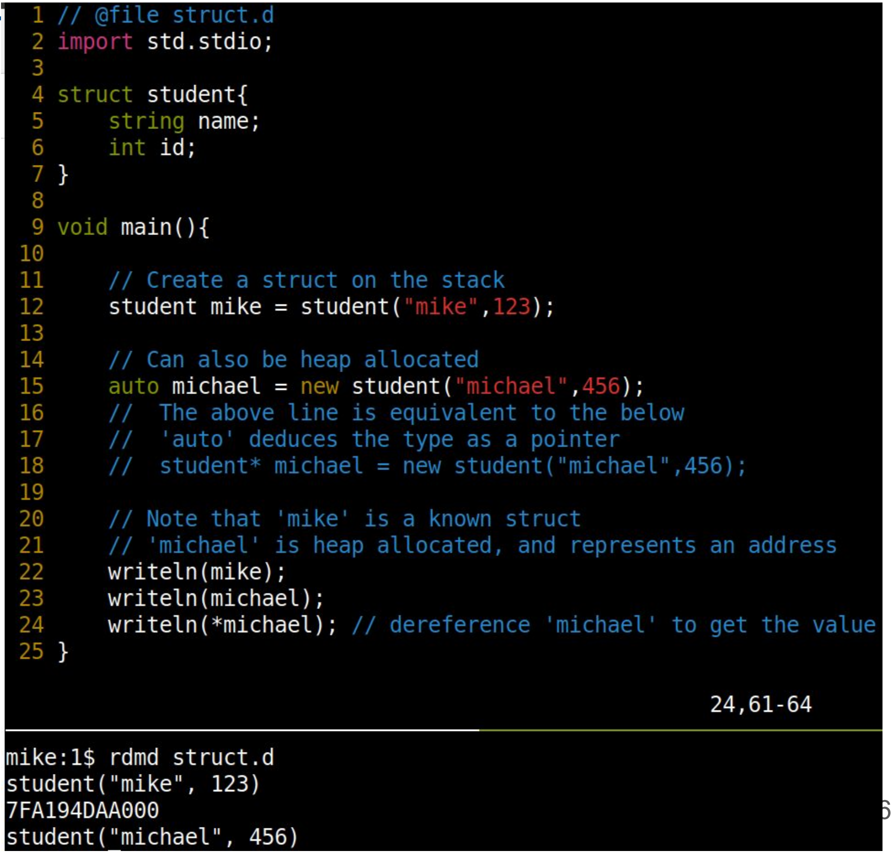
```
mike:1$ rdmd ranges.d
1,2,3,4,5,
5,4,3,2,1,
4,5,
```

Multi-Paradigm

- D supports procedural
  - Object Oriented
  - Functional
  - Generic
  - Multi-threaded
  - Parallel
  - etc.

Object-Oriented Programming Paradigm: structs

- structs are aggregate types, made up of 1 or more other types
- structs are known as 'value types'
  - They are by default stack allocated
    - (Can be heap allocated with 'new' however)
  - They do not allow for inheritance however
    - (The type is final)
  - This distinguishment can often be tricky for new programmers, but it's a good "design decision" that you have to make up front when writing code.



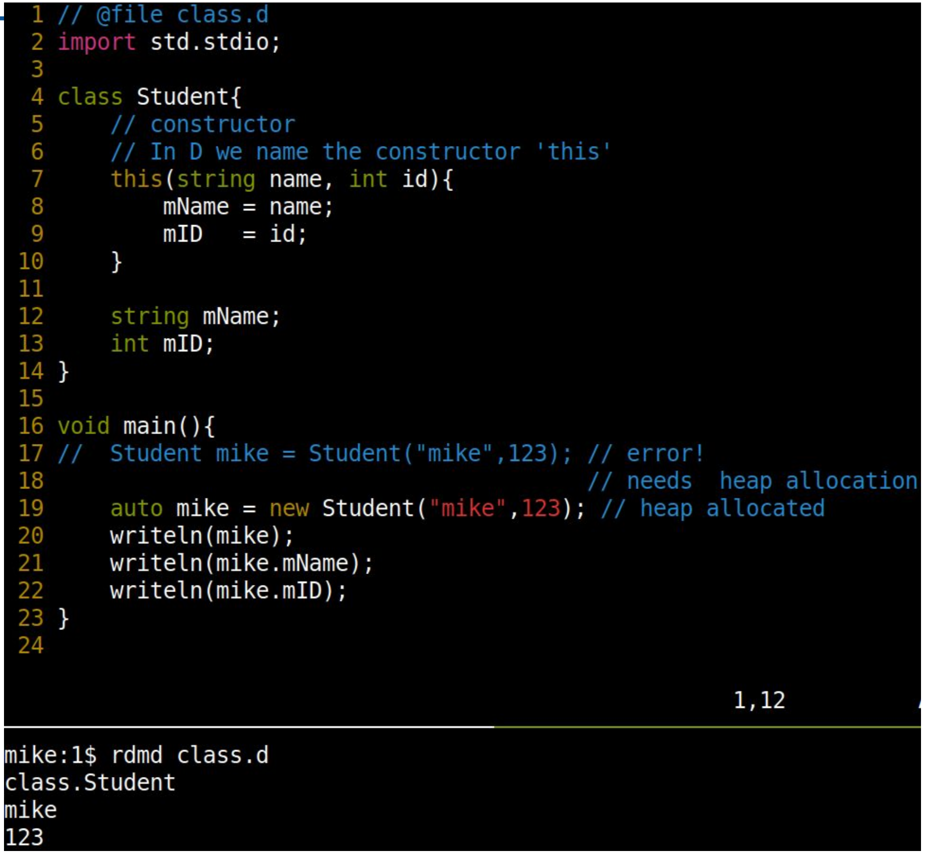
```d
// @file struct.d
import std.stdio;

struct student{
    string name;
    int id;
}

void main(){

    // Create a struct on the stack
    student mike = student("mike",123);

    // Can also be heap allocated
    auto michael = new student("michael",456);
    //  The above line is equivalent to the below
    //  'auto' deduces the type as a pointer
    //  student* michael = new student("michael",456);

    // Note that 'mike' is a known struct
    // 'michael' is heap allocated, and represents an address
    writeln(mike);
    writeln(michael);
    writeln(*michael); // dereference 'michael' to get the value
}
```

```
                                                    24,61-64
```

```
mike:1$ rdmd struct.d
student("mike", 123)
7FA194DAA000
student("michael", 456)
```

Object-Oriented Programming Paradigm: Classes

- Classes are accessed solely by references (i.e. they must be dynamically allocated)
  - They're meant for dynamic polymorphism.
  - D is similar to Java in how it utilizes classes for inheritance
    - D supports single inheritance of implementation
    - D supports multiple inheritance using interfaces



```d
// @file class.d
import std.stdio;

class Student{
    // constructor
    // In D we name the constructor 'this'
    this(string name, int id){
        mName = name;
        mID   = id;
    }

    string mName;
    int mID;
}

void main(){
//  Student mike = Student("mike",123); // error!
                                        // needs  heap allocation
    auto mike = new Student("mike",123); // heap allocated
    writeln(mike);
    writeln(mike.mName);
    writeln(mike.mID);
}

                                                              1,12
```

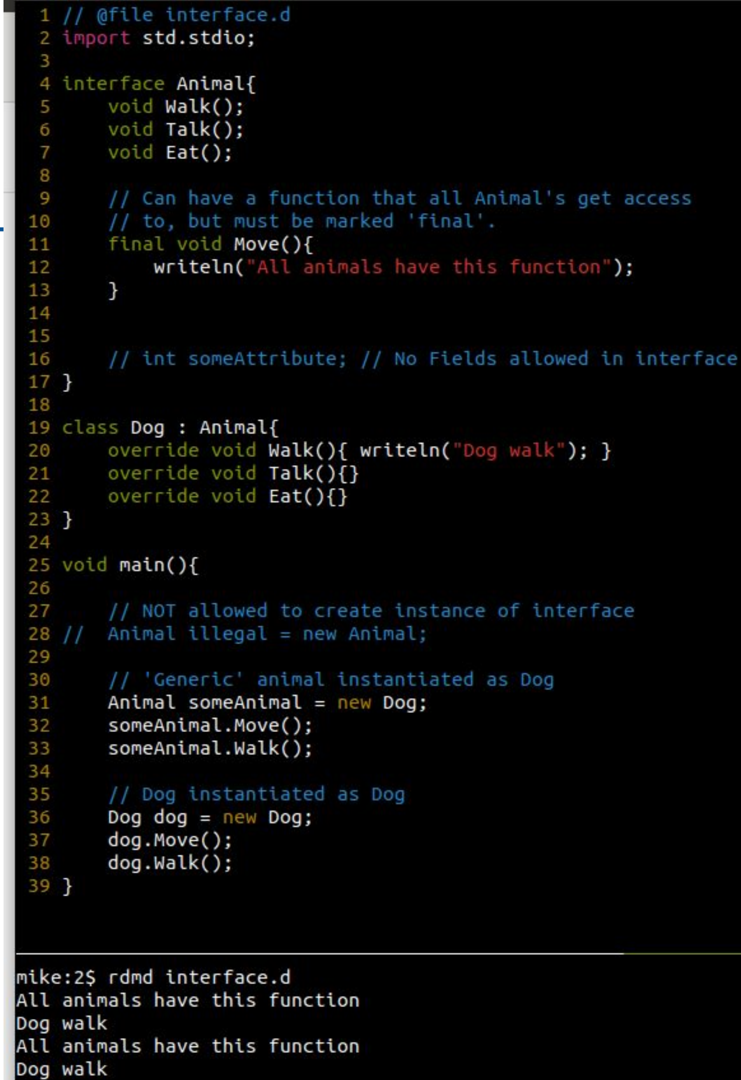
```
mike:1$ rdmd class.d
class.Student
mike
123
```

Struct versus classes

- In some languages there is no distinguishment, but in more modern languages the distinguishment between structs and classes is important
    - You decide up front if a type can be inherited from (i.e. by using a class)

Classes and Interfaces

- An **interface** in D provides a ‘blueprint’ that classes must inherit from
  - We cannot otherwise create an instance of ‘Animal’ from the example on the right.

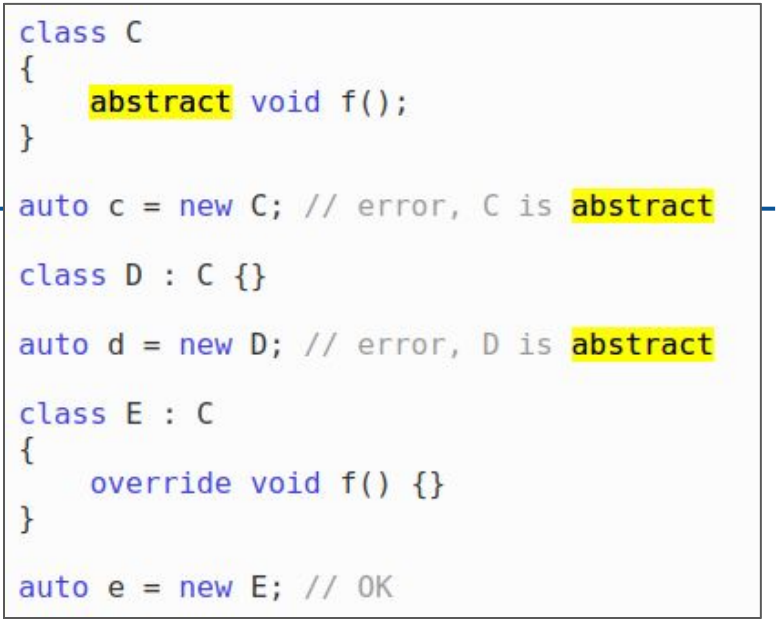


```d
// @file interface.d
import std.stdio;

interface Animal{
    void Walk();
    void Talk();
    void Eat();

    // Can have a function that all Animal's get access
    // to, but must be marked 'final'.
    final void Move(){
        writeln("All animals have this function");
    }


    // int someAttribute; // No Fields allowed in interface
}

class Dog : Animal{
    override void Walk(){ writeln("Dog walk"); }
    override void Talk(){}
    override void Eat(){}
}

void main(){

    // NOT allowed to create instance of interface
//  Animal illegal = new Animal;

    // 'Generic' animal instantiated as Dog
    Animal someAnimal = new Dog;
    someAnimal.Move();
    someAnimal.Walk();

    // Dog instantiated as Dog
    Dog dog = new Dog;
    dog.Move();
    dog.Walk();
}
```

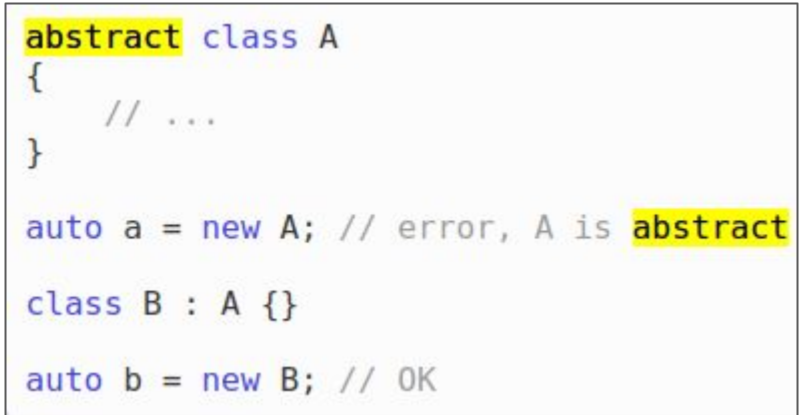
```
mike:2$ rdmd interface.d
All animals have this function
Dog walk
All animals have this function
Dog walk
```

abstract class (1/2)

- Note: In D we also have abstract classes
  - https://dlang.org/spec/class.html#abstract
  - These are similar to interfaces, but you can also add properties (i.e. member variables)
  - Same rules as an interface however -- cannot instantiate a class marked as abstract or with members that are abstract.



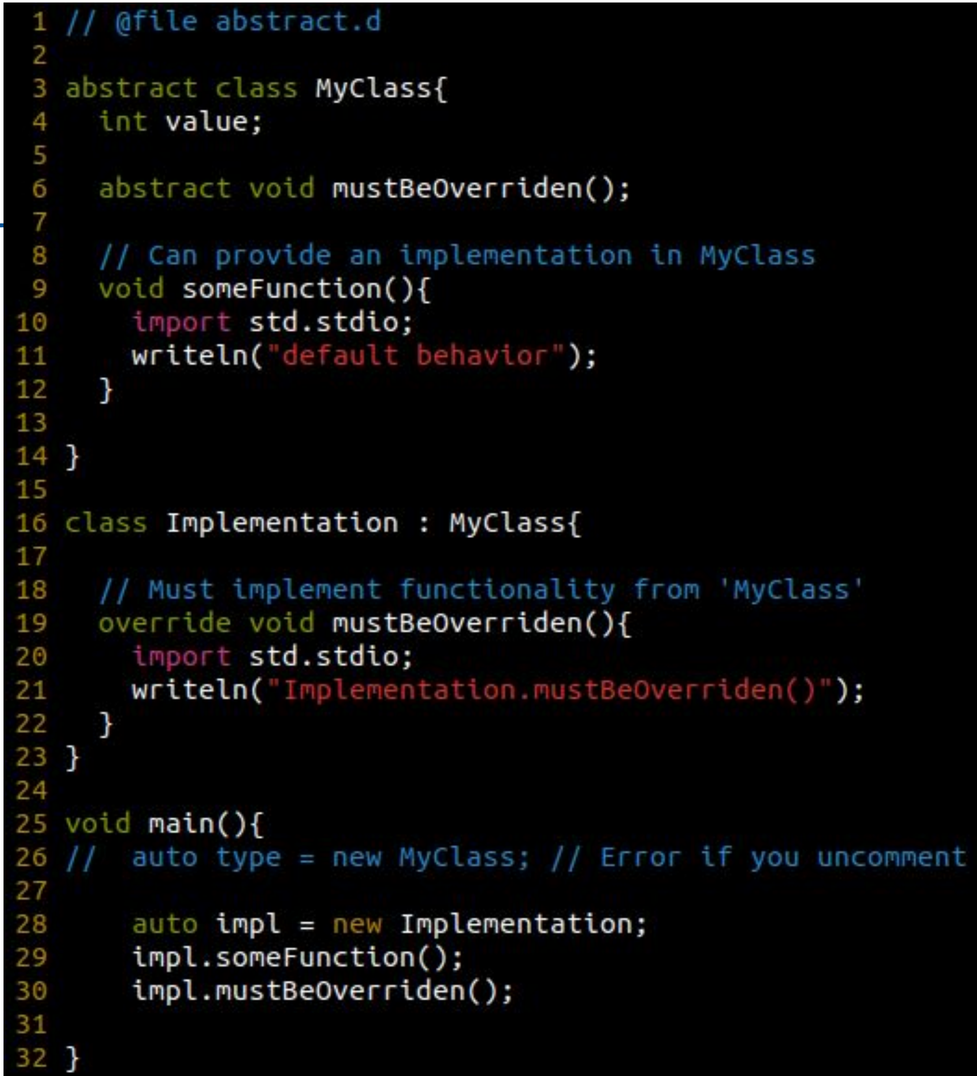
```
class C
{
    abstract void f();
}

auto c = new C; // error, C is abstract

class D : C {}

auto d = new D; // error, D is abstract

class E : C
{
    override void f() {}
}

auto e = new E; // OK
```



```
abstract class A
{
    // ...
}

auto a = new A; // error, A is abstract

class B : A {}

auto b = new B; // OK
```

abstract class (2/2)

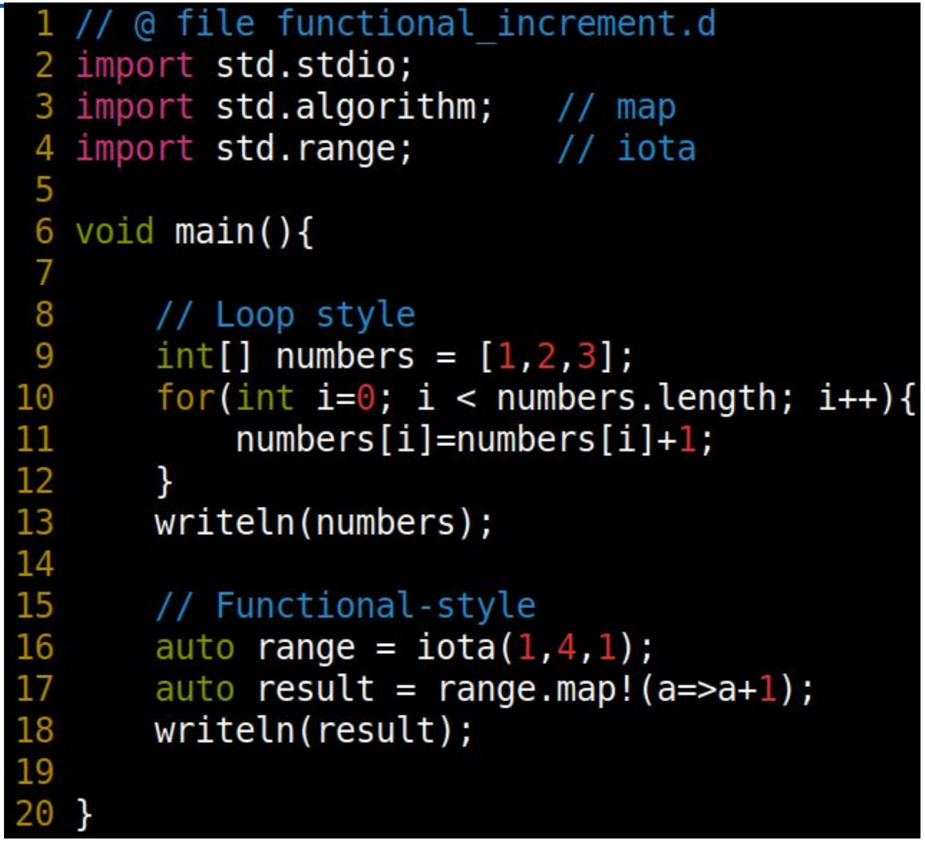
- Here’s an example
    - Think of ‘abstract classes’ as a way to provide a more powerful interface, if you think there is some default functionality that is needed in a class.
    - (Note: Another way to achieve this in an Interface is by providing ‘final’ functions, but you still cannot have any ‘state’ in in interface (i.e. no member variables)



```d
// @file abstract.d

abstract class MyClass{
  int value;

  abstract void mustBeOverriden();

  // Can provide an implementation in MyClass
  void someFunction(){
    import std.stdio;
    writeln("default behavior");
  }

}

class Implementation : MyClass{

  // Must implement functionality from 'MyClass'
  override void mustBeOverriden(){
    import std.stdio;
    writeln("Implementation.mustBeOverriden()");
  }
}

void main(){
//   auto type = new MyClass; // Error if you uncomment

    auto impl = new Implementation;
    impl.someFunction();
    impl.mustBeOverriden();

}
```

Code Analysis - Loop versus ‘map’

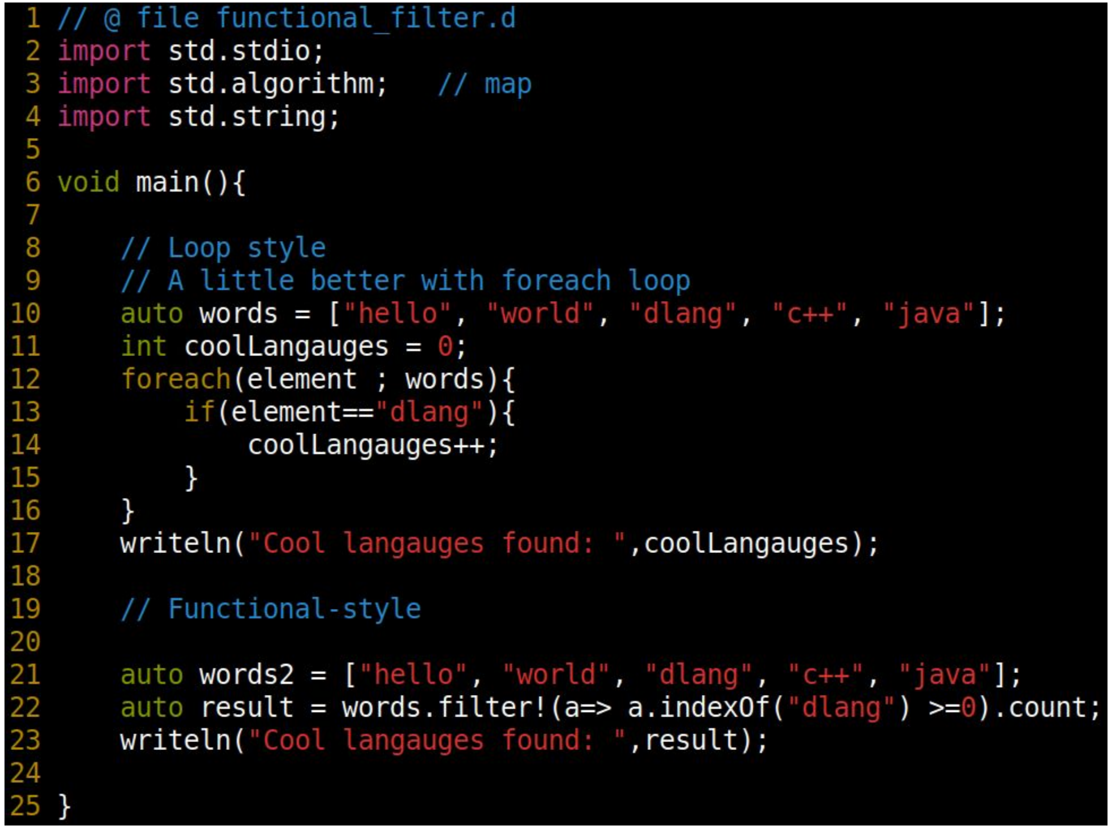
- Let’s take a look at this piece of code
  - See if you can figure out the results that will be written out
  - **map** again may be new for folks who haven’t done functional programming
    - But I assure you -- the ‘Loop style’ and ‘Functional-style’ will generate the same result.
- (Note: You could write this as a line-line function:
  - `iota(1,4,1).map!(a=>a+1).writeln;`



```d
// @ file functional_increment.d
import std.stdio;
import std.algorithm;    // map
import std.range;        // iota

void main(){

    // Loop style
    int[] numbers = [1,2,3];
    for(int i=0; i < numbers.length; i++){
        numbers[i]=numbers[i]+1;
    }
    writeln(numbers);

    // Functional-style
    auto range = iota(1,4,1);
    auto result = range.map!(a=>a+1);
    writeln(result);

}
```

Code Analysis - Loop versus ‘filter’

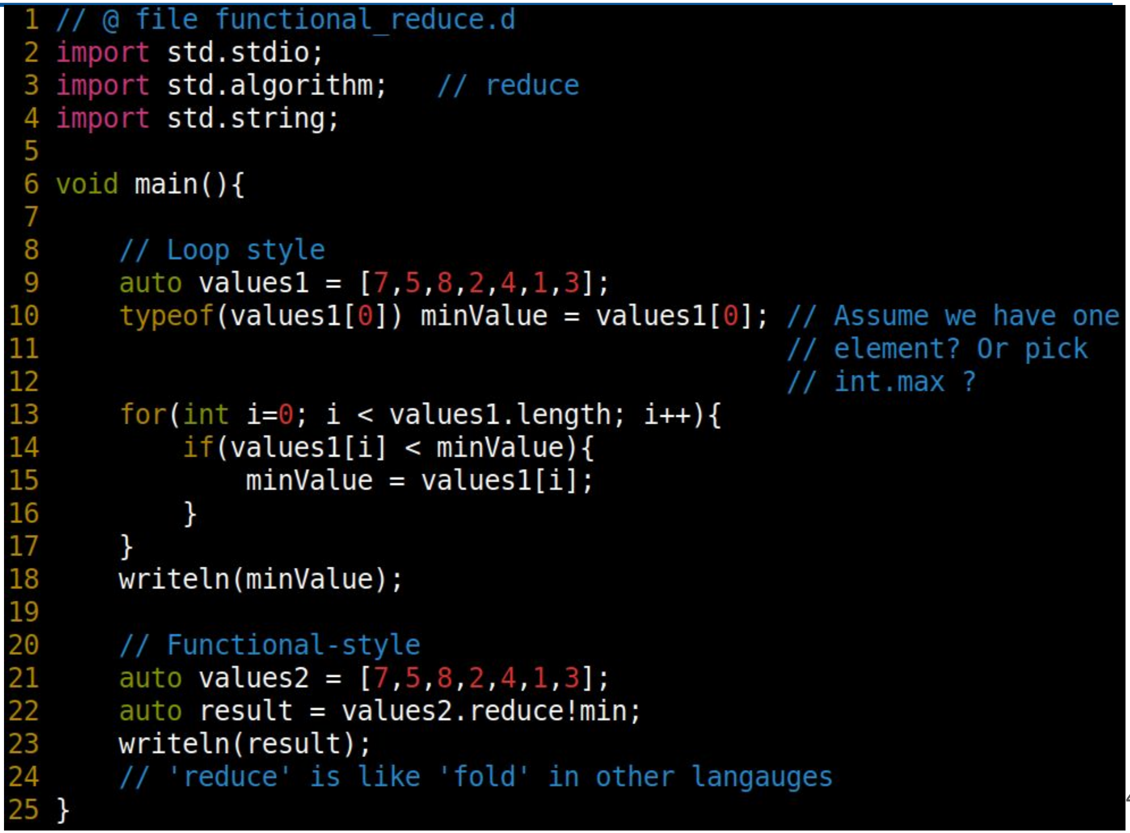
- Let’s take a look at this one -- **filter**
  - Again lines 10-17 represents one experiment
  - Lines 21-23 represent the functional style.



```
 1 // @ file functional_filter.d
 2 import std.stdio;
 3 import std.algorithm;   // map
 4 import std.string;
 5
 6 void main(){
 7
 8     // Loop style
 9     // A little better with foreach loop
10     auto words = ["hello", "world", "dlang", "c++", "java"];
11     int coolLangauges = 0;
12     foreach(element ; words){
13         if(element=="dlang"){
14             coolLangauges++;
15         }
16     }
17     writeln("Cool langauges found: ",coolLangauges);
18
19     // Functional-style
20
21     auto words2 = ["hello", "world", "dlang", "c++", "java"];
22     auto result = words.filter!(a=> a.indexOf("dlang") >=0).count;
23     writeln("Cool langauges found: ",result);
24
25 }
```

Code Analysis - Loop versus ‘reduce’

- Observe again the same experiment -- **reduce**
  - Which code has fewer branches?
  - Which code has fewer decisions?

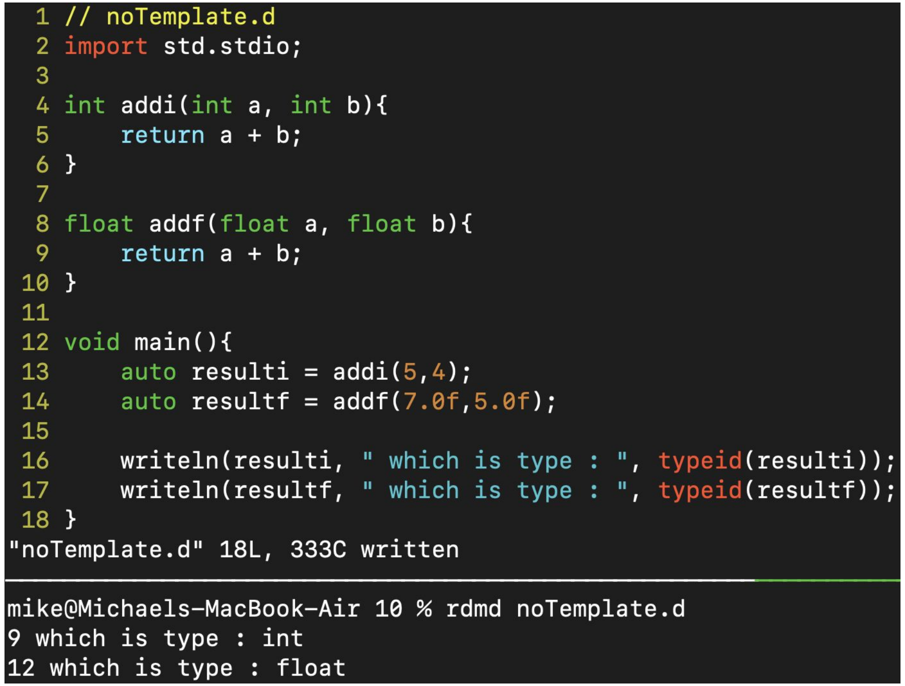


```d
// @ file functional_reduce.d
import std.stdio;
import std.algorithm;   // reduce
import std.string;

void main(){

    // Loop style
    auto values1 = [7,5,8,2,4,1,3];
    typeof(values1[0]) minValue = values1[0]; // Assume we have one
                                              // element? Or pick
                                              // int.max ?
    for(int i=0; i < values1.length; i++){
        if(values1[i] < minValue){
            minValue = values1[i];
        }
    }
    writeln(minValue);

    // Functional-style
    auto values2 = [7,5,8,2,4,1,3];
    auto result = values2.reduce!min;
    writeln(result);
    // 'reduce' is like 'fold' in other langauges
}
```

D Language - Templates

Types

- D is a statically typed language
    - This means that at compile-time, symbols (i.e. variables and functions) store data in a format (i.e. integer, float, etc.) that does not change.
- What this means is, we often have to write different variations of functions to handle different inputs.
    - Observe the two different ‘add’ functions (addi and addf) to the right with different types for the parameters.

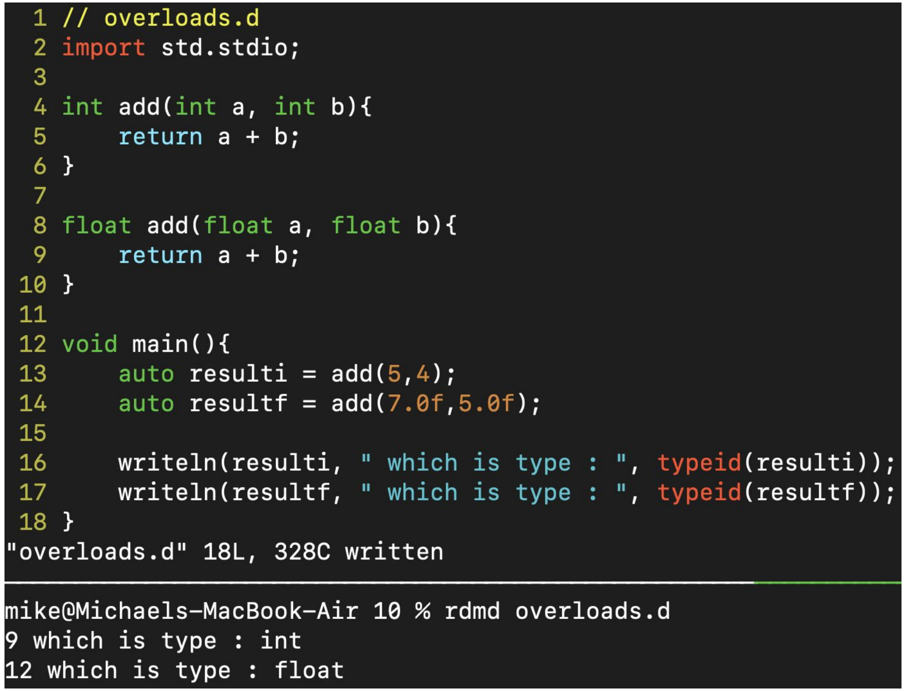


```
// noTemplate.d
import std.stdio;

int addi(int a, int b){
    return a + b;
}

float addf(float a, float b){
    return a + b;
}

void main(){
    auto resulti = addi(5,4);
    auto resultf = addf(7.0f,5.0f);

    writeln(resulti, " which is type : ", typeid(resulti));
    writeln(resultf, " which is type : ", typeid(resultf));
}
"noTemplate.d" 18L, 333C written
```

```
mike@Michaels-MacBook-Air 10 % rdmd noTemplate.d
9 which is type : int
12 which is type : float
```

Function Overloads (1/2)

- Note that languages like D (and C++, Java) support function overloading, such that every function does not have to be uniquely named.
    - This means that we can just name the function ‘add’,
    - The compiler is smart enough to deduce from the function signature (the name + parameter types) which version of the add function to call.

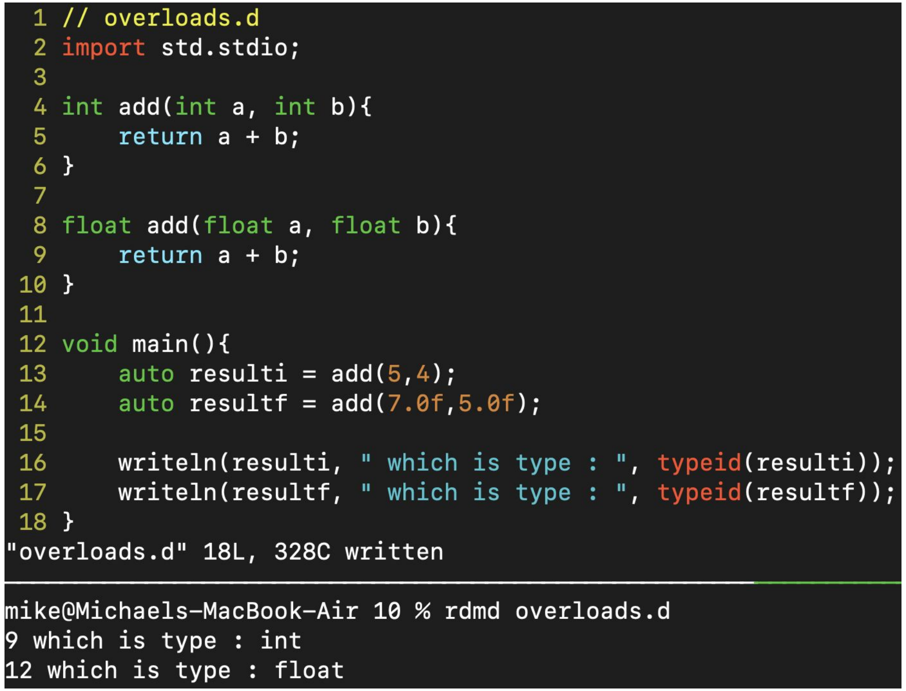


```d
// overloads.d
import std.stdio;

int add(int a, int b){
    return a + b;
}

float add(float a, float b){
    return a + b;
}

void main(){
    auto resulti = add(5,4);
    auto resultf = add(7.0f,5.0f);

    writeln(resulti, " which is type : ", typeid(resulti));
    writeln(resultf, " which is type : ", typeid(resultf));
}
"overloads.d" 18L, 328C written
```

```
mike@Michaels-MacBook-Air 10 % rdmd overloads.d
9 which is type : int
12 which is type : float
```

Function Overloads (2/2)

- Also observe in this example though, that it may become 'exhaustive' to type out every particular combination
  - This is where **templates** can be useful
  - We will however find out, that templates are even more useful of a mechanism for meta-programming

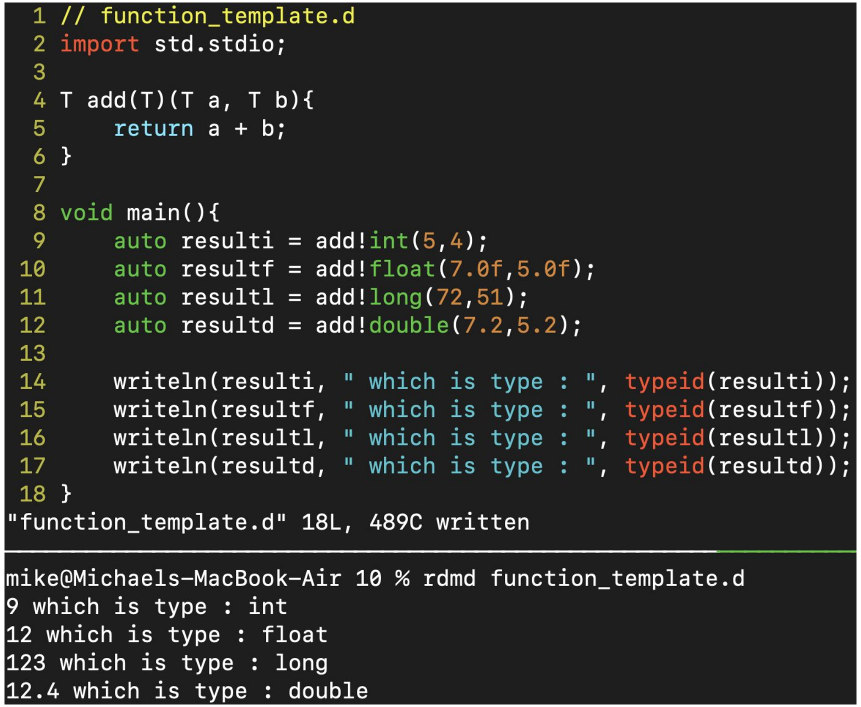


```d
// overloads.d
import std.stdio;

int add(int a, int b){
    return a + b;
}

float add(float a, float b){
    return a + b;
}

void main(){
    auto resulti = add(5,4);
    auto resultf = add(7.0f,5.0f);

    writeln(resulti, " which is type : ", typeid(resulti));
    writeln(resultf, " which is type : ", typeid(resultf));
}
"overloads.d" 18L, 328C written
```

```
mike@Michaels-MacBook-Air 10 % rdmd overloads.d
9 which is type : int
12 which is type : float
```

Templates - Function Template

- Templates are a mechanism for generating code.
  - One of the main uses of templates is for enabling **generic programming paradigm**.
- Observe the code example on the right for writing a ‘generic’ or ‘templated’ version of add.
  - The first template parameter (T) after add is the type that will be replaced
  - T is also the return type.
  - Then when using our add function we use the **!** symbol to indicate the start of the template parameter we are supplying.

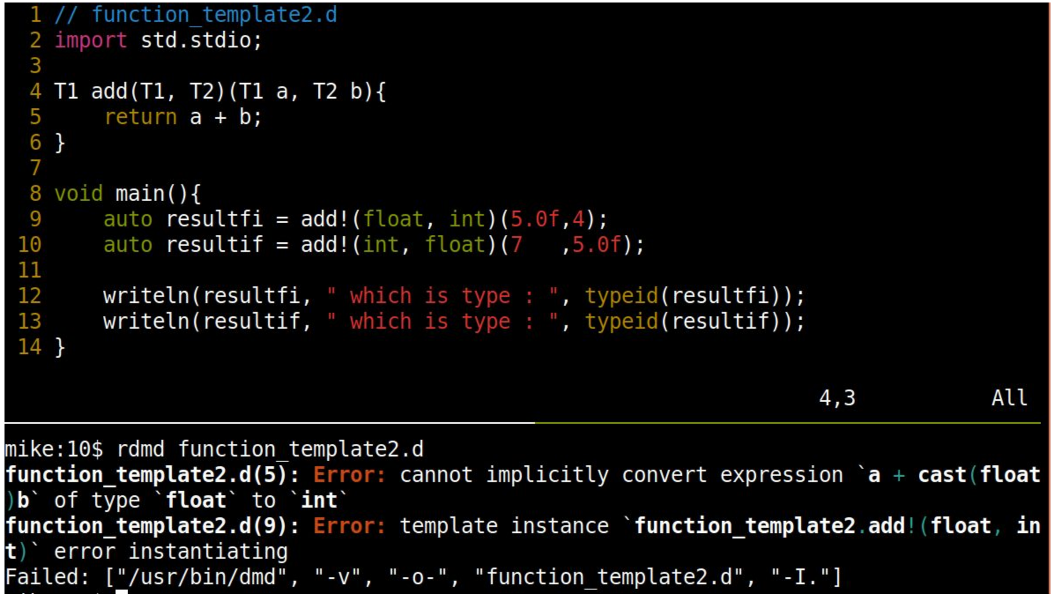


```
// function_template.d
import std.stdio;

T add(T)(T a, T b){
    return a + b;
}

void main(){
    auto resulti = add!int(5,4);
    auto resultf = add!float(7.0f,5.0f);
    auto resultl = add!long(72,51);
    auto resultd = add!double(7.2,5.2);

    writeln(resulti, " which is type : ", typeid(resulti));
    writeln(resultf, " which is type : ", typeid(resultf));
    writeln(resultl, " which is type : ", typeid(resultl));
    writeln(resultd, " which is type : ", typeid(resultd));
}
"function_template.d" 18L, 489C written
```

```
mike@Michaels-MacBook-Air 10 % rdmd function_template.d
9 which is type : int
12 which is type : float
123 which is type : long
12.4 which is type : double
```

Multiple template parameters

- Note, you can provide as many template parameters as is reasonable -- in this case I show two
  - One issue however, is for figuring out the return type.
  - Should it be T1 or T2?
    - In this case, it can be neither as D avoids implicitly converting the types.
    - **See next slide for solution**

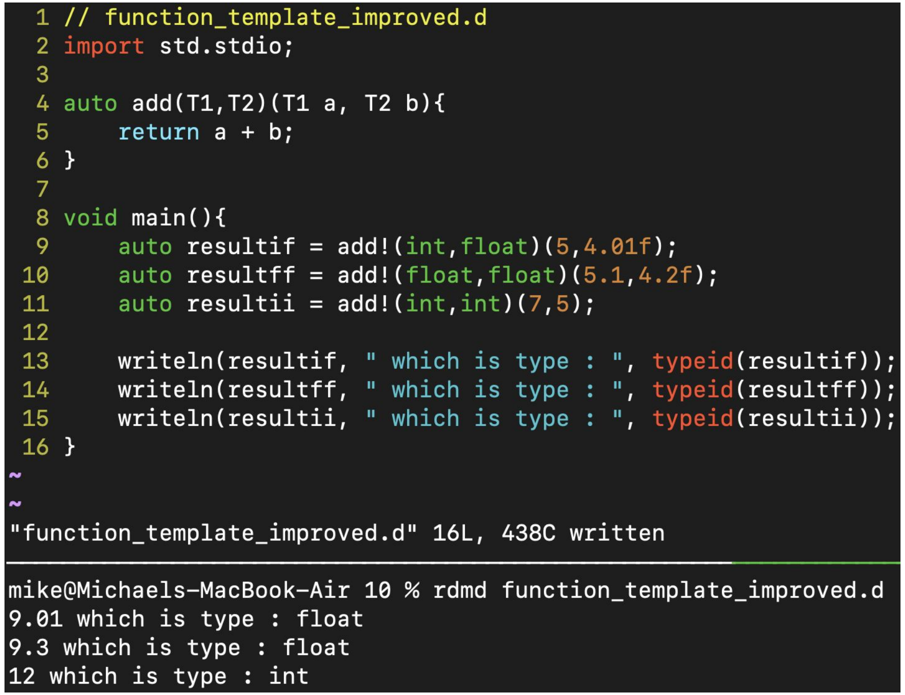


```
// function_template2.d
import std.stdio;

T1 add(T1, T2)(T1 a, T2 b){
    return a + b;
}

void main(){
    auto resultfi = add!(float, int)(5.0f,4);
    auto resultif = add!(int, float)(7   ,5.0f);

    writeln(resultfi, " which is type : ", typeid(resultfi));
    writeln(resultif, " which is type : ", typeid(resultif));
}
                                                4,3           All
```

```
mike:10$ rdmd function_template2.d
function_template2.d(5): Error: cannot implicitly convert expression `a + cast(float
)b` of type `float` to `int`
function_template2.d(9): Error: template instance `function_template2.add!(float, in
t)` error instantiating
Failed: ["/usr/bin/dmd", "-v", "-o-", "function_template2.d", "-I."]
```

Deducing the return type with ‘auto’

- D allows for us to use ‘auto’ to deduce the return type.
    - This is probably the cleanest way to do things if you have an operation (i.e. ‘+’) that may mix types.

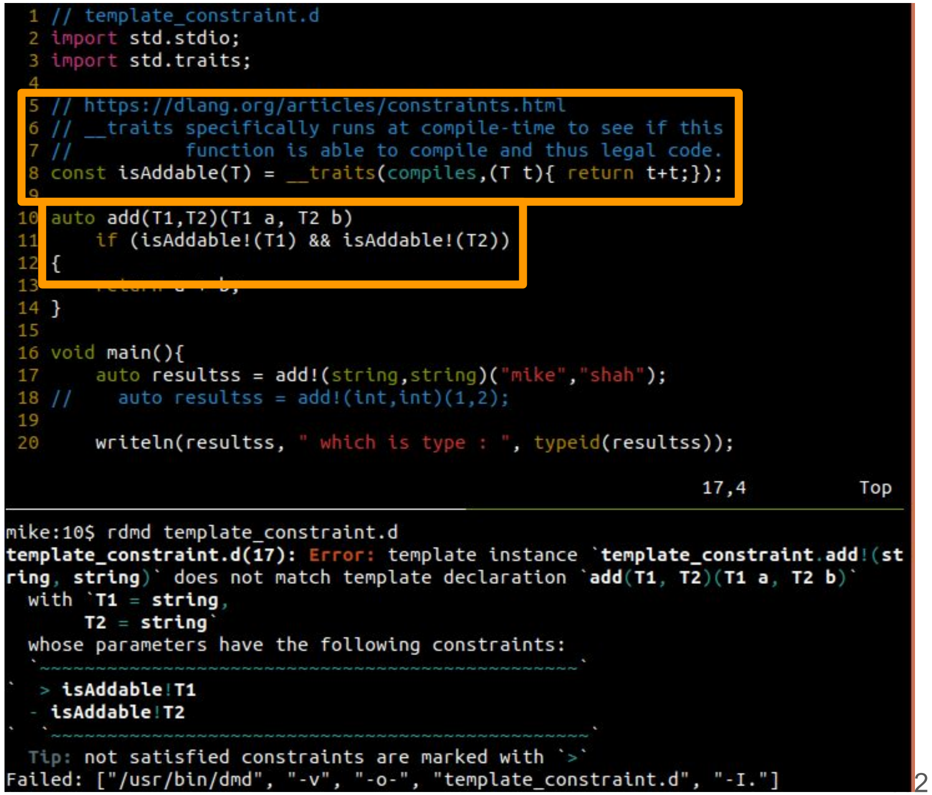


```d
// function_template_improved.d
import std.stdio;

auto add(T1,T2)(T1 a, T2 b){
    return a + b;
}

void main(){
    auto resultif = add!(int,float)(5,4.01f);
    auto resultff = add!(float,float)(5.1,4.2f);
    auto resultii = add!(int,int)(7,5);

    writeln(resultif, " which is type : ", typeid(resultif));
    writeln(resultff, " which is type : ", typeid(resultff));
    writeln(resultii, " which is type : ", typeid(resultii));
}
~
~
"function_template_improved.d" 16L, 438C written
```

```
mike@Michaels-MacBook-Air 10 % rdmd function_template_improved.d
9.01 which is type : float
9.3 which is type : float
12 which is type : int
```

Template Specialization - Template Constraints

- One option is to provide a template constraint
  - This is also known as a ‘concept’ in Modern C++
- The idea is that we have checks on a function to ensure that some rule(s) is being followed.
  - We can have as many constraints as we like
- Note: We’ll look at traits shortly.

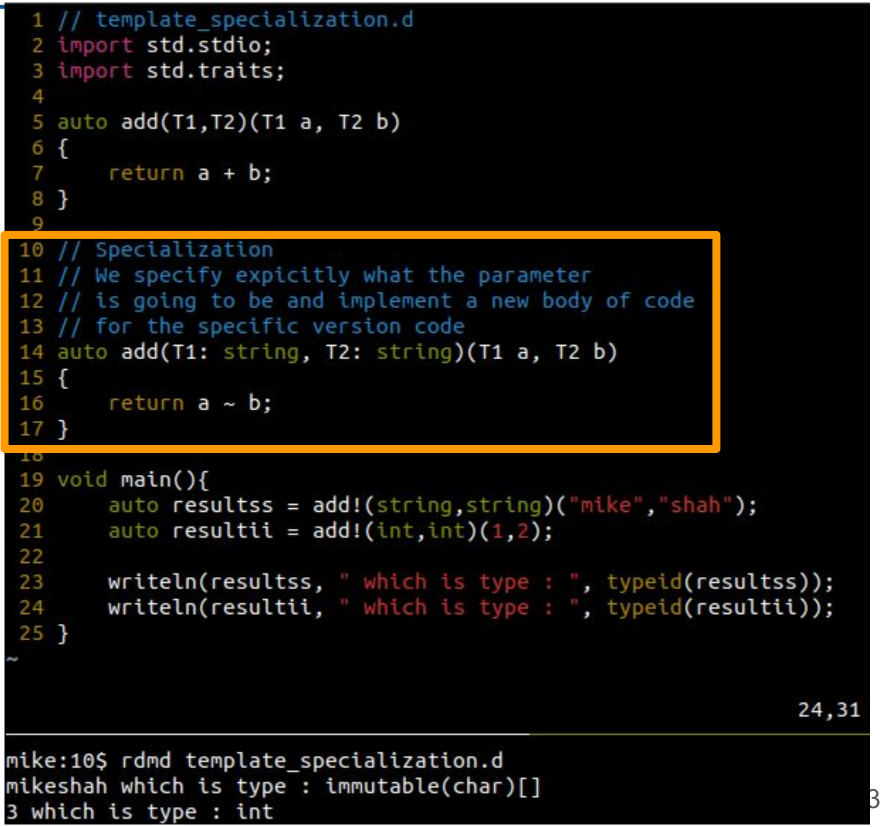


```
 1 // template_constraint.d
 2 import std.stdio;
 3 import std.traits;
 4
 5 // https://dlang.org/articles/constraints.html
 6 // __traits specifically runs at compile-time to see if this
 7 //              function is able to compile and thus legal code.
 8 const isAddable(T) = __traits(compiles,(T t){ return t+t;});
 9
10 auto add(T1,T2)(T1 a, T2 b)
11     if (isAddable!(T1) && isAddable!(T2))
12 {
13     return a + b;
14 }
15
16 void main(){
17     auto resultss = add!(string,string)("mike","shah");
18 //     auto resultss = add!(int,int)(1,2);
19
20     writeln(resultss, " which is type : ", typeid(resultss));
                                                           17,4          Top
```

```
mike:10$ rdmd template_constraint.d
template_constraint.d(17): Error: template instance `template_constraint.add!(st
ring, string)` does not match template declaration `add(T1, T2)(T1 a, T2 b)`
  with `T1 = string,
       T2 = string`
  whose parameters have the following constraints:
  `~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`
`   > isAddable!T1
  - isAddable!T2
`  `~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`
  Tip: not satisfied constraints are marked with `>`
Failed: ["/usr/bin/dmd", "-v", "-o-", "template_constraint.d", "-I."]
```

Template Specialization - Specialization

- Observe that in this example we 'specialize' the template with 1 or all of the arguments
  - This selects the best match, and we can implement the body of hte code as appropriate
  - This time the specialization uses the correct concatenation operation

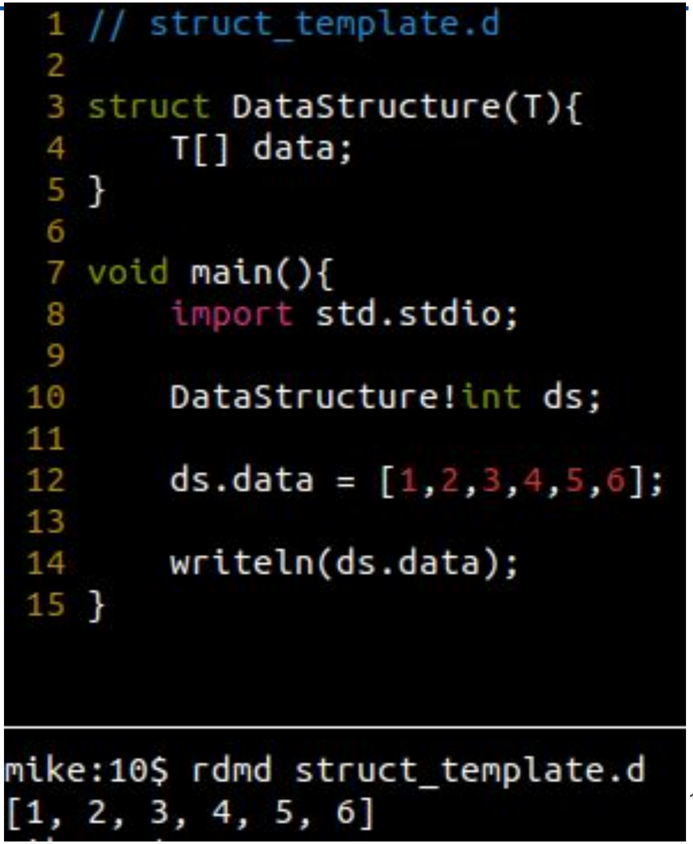


```d
// template_specialization.d
import std.stdio;
import std.traits;

auto add(T1,T2)(T1 a, T2 b)
{
    return a + b;
}

// Specialization
// We specify expicitly what the parameter
// is going to be and implement a new body of code
// for the specific version code
auto add(T1: string, T2: string)(T1 a, T2 b)
{
    return a ~ b;
}

void main(){
    auto resultss = add!(string,string)("mike","shah");
    auto resultii = add!(int,int)(1,2);

    writeln(resultss, " which is type : ", typeid(resultss));
    writeln(resultii, " which is type : ", typeid(resultii));
}
~
                                                        24,31
```

```
mike:10$ rdmd template_specialization.d
mikeshah which is type : immutable(char)[]
3 which is type : int
```

Templates for structs and classes

- Data structures can also be templated
    - This is in fact one of the best use cases of templates in my opinion, for making containers (i.e. data structures) that can store any type of data.

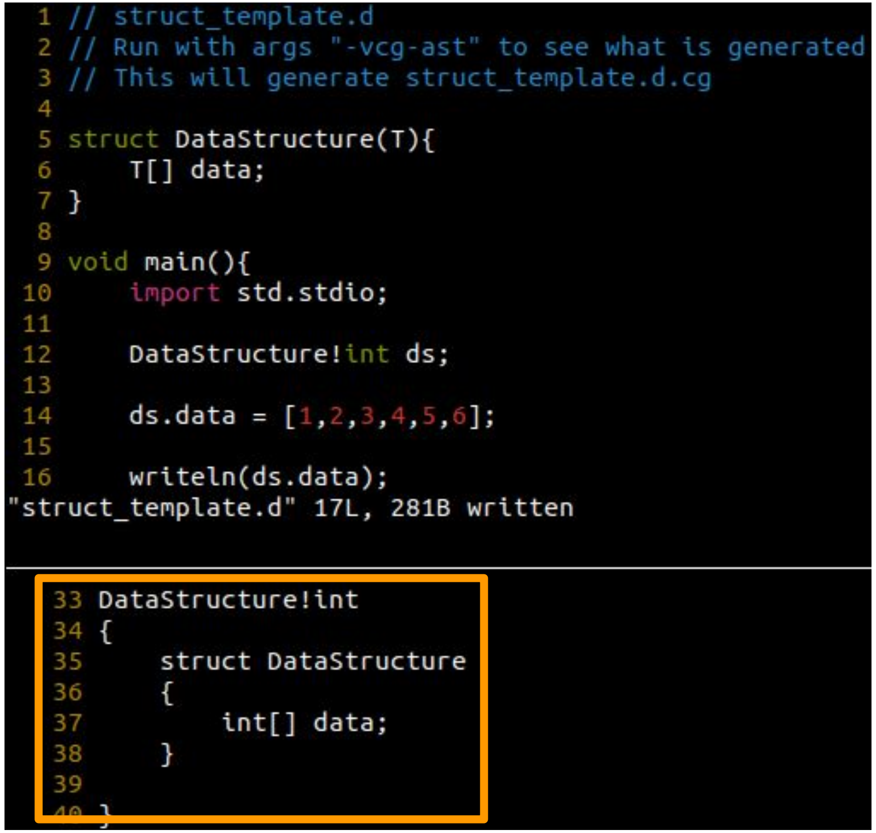


```d
// struct_template.d

struct DataStructure(T){
    T[] data;
}

void main(){
    import std.stdio;

    DataStructure!int ds;

    ds.data = [1,2,3,4,5,6];

    writeln(ds.data);
}
```

```
mike:10$ rdmd struct_template.d
[1, 2, 3, 4, 5, 6]
```

Generated Template Code

- It’s useful to explore the compiler and observe that a template is actually generating code for each instantiation.
  - For example, we have an explicit ‘DataStructure!int’ instantiation, where you can see that the templated data is in fact ‘int[]’
    - (i.e. the substitutions are performed for us based on the template arguments)

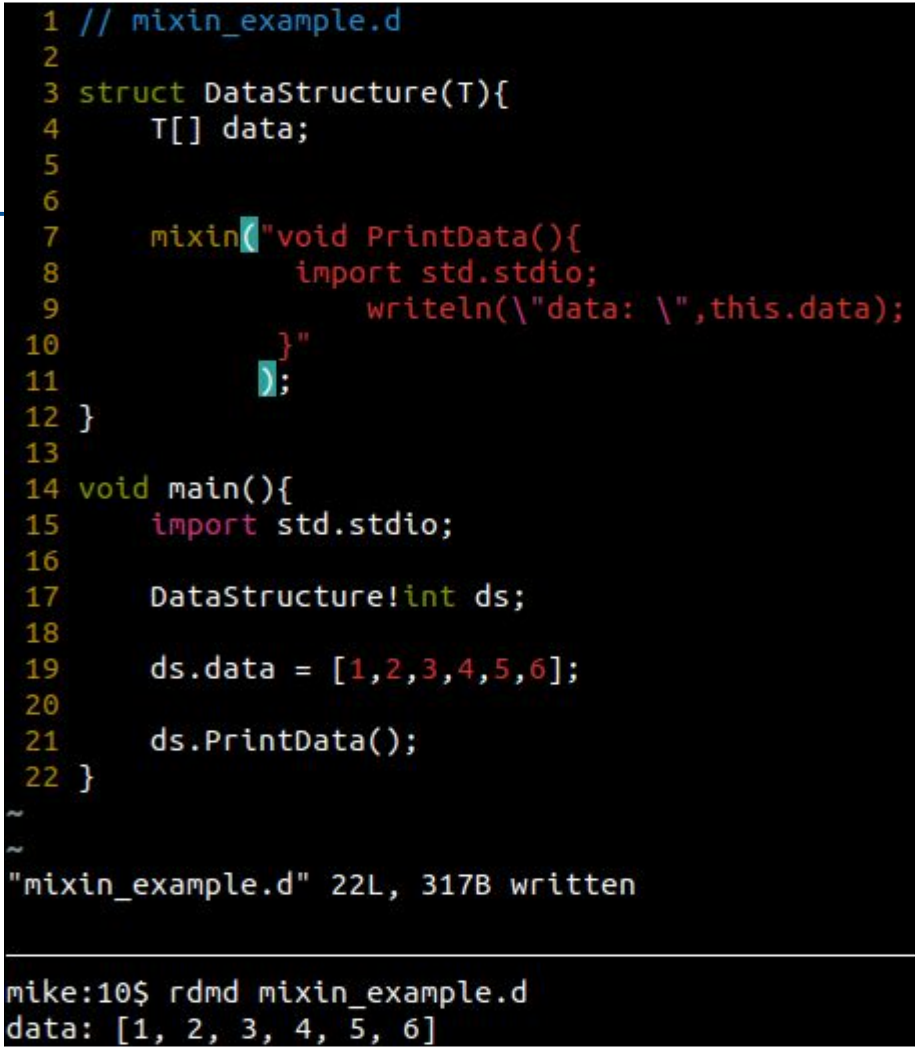


```
// struct_template.d
// Run with args "-vcg-ast" to see what is generated
// This will generate struct_template.d.cg

struct DataStructure(T){
    T[] data;
}

void main(){
    import std.stdio;

    DataStructure!int ds;

    ds.data = [1,2,3,4,5,6];

    writeln(ds.data);
"struct_template.d" 17L, 281B written
```

```
DataStructure!int
{
    struct DataStructure
    {
        int[] data;
    }

}
```

Mixin

- The conversation around generating code starts to get interesting with the idea of a ‘mixin’
- A ‘mixin’ is literally just a piece of legal D code that will be replaced when you compile.
  - Observe the example to the right
  - You might be wondering why exactly we would use what is on the right, and we probably would not in this fashion -- however the ability to mixin strings at compile-time is powerful with generative capabilities.

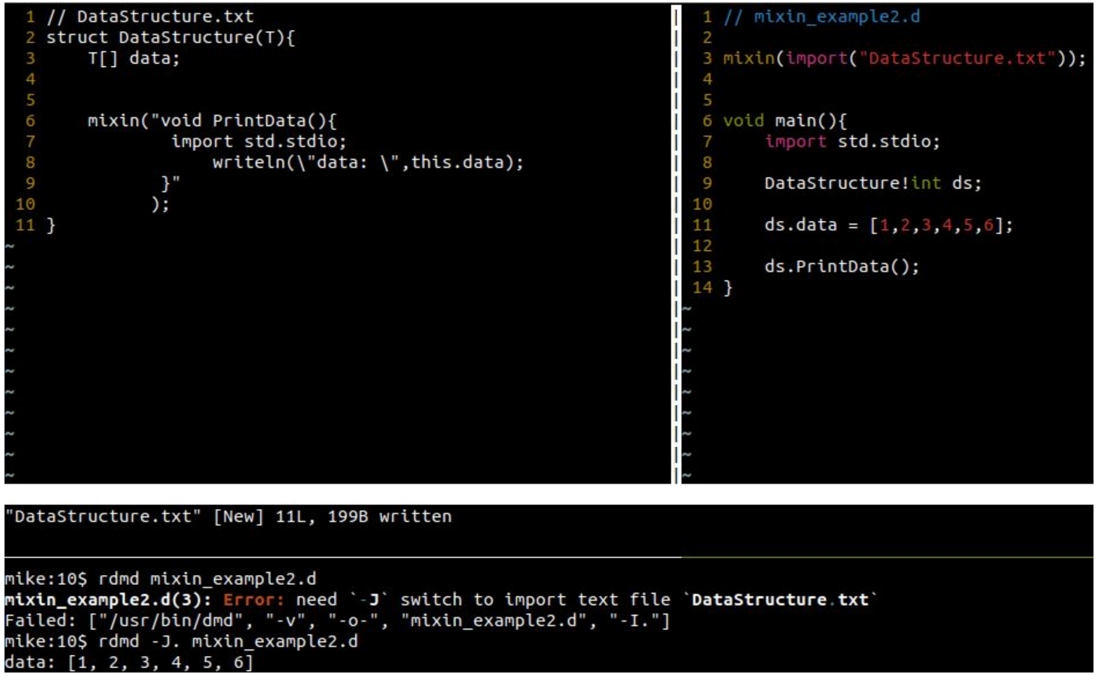


```d
// mixin_example.d

struct DataStructure(T){
    T[] data;


    mixin("void PrintData(){
                import std.stdio;
                    writeln(\"data: \",this.data);
               }"
          );
}

void main(){
    import std.stdio;

    DataStructure!int ds;

    ds.data = [1,2,3,4,5,6];

    ds.PrintData();
}
~
~
"mixin_example.d" 22L, 317B written
```

```
mike:10$ rdmd mixin_example.d
data: [1, 2, 3, 4, 5, 6]
```

Mixin and import

- With mixin’s you can actually read in code at compile-time from another file.
  - Note: On the example, that we provide with ‘-J’ a path (in this case a ‘.’) for where to search for import directories.
    - It’s probably best to have a dedicated directory for imports if you use them this way.



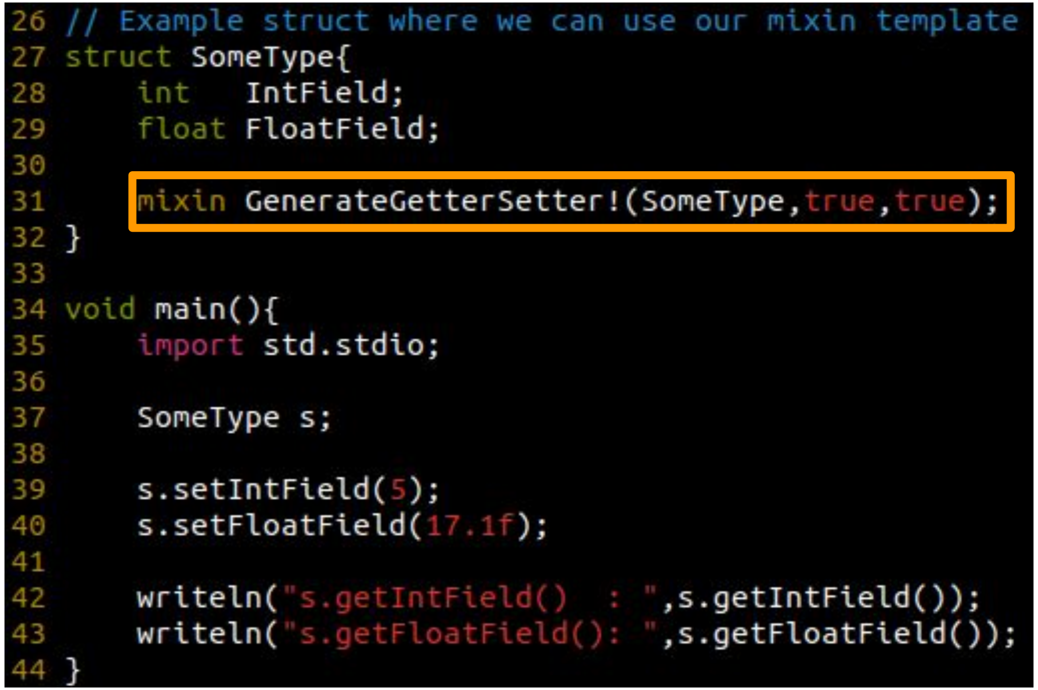
```
// DataStructure.txt
struct DataStructure(T){
    T[] data;


    mixin("void PrintData(){
                import std.stdio;
                    writeln(\"data: \",this.data);
               }"
             );
}
```

```
// mixin_example2.d

mixin(import("DataStructure.txt"));


void main(){
    import std.stdio;

    DataStructure!int ds;

    ds.data = [1,2,3,4,5,6];

    ds.PrintData();
}
```

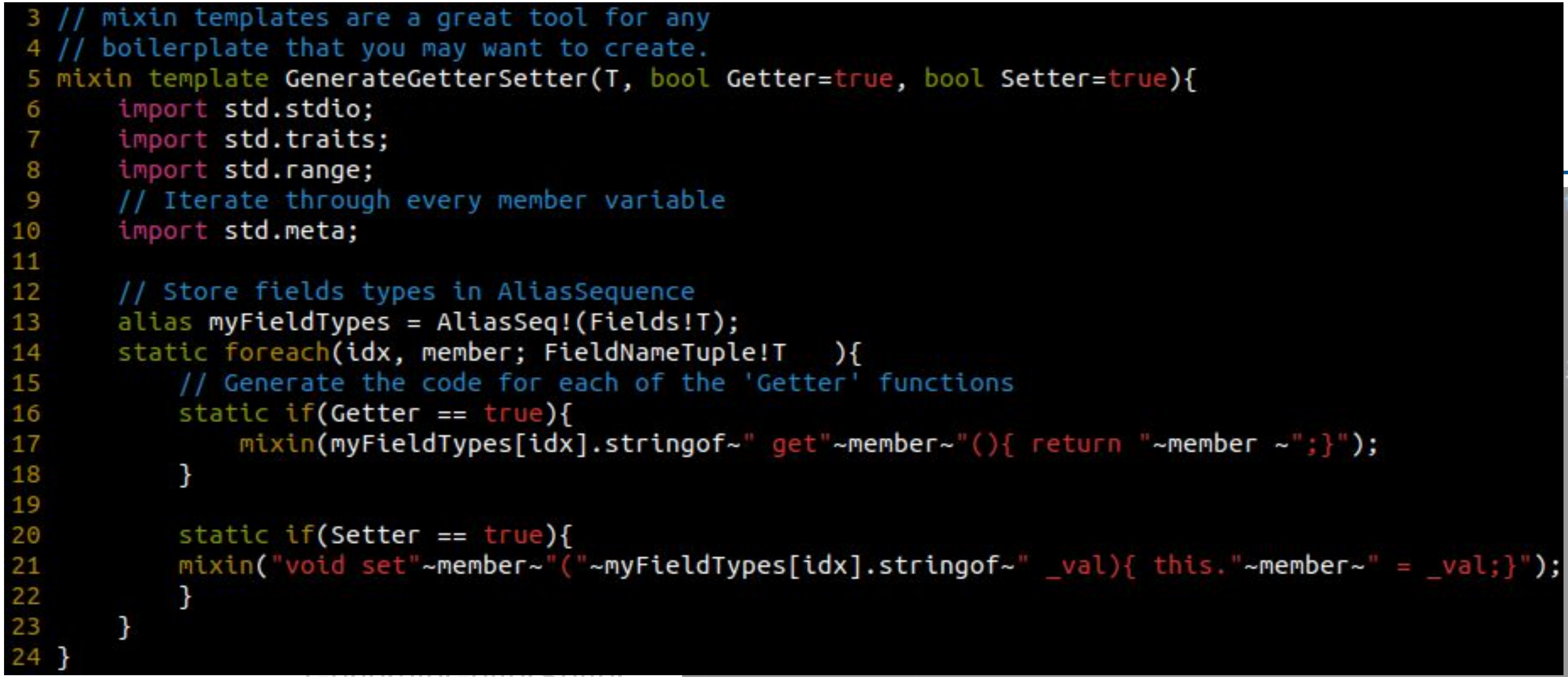
```
"DataStructure.txt" [New] 11L, 199B written

mike:10$ rdmd mixin_example2.d
mixin_example2.d(3): Error: need `-J` switch to import text file `DataStructure.txt`
Failed: ["/usr/bin/dmd", "-v", "-o-", "mixin_example2.d", "-I."]
mike:10$ rdmd -J. mixin_example2.d
data: [1, 2, 3, 4, 5, 6]
```

Mixin template and compile-time code generation (1/2)

- Here's a more powerful use case of templates and mixins.
    - Observe in this example that I do not define 'getter' and 'setter' member functions anywhere.
    - But how am I using them?
        - The secret lies in the 'GenerateGetterSetter' mixin template.



```d
// Example struct where we can use our mixin template
struct SomeType{
        int   IntField;
        float FloatField;

        mixin GenerateGetterSetter!(SomeType,true,true);
}

void main(){
        import std.stdio;

        SomeType s;

        s.setIntField(5);
        s.setFloatField(17.1f);

        writeln("s.getIntField()  : ",s.getIntField());
        writeln("s.getFloatField(): ",s.getFloatField());
}
```

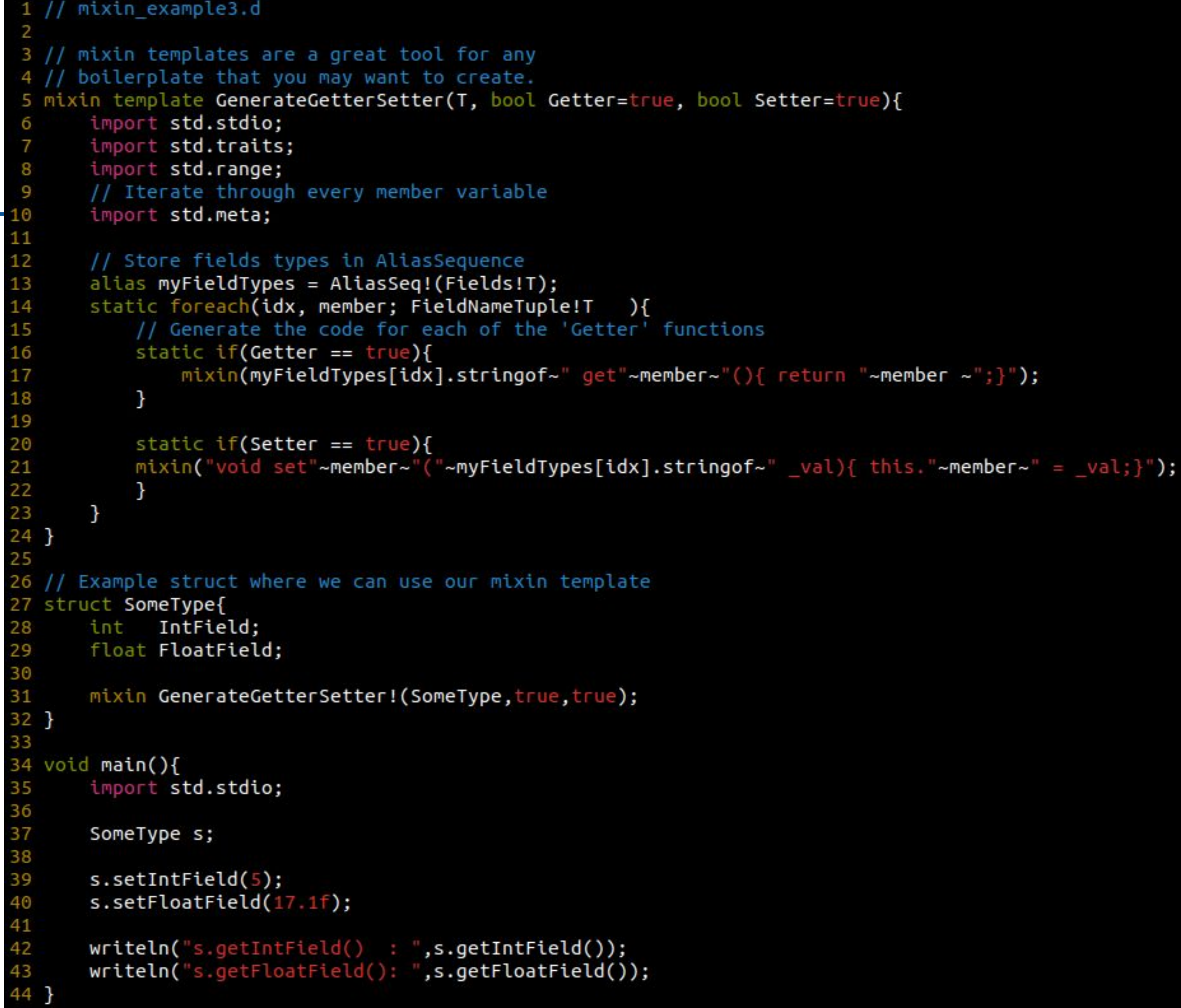


```
 3 // mixin templates are a great tool for any
 4 // boilerplate that you may want to create.
 5 mixin template GenerateGetterSetter(T, bool Getter=true, bool Setter=true){
 6     import std.stdio;
 7     import std.traits;
 8     import std.range;
 9     // Iterate through every member variable
10     import std.meta;
11 
12     // Store fields types in AliasSequence
13     alias myFieldTypes = AliasSeq!(Fields!T);
14     static foreach(idx, member; FieldNameTuple!T   ){
15         // Generate the code for each of the 'Getter' functions
16         static if(Getter == true){
17             mixin(myFieldTypes[idx].stringof~" get"~member~"(){ return "~member ~";}");
18         }
19 
20         static if(Setter == true){
21         mixin("void set"~member~"("~myFieldTypes[idx].stringof~" _val){ this."~member~" = _val;}");
22         }
23     }
24 }
```

- This snippet generates how to use various compile-time features of dlang to generate code.
- The std.traits library is used for ‘introspection’ to figure out the field names of a type
  - Line 13 creates an aliasSeq (i.e. tuple) for the Fields (from std.traits) of a given type
  - Then at line 14 we iterate through every field, and effectively write a new function.
  - The mixin template (line 5) itself is paramertized for whether to generate getter or setters.

Full example

- Same as previous slides, but all code on one slide



```d
// mixin_example3.d

// mixin templates are a great tool for any
// boilerplate that you may want to create.
mixin template GenerateGetterSetter(T, bool Getter=true, bool Setter=true){
    import std.stdio;
    import std.traits;
    import std.range;
    // Iterate through every member variable
    import std.meta;

    // Store fields types in AliasSequence
    alias myFieldTypes = AliasSeq!(Fields!T);
    static foreach(idx, member; FieldNameTuple!T    ){
        // Generate the code for each of the 'Getter' functions
        static if(Getter == true){
            mixin(myFieldTypes[idx].stringof~" get"~member~"(){ return "~member ~";}");
        }

        static if(Setter == true){
        mixin("void set"~member~"("~myFieldTypes[idx].stringof~" _val){ this."~member~" = _val;}");
        }
    }
}

// Example struct where we can use our mixin template
struct SomeType{
    int   IntField;
    float FloatField;

    mixin GenerateGetterSetter!(SomeType,true,true);
}

void main(){
    import std.stdio;

    SomeType s;

    s.setIntField(5);
    s.setFloatField(17.1f);

    writeln("s.getIntField()  : ",s.getIntField());
    writeln("s.getFloatField(): ",s.getFloatField());
}
```

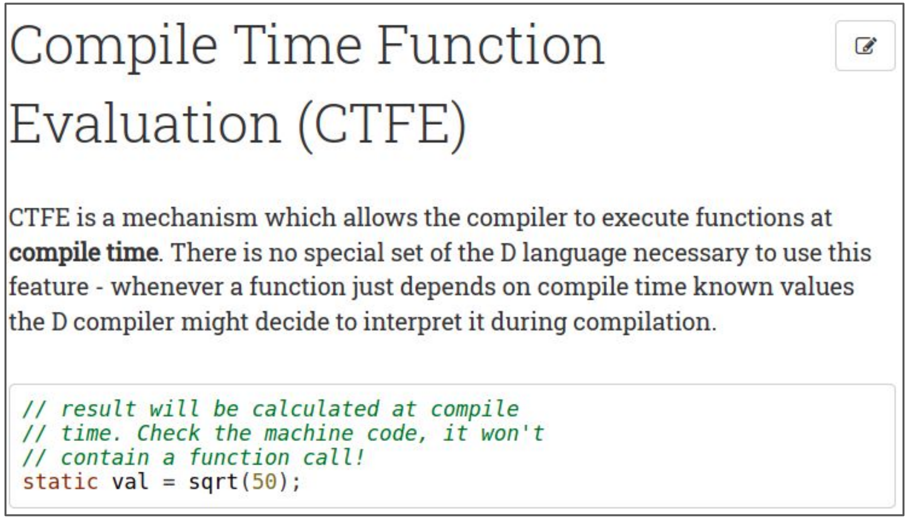
(Aside) What’s the point of this ‘meta-programming’

- Generic programming and generative programming paradigms can be very powerful
  - Often you are thinking about how to gather information at compile-time with some ‘std.traits’
  - The point is that you can generate lots of code:
    - Sometimes that is boilerplate code that may be tedious or error-prone to type
      - i.e. We probably could add various attributes like ‘pure’ to our generated code to make it even safer
        - (Note: however, most templated code in D does infer those traits automatically)
      - Generative code can also be quite useful in the sense that in an evolving codebase, I don’t have to constantly manage many things.
        - In our previous example, I can simply just add and remove fields and get getter/setter functions for free without thinking about it.

Other things in DLang

Compile-Time Features

- Full string manipulation capabilities at compile-time
  - (concatenation, indexing, selecting a substring, iterating, comparison....)
- We'll talk more about the various features like **compile-time function evaluation (CTFE)** as needed.



Compile Time Function Evaluation (CTFE)

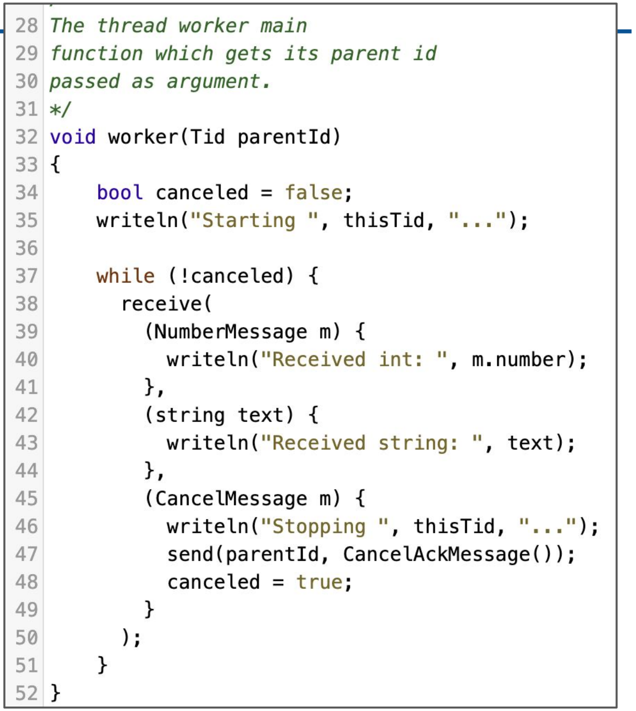
CTFE is a mechanism which allows the compiler to execute functions at **compile time**. There is no special set of the D language necessary to use this feature - whenever a function just depends on compile time known values the D compiler might decide to interpret it during compilation.

```
// result will be calculated at compile
// time. Check the machine code, it won't
// contain a function call!
static val = sqrt(50);
```

https://tour.dlang.org/tour/en/gems/compile-time-function-evaluation-ctfe

Threading and Safety

- When it comes to threading
  - D offers some memory safety when it comes to using threads
    - " Memory is thread-private by default, shared on demand." [Alexandrescu]
  - D otherwise has std.concurrency, which can be seen as similar to #include <thread>
    - Message passing is also built-in however to allow the 'send'ing and 'receiving' of messages from objects.



```
The thread worker main
function which gets its parent id
passed as argument.
*/
void worker(Tid parentId)
{
    bool canceled = false;
    writeln("Starting ", thisTid, "...");

    while (!canceled) {
      receive(
        (NumberMessage m) {
          writeln("Received int: ", m.number);
        },
        (string text) {
          writeln("Received string: ", text);
        },
        (CancelMessage m) {
          writeln("Stopping ", thisTid, "...");
          send(parentId, CancelAckMessage());
          canceled = true;
        }
      );
    }
}
```

https://tour.dlang.org/tour/en/multithreading/message-passing

SafeD

- SafeD is a subset of the D language that focuses on eliminating memory corruption possibilities
  - It’s still evolving, but again, D allows you to program in a safe way.
  - https://dlang.org/articles/safed.html
- There are *whispers* of a safe or certified D compiler -- as I learn more publically I can share.

Yet more to cover!

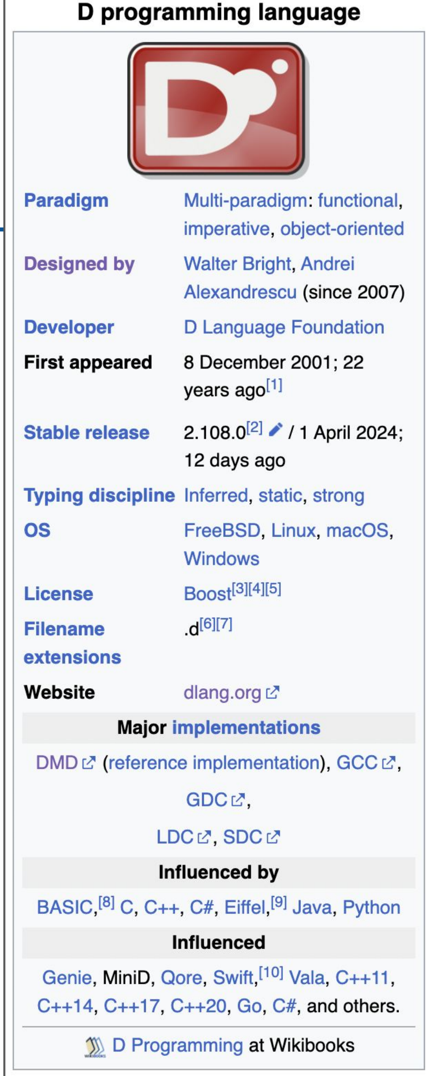
- Stack and Heap
- Avoiding Garbage Collection and the tradeoffs (@nogc)
- Ranges and iterators
- Emphasis on compile-time versus run-time
- assertions and unit testing
- casting data
- scope and safety of writing code
- interfaces
- Inheritance in classes example
- structure of a program with dub and other build systems

1. Some Thoughts on Programming Languages
2. A D Language preview
3. **C++ and DLang as they compliment each other**

1. Some Thoughts on Programming Languages
2. A D Language preview
3. **C++ and DLang as they compliment each other (From my perspective)**

C++ and DLang

- In a similar way that Modern C++ has evolved greatly since C++98, DLang has also been evolving and growing since it was first created in 2001 [wiki]
  - Learning both languages in their whole and in isolation to achieve mastery I have found difficult -- thus why I have spent a great deal of time learning these two languages side-by-side (Or rather -- picking up D after many years of C++).



D programming language

Paradigm	Multi-paradigm: functional, imperative, object-oriented
Designed by	Walter Bright, Andrei Alexandrescu (since 2007)
Developer	D Language Foundation
First appeared	8 December 2001; 22 years ago[1]
Stable release	2.108.0[2] / 1 April 2024; 12 days ago
Typing discipline	Inferred, static, strong
OS	FreeBSD, Linux, macOS, Windows
License	Boost[3][4][5]
Filename extensions	.d[6][7]
Website	dlang.org

Major implementations

DMD (reference implementation), GCC, GDC, LDC, SDC

Influenced by

BASIC,[8] C, C++, C#, Eiffel,[9] Java, Python

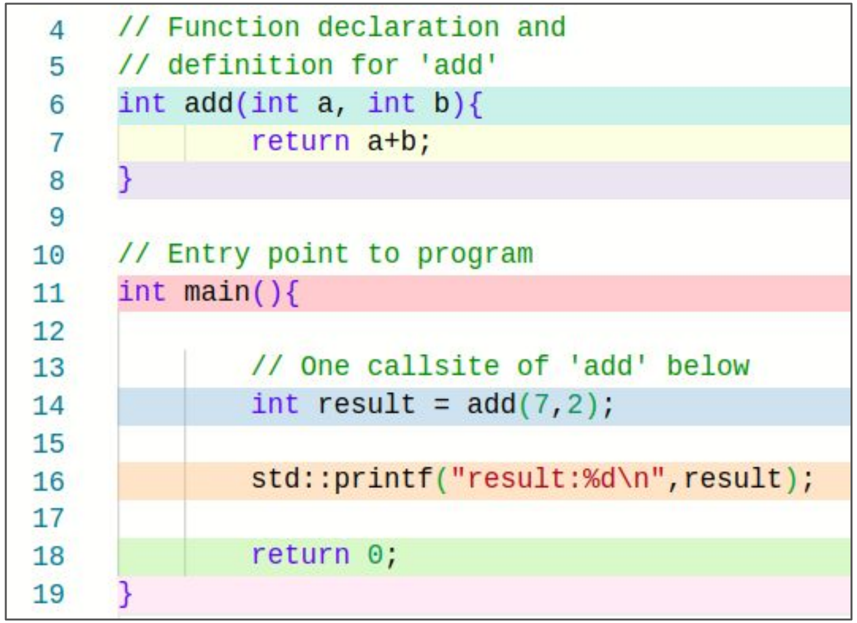
Influenced

Genie, MiniD, Qore, Swift,[10] Vala, C++11, C++14, C++17, C++20, Go, C#, and others.

D Programming at Wikibooks

C++: How C++ Improved my DLang - Performance Minded

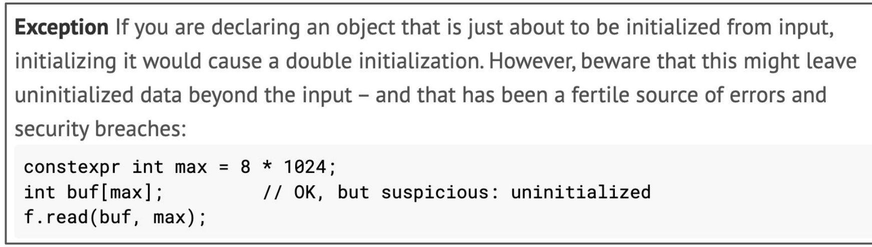
- Having worked in C++, perhaps one of the big advantages is the ability to think as a ‘systems programmer’
    - C++ has perhaps set me up to always be a ‘systems’ thinker.
    - A concrete example is that it is very common for me to try examples on compiler explorer, and review the generated assembly.
- C++ makes me wear my performance hat!



```cpp
// Function declaration and
// definition for 'add'
int add(int a, int b){
        return a+b;
}

// Entry point to program
int main(){

        // One callsite of 'add' below
        int result = add(7,2);

        std::printf("result:%d\n",result);

        return 0;
}
```

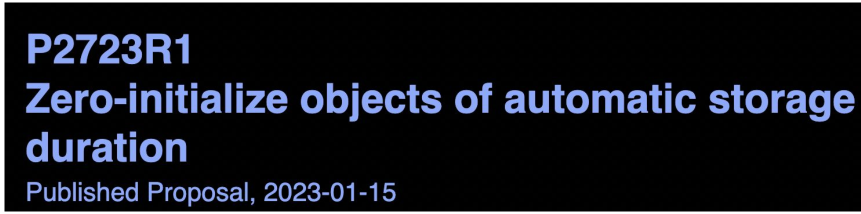
DLang: How DLang Improved my C++ - Security Minded

- In C++, memory is not initialized by default
  - C++ Core Guideline ‘Always initialize an object’ -- thus provides what is the default behavior in D (to always initialize values)
    - https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#es20-always-initialize-an-object
    - (Top-right image) shows the exception to this rule if you have some buffer that will be immediately filled
    - In DLang, we can use ‘=void’ to leave memory uninitialized
      - Similar proposals (bottom-right) have showed up in C++ to change the default behavior.



Exception If you are declaring an object that is just about to be initialized from input, initializing it would cause a double initialization. However, beware that this might leave uninitialized data beyond the input – and that has been a fertile source of errors and security breaches:

```
constexpr int max = 8 * 1024;
int buf[max];          // OK, but suspicious: uninitialized
f.read(buf, max);
```

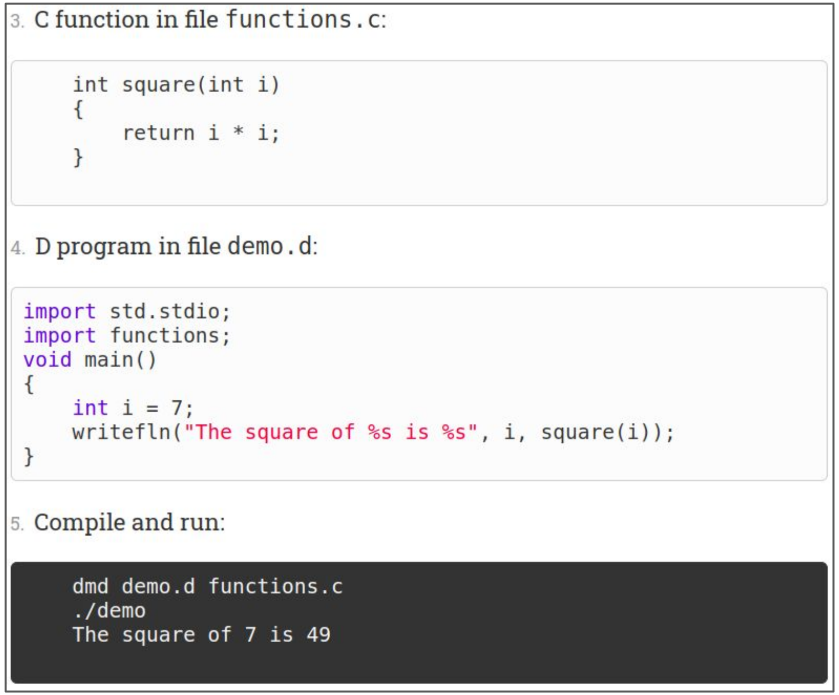


P2723R1
Zero-initialize objects of automatic storage duration
Published Proposal, 2023-01-15

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2723r1.html

C++: How C++ Improved my DLang - C Interop

- C++ naturally interops with the C programming language
  - Working with and interfacing with C libraries is something most C++ programmers do.
  - So I understood DLang's decision to use the same ABI as C
    - Additionally, I understand the value of 'importC' -- which is a compiler for C, built into the D compiler.
      - https://dlang.org/spec/importc.html
    - Interfacing to C in D is otherwise relatively trivial
      - https://dlang.org/spec/interfaceToC.html


3. C function in file functions.c:

```c
int square(int i)
{
    return i * i;
}
```

4. D program in file demo.d:

```d
import std.stdio;
import functions;
void main()
{
    int i = 7;
    writefln("The square of %s is %s", i, square(i));
}
```

5. Compile and run:

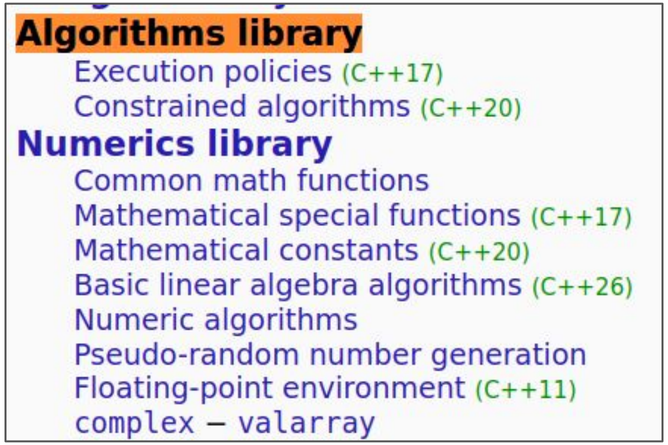
```
dmd demo.d functions.c
./demo
The square of 7 is 49
```

C++: How C++ Improved my DLang - std::vector

- C++ knowledge of std::vector applies to dlang's built-in dynamic array
  - In D we simply use 'int[] myIntArray;' to create a dynamic array.
    - However -- understanding the std::vector allocation means we may want to avoid the expense of copying when we resize.
  - e.g. Thus, in dlang .length can be set to change the size

C++: How C++ Improved my DLang - Trust the STL

- C++98 through 26 has had a vast amount of documentation for the STL
  - To new programmers -- learning to use #include <algorithm> in C++ is probably where you'll get the most growth from the C++ language -- at least to start as a programmer once learning the basics
- Leaning into the STL in C++, made it easy for me to lean into DLang's standard library (Phobos) and use it.



Algorithms library
- Execution policies (C++17)
- Constrained algorithms (C++20)

Numerics library
- Common math functions
- Mathematical special functions (C++17)
- Mathematical constants (C++20)
- Basic linear algebra algorithms (C++26)
- Numeric algorithms
- Pseudo-random number generation
- Floating-point environment (C++11)
- `complex` – `valarray`

https://en.cppreference.com/w/

DLang: How DLang Improved my C++ - Trust Phobos Library

- D has a larger standard library
  - Networking (std.sockets)
  - Curl
  - sql
  - json
  - zip
  - simd
  - memory mapped files -- https://dlang.org/phobos/std_mmfile.html
  - etc.
- I believe we need some form of these in the C++ STL
  - In D it makes it very easy to create at least the first prototype of tools with these libraries available.

DLang: How DLang Improved my C++ - Slices

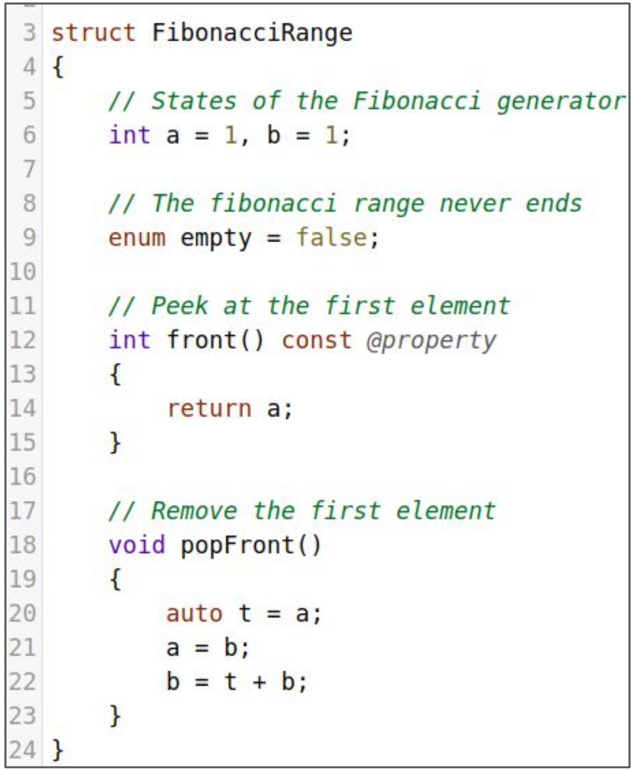
- D’s uses effectively slices (i.e. fat pointers) to access data
  - Arrays thus have a ‘size’ and a ‘length’ -- similar to ‘std::span’ and ‘std::string_view’ in C++
  - Slices are non-owning (we reference the array)
    - Slices are an ‘alias’ for a part of an array, and do not trigger dynamic memory allocation.
  - Having grown comfortable with D -- it becomes difficult to no use span and string_view in C++
- Read more on the design
  - https://digitalmars.com/articles/C-biggest-mistake.html (idea of ‘fat pointers’)
  - https://dlang.org/articles/d-array-article.html (slices)

DLang: How DLang Improved my C++ - struct vs class

- In C++ we carry the legacy of the ‘C’ struct
- In D we make the delineation that a struct is a ‘monomorphic type’ (i.e. no polymorphism), and a value type by default
  - I treat my C++ ‘structs’ the same way.
    - Thus, in C++ I use no inheritance for structs (mostly treating them as plain-old datatypes (POD))
  - I find this is a helpful ‘signal’ to the design and architecture of my codebase.

DLang: How DLang Improved my C++ - Ranges

- Dlang uses ranges by default
  - To be honest -- one of the first things this did was make me less scared of writing iterators in C++
    - i.e. It was easy to digest in DLang writing an ‘empty’, ‘front’, and ‘popFront’ function(), that I understood the same idea in C++
  - Ranges also made me understand the benefits of composition, infinite ranges, and being able to evaluate algorithms lazily
    - One level of abstraction higher than iterators, allows for some additional safety (i.e. harder to provide a pair of wrong iterators -- still some risk of invalidation however)



```
struct FibonacciRange
{
    // States of the Fibonacci generator
    int a = 1, b = 1;

    // The fibonacci range never ends
    enum empty = false;

    // Peek at the first element
    int front() const @property
    {
        return a;
    }

    // Remove the first element
    void popFront()
    {
        auto t = a;
        a = b;
        b = t + b;
    }
}
```

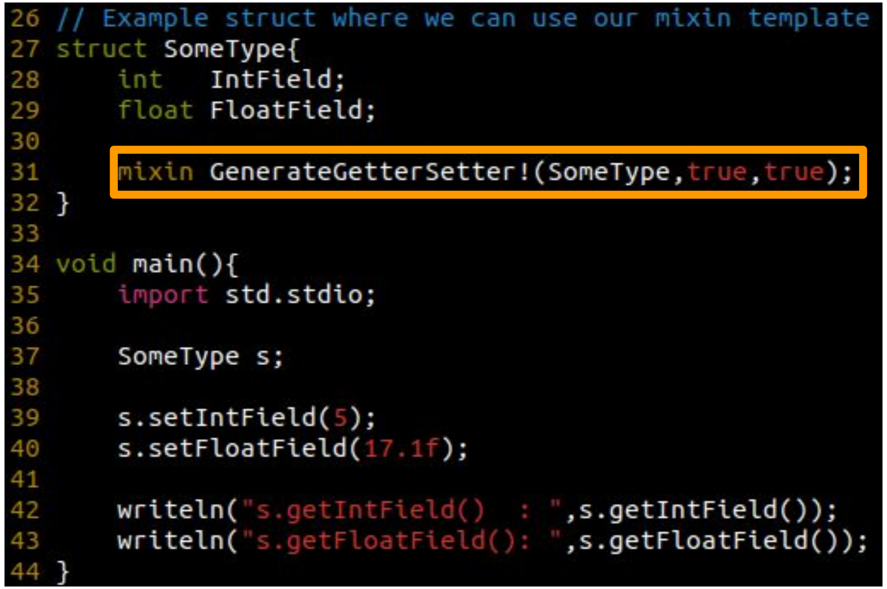
https://tour.dlang.org/tour/en/basics/ranges

DLang: How DLang Improved my C++ - UFCS

- Universal Function Call Syntax (UFCS)
  - This is a feature available in many languages besides D
  - When you get use to UFCS, you start thinking more so about writing ‘pure’ functions that compose.
    - ‘pure’ functions are more likely to be able to evaluate at compile-time
    - I *think* UFCS itself is better reminding me to write functions that have one job.
  - In C++ we utilize the overloads of the pipe operator to write more composable code (or otherwise use std::ranges)
    - An Overview of Standard Ranges - Tristan Brindle - CppCon 2019
    - https://youtu.be/SYLgG7Q5Zws?t=3335

DLang: How DLang Improved my C++ - Mixins

- In DLang, mixins and mixin templates are very powerful features
    - A simple ‘mixin(import(“some_file”)’ is a useful way to embed data at compile-time.
    - There are ways to do this in C++, but I ultimately hope we get an #embed like in C23



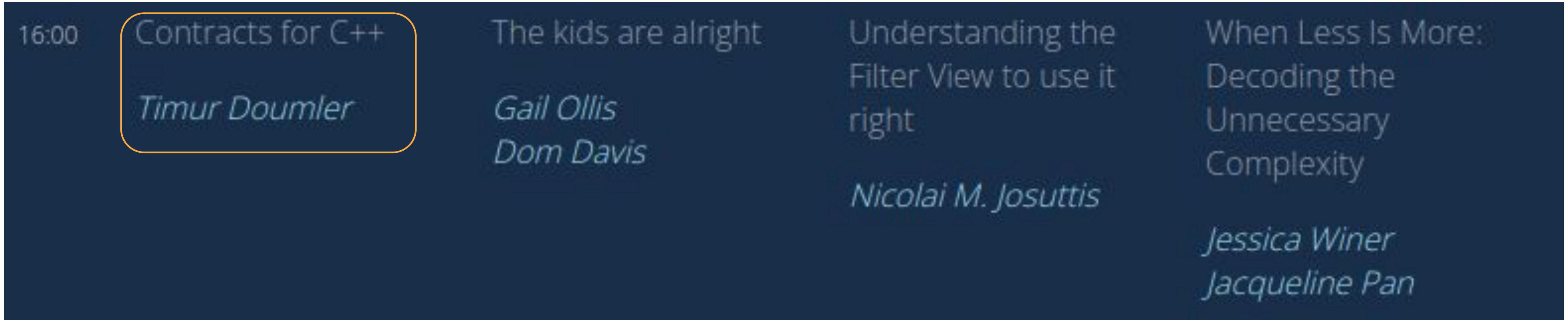
```d
// Example struct where we can use our mixin template
struct SomeType{
     int   IntField;
     float FloatField;

     mixin GenerateGetterSetter!(SomeType,true,true);
}

void main(){
     import std.stdio;

     SomeType s;

     s.setIntField(5);
     s.setFloatField(17.1f);

     writeln("s.getIntField()  : ",s.getIntField());
     writeln("s.getFloatField(): ",s.getFloatField());
}
```

DLang: How DLang Improved my C++ - The Defaults

- Covered earlier, but D fixes many defaults that C++ inherited from C
- Initialization of values
- And several other small quirks --
  - https://dlang.org/blog/the-d-and-c-series/
  - https://dlang.org/articles/cpptod.html
  - int* x,y; // in D produces two pointers to integers
  - int* x,y; // in C produces x as type int* and y as type int.
- Overall, I just try to do what (I consider) the right defaults are in my D code in my C++ code.

DLang: How DLang Improved my C++ - Contracts

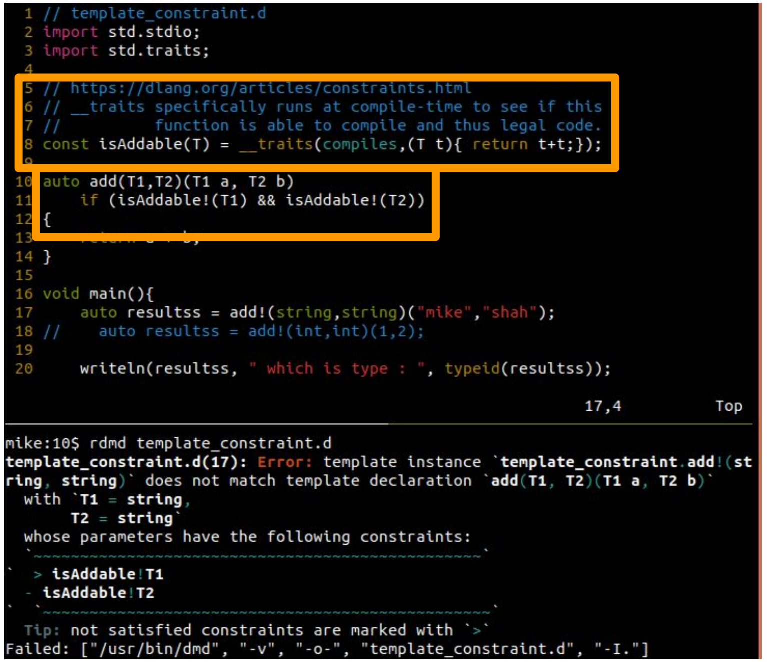
- D has ‘in’ (pre) and ‘out’ (post) conditions
  - https://dlang.org/spec/contracts.html
  - Nice to understand how these are used in debug build for checking conditions
  - Also keeps code nice and clean for my assertions
- See ACCU 2024 talk later today by Timur on contracts
  - I think this will help me with whatever is coming in C++ 26 (I’ll tell you in exactly 3.5 hours)



16:00	Contracts for C++ *Timur Doumler*	The kids are alright *Gail Ollis* *Dom Davis*	Understanding the Filter View to use it right *Nicolai M. Josuttis*	When Less Is More: Decoding the Unnecessary Complexity *Jessica Winer* *Jacqueline Pan*

Other “little things”

- Sometimes learning a simpler syntax helps make scary concepts or keywords make sense
  - In Dlang ‘typeof’ is like ‘decltype’
  - In DLang ‘template constraint’ is like ‘concept’ (‘a type system for templates’)
    - C++ Paper https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1886.pdf
  - Delegates in the D programming language carry state with them when passing functions around (thus making them first-class citizens, versus a function pointer which is not)
    - Implementing ‘functors’ in C++ and eventually understanding the connection with lambda’s lowering to functors was useful for wrapping my head around a ‘delegate’

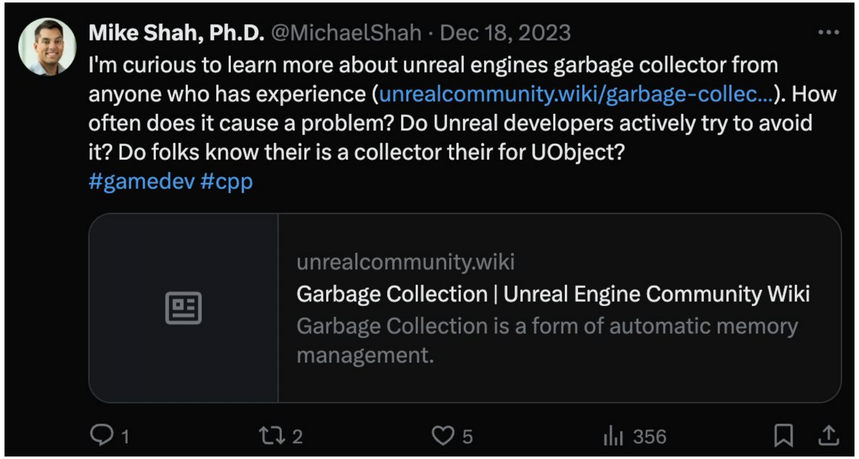


```d
// template_constraint.d
import std.stdio;
import std.traits;

// https://dlang.org/articles/constraints.html
// __traits specifically runs at compile-time to see if this
//           function is able to compile and thus legal code.
const isAddable(T) = __traits(compiles,(T t){ return t+t;});

auto add(T1,T2)(T1 a, T2 b)
    if (isAddable!(T1) && isAddable!(T2))
{
    return a + b;
}

void main(){
    auto resultss = add!(string,string)("mike","shah");
//    auto resultss = add!(int,int)(1,2);

    writeln(resultss, " which is type : ", typeid(resultss));
```

```
                                                  17,4          Top
mike:10$ rdmd template_constraint.d
template_constraint.d(17): Error: template instance `template_constraint.add!(st
ring, string)` does not match template declaration `add(T1, T2)(T1 a, T2 b)`
  with `T1 = string,
        T2 = string`
  whose parameters have the following constraints:
  `~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`
`  > isAddable!T1
  - isAddable!T2
`  `~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`
  Tip: not satisfied constraints are marked with `>`
Failed: ["/usr/bin/dmd", "-v", "-o-", "template_constraint.d", "-I."]
```

Other “big things” - Dlang and GC

- People are afraid of Garbage Collection
  - But you get memory safety effectively for free.
  - Allocation is just as fast as with ‘new’ or ‘malloc’
    - The scan/pause is the part that probably needs work on the allocator.
  - You don’t have to use the garbage collector (as previously shown)
  - It looks like high powered C++ game engines have portions that are collected
    - Maybe someone can respond to my tweet (Is it a GC, reference counted, arena -- help me if you know!)
  - See more on dlang garbage collection: https://dlang.org/blog/the-gc-series/


Mike Shah, Ph.D. @MichaelShah · Dec 18, 2023

I'm curious to learn more about unreal engines garbage collector from anyone who has experience (unrealcommunity.wiki/garbage-collec...). How often does it cause a problem? Do Unreal developers actively try to avoid it? Do folks know their is a collector their for UObject? #gamedev #cpp

unrealcommunity.wiki
Garbage Collection | Unreal Engine Community Wiki
Garbage Collection is a form of automatic memory management.

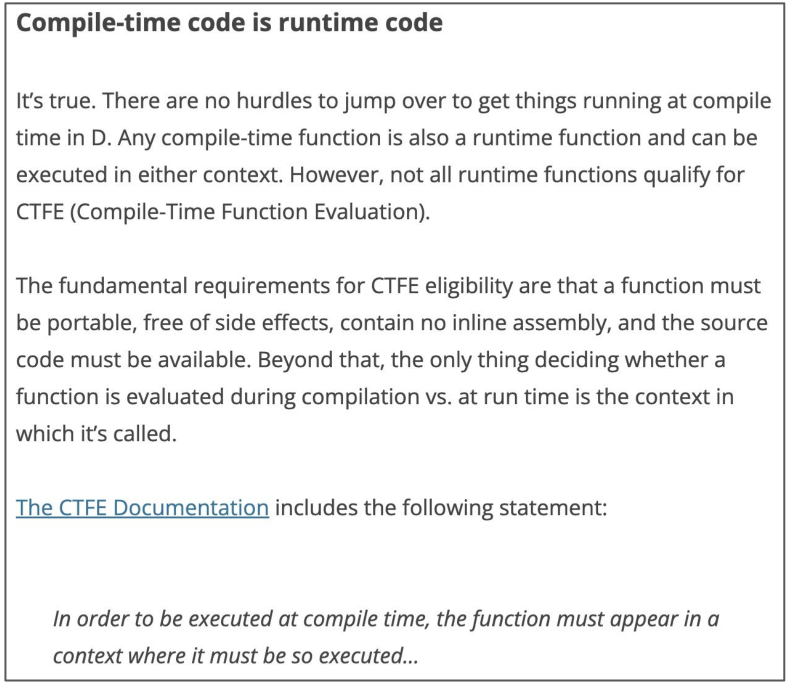
https://twitter.com/MichaelShah/status/1736695501259415873

Other “big things” - Dlang and Modules

- As soon as modules are fully available in most C++ compilers -- I will use them
  - D’s compilation is very fast, and I believe modules have one part to do with that (the language itself, and the ability to concurrently parse it is the other)

Other “big things” - constexpr

- In C++ the more ‘constexpr’ stuff the better
    - In DLang, The Compile Time Function Execution (CTFE) (Shown to the right) means that D pretty much tries to run everything it can at compile-time
        - Again -- this is probably the right default



Compile-time code is runtime code

It’s true. There are no hurdles to jump over to get things running at compile time in D. Any compile-time function is also a runtime function and can be executed in either context. However, not all runtime functions qualify for CTFE (Compile-Time Function Evaluation).

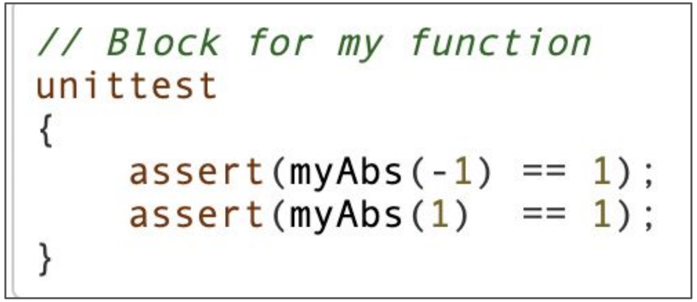
The fundamental requirements for CTFE eligibility are that a function must be portable, free of side effects, contain no inline assembly, and the source code must be available. Beyond that, the only thing deciding whether a function is evaluated during compilation vs. at run time is the context in which it’s called.

The CTFE Documentation includes the following statement:

> *In order to be executed at compile time, the function must appear in a context where it must be so executed...*

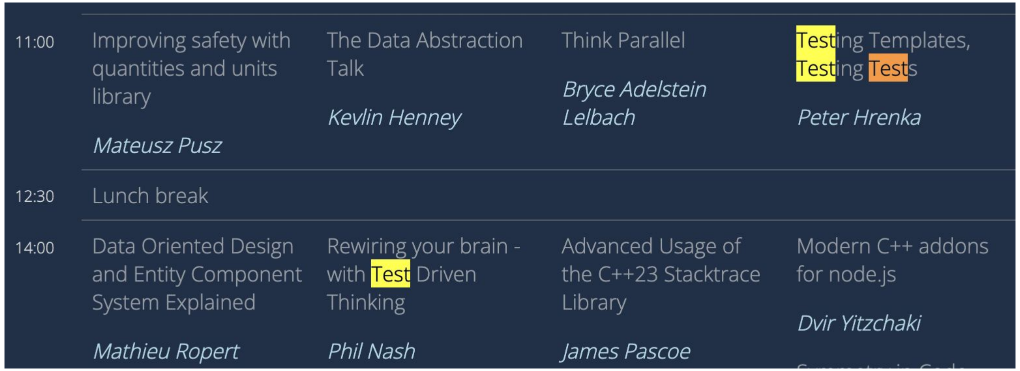
Other “big things” - unit testing

- Dlang has built-in unittest blocks -- the fact that they’re there means they get used
  - Very low friction
  - An important lesson that sometimes it’s worth adding the feature, even if it’s not perfect
- We’ll learn more about testing on Friday at this conference :)



```
// Block for my function
unittest
{
    assert(myAbs(-1) == 1);
    assert(myAbs(1)  == 1);
}
```

https://tour.dlang.org/tour/en/gems/unittesting



11:00	Improving safety with quantities and units library *Mateusz Pusz*	The Data Abstraction Talk *Kevlin Henney*	Think Parallel *Bryce Adelstein Lelbach*	Testing Templates, Testing Tests *Peter Hrenka*
12:30	Lunch break			
14:00	Data Oriented Design and Entity Component System Explained *Mathieu Ropert*	Rewiring your brain - with Test Driven Thinking *Phil Nash*	Advanced Usage of the C++23 Stacktrace Library *James Pascoe*	Modern C++ addons for node.js *Dvir Yitzchaki*

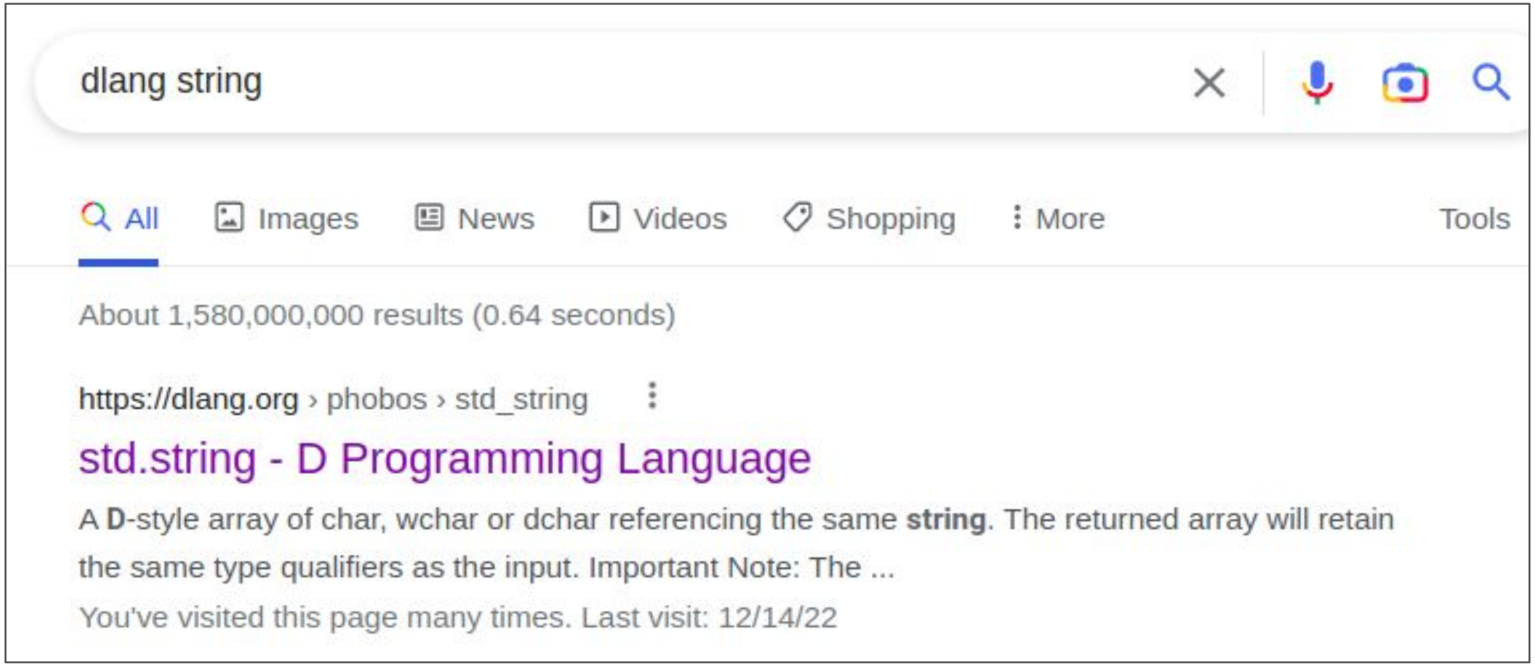
Other “big things” - ecosystem

- DLang has built-in package manager, profiler, and code-coverage (and can utilize some of the static analyzers in gcc/ldc2)
- C++ has a vast ecosystem, but no default
  - Learning from Dlang -- For C++ I have opted to pick popular tools (e.g. cmake, perf) and consider them my default for C++

Learning More About the D Language

More on Getting Help in D

- Usually 'dlang keyword' yields a result on the dlang.org homepage that I need.



dlang string

About 1,580,000,000 results (0.64 seconds)

https://dlang.org › phobos › std_string

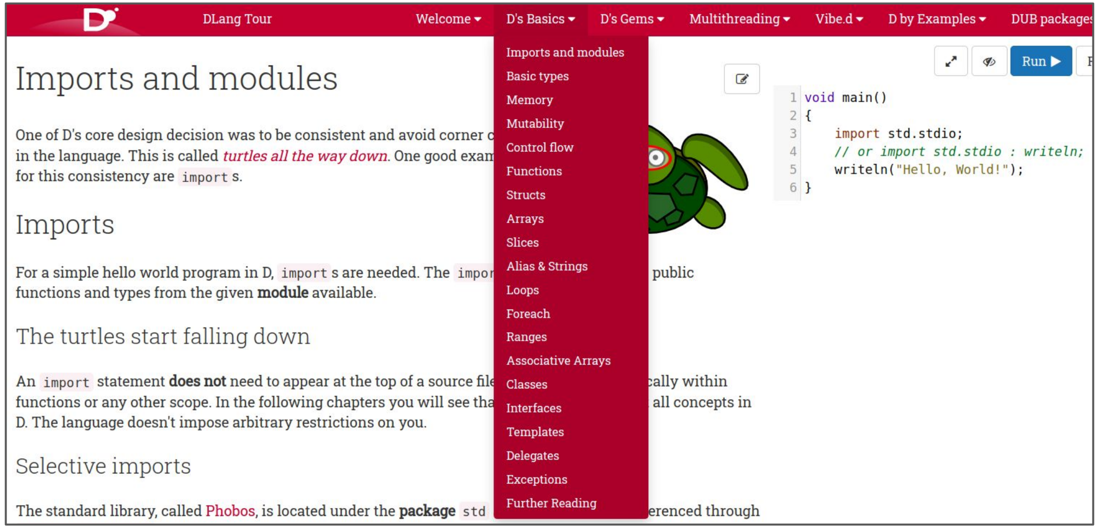
std.string - D Programming Language

A **D**-style array of char, wchar or dchar referencing the same **string**. The returned array will retain the same type qualifiers as the input. Important Note: The ...

You've visited this page many times. Last visit: 12/14/22

The D language tour

- Nice set of online tutorials that you can work through in one hour
  - Found directly on the D language website under ‘Learn’



Imports and modules

One of D's core design decision was to be consistent and avoid corner c in the language. This is called *turtles all the way down*. One good exam for this consistency are `import` s.

Imports

For a simple hello world program in D, `import` s are needed. The `impor` public functions and types from the given **module** available.

The turtles start falling down

An `import` statement **does not** need to appear at the top of a source file cally within functions or any other scope. In the following chapters you will see tha all concepts in D. The language doesn't impose arbitrary restrictions on you.

Selective imports

The standard library, called Phobos, is located under the **package** `std` erenced through

```d
void main()
{
    import std.stdio;
    // or import std.stdio : writeln;
    writeln("Hello, World!");
}
```

https://tour.dlang.org/

Understanding D: [From C to D] and [C++ to D]

- For those who have done some C and C++ programming, D should feel very familiar
  - I’d also suggest that folks who have used Java, may find D with just as rich of libraries, but a much more clean syntax.
- Useful guides
  - C to D
    - https://dlang.org/articles/ctod.html
  - C++ to D
    - https://dlang.org/articles/cpptod.html

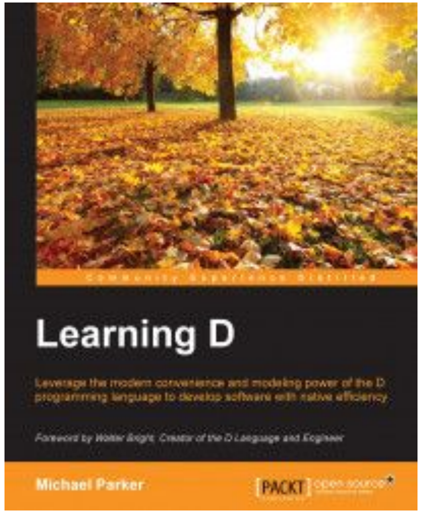
More Resources for Learning D

I would start with these two books

1. Programming in D by Ali Çehreli
   a. Freely available http://ddili.org/
2. Learning D by Michael Parker

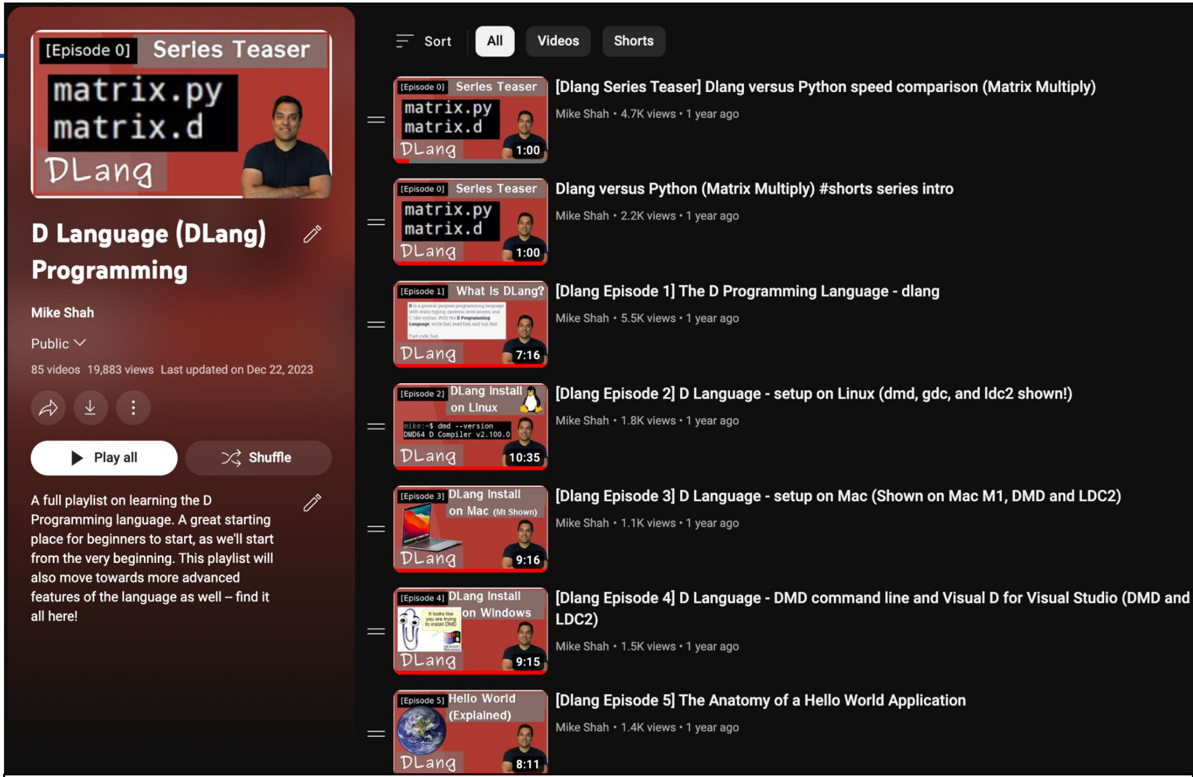
Any other books you find on D are also very good -- folks in the D community write books out of passion!

The online forums and discord are otherwise very active



YouTube - DLang

- I am actively adding more lessons (nearly 100 already) about the D programming language
  - https://www.youtube.com/c/MikeShah



D Language (DLang) Programming

Mike Shah

Public

85 videos 19,883 views Last updated on Dec 22, 2023

A full playlist on learning the D Programming language. A great starting place for beginners to start, as we'll start from the very beginning. This playlist will also move towards more advanced features of the language as well -- find it all here!

[Dlang Series Teaser] Dlang versus Python speed comparison (Matrix Multiply)
Mike Shah • 4.7K views • 1 year ago

Dlang versus Python (Matrix Multiply) #shorts series intro
Mike Shah • 2.2K views • 1 year ago

[Dlang Episode 1] The D Programming Language - dlang
Mike Shah • 5.5K views • 1 year ago

[Dlang Episode 2] D Language - setup on Linux (dmd, gdc, and ldc2 shown!)
Mike Shah • 1.8K views • 1 year ago

[Dlang Episode 3] D Language - setup on Mac (Shown on Mac M1, DMD and LDC2)
Mike Shah • 1.1K views • 1 year ago

[Dlang Episode 4] D Language - DMD command line and Visual D for Visual Studio (DMD and LDC2)
Mike Shah • 1.5K views • 1 year ago

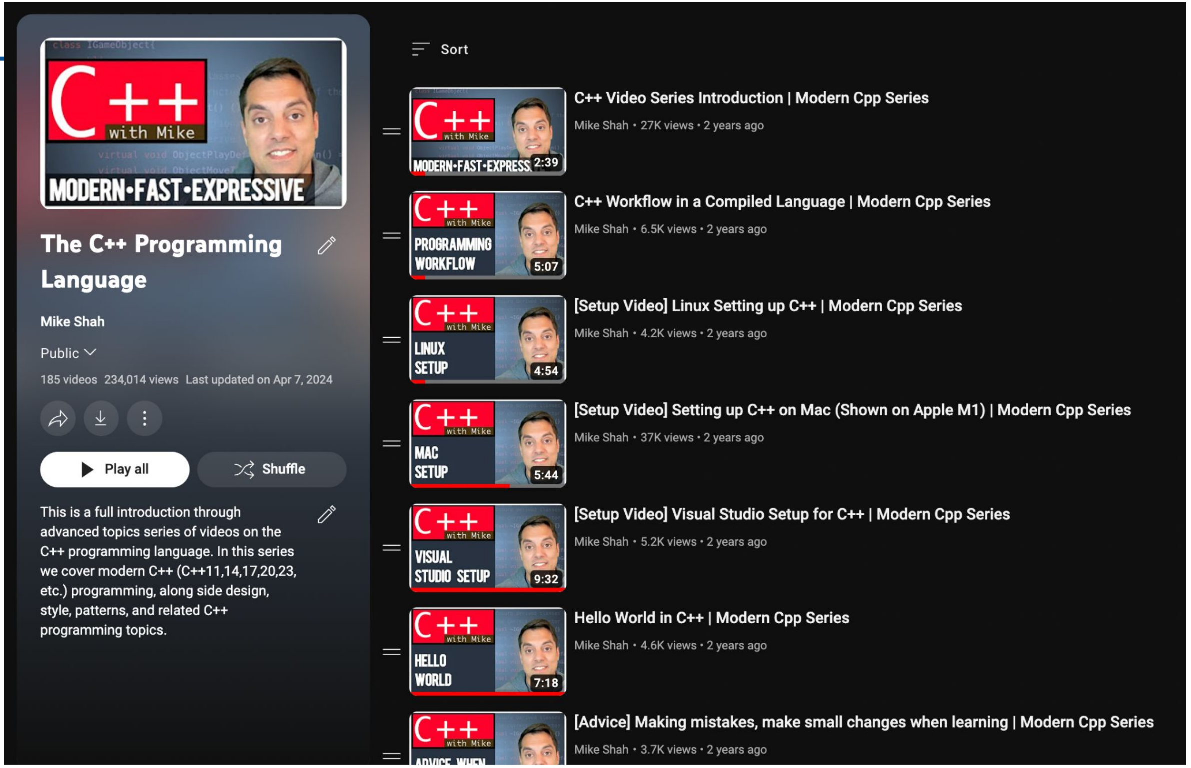
[Dlang Episode 5] The Anatomy of a Hello World Application
Mike Shah • 1.4K views • 1 year ago

Sort
All
Videos
Shorts
Play all
Shuffle
[Episode 0] Series Teaser
matrix.py
matrix.d
DLang
1:00
[Episode 1] What Is DLang?
7:16
[Episode 2] DLang Install on Linux
10:35
[Episode 3] DLang Install on Mac (M1 Shown)
9:16
[Episode 4] DLang Install on Windows
9:15
[Episode 5] Hello World (Explained)
8:11

https://www.youtube.com/playlist?list=PLvv0ScY6vfd9Fso-3cB4CGnSlW0E4btJV

YouTube - C++

- I am actively adding more lessons (nearly 180 already) about the C++ programming language
  - https://www.youtube.com/c/MikeShah



The C++ Programming Language

Mike Shah

Public

185 videos 234,014 views Last updated on Apr 7, 2024

This is a full introduction through advanced topics series of videos on the C++ programming language. In this series we cover modern C++ (C++11,14,17,20,23, etc.) programming, along side design, style, patterns, and related C++ programming topics.

C++ Video Series Introduction | Modern Cpp Series
Mike Shah • 27K views • 2 years ago

C++ Workflow in a Compiled Language | Modern Cpp Series
Mike Shah • 6.5K views • 2 years ago

[Setup Video] Linux Setting up C++ | Modern Cpp Series
Mike Shah • 4.2K views • 2 years ago

[Setup Video] Setting up C++ on Mac (Shown on Apple M1) | Modern Cpp Series
Mike Shah • 37K views • 2 years ago

[Setup Video] Visual Studio Setup for C++ | Modern Cpp Series
Mike Shah • 5.2K views • 2 years ago

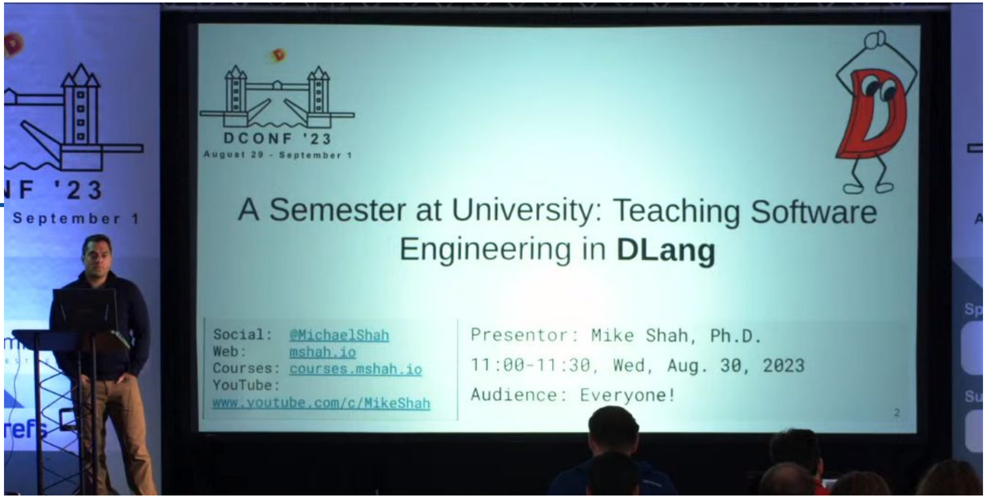
Hello World in C++ | Modern Cpp Series
Mike Shah • 4.6K views • 2 years ago

[Advice] Making mistakes, make small changes when learning | Modern Cpp Series
Mike Shah • 3.7K views • 2 years ago

https://www.youtube.com/playlist?list=PLvv0ScY6vfd8j-tlhYVPYgiIyXduu6m-L

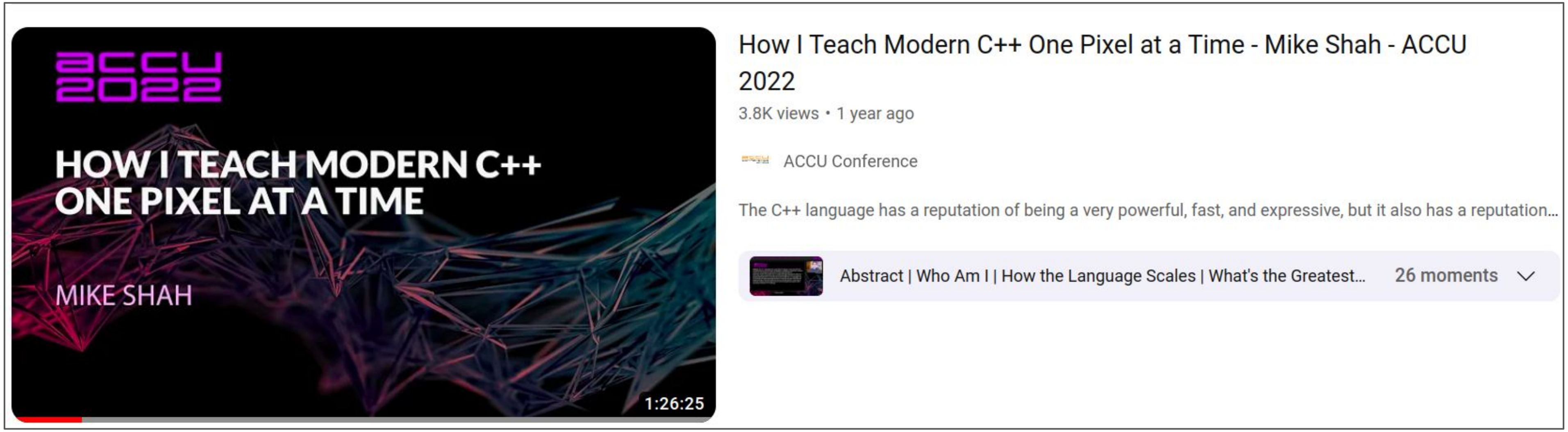
Teaching D Language

- You can hear my perspective
- **Even better** -- you can hear the students perspective
  - They built a networked collaborative paint program that is also available.
- D Conf 2023:
  - YouTube: https://www.youtube.com/live/wXTlafzlJVY?si=Xpy6g5h4wtIUrt2E&t=7711
  - Link to Conference Talk Description: https://dconf.org/2023/index.html


DCONF '23
August 29 - September 1
A Semester at University: Teaching Software Engineering in DLang
Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube: www.youtube.com/c/MikeShah
Presentor: Mike Shah, Ph.D.
11:00-11:30, Wed, Aug. 30, 2023
Audience: Everyone!

Teaching C++ Language

- You can hear my perspective from ACCU 2022.
- How I Teach Modern C++ One Pixel at a Time - Mike Shah - ACCU 2022
  - YouTube: https://www.youtube.com/watch?v=qwCc__MfAWk&t=2s



How I Teach Modern C++ One Pixel at a Time - Mike Shah - ACCU 2022

3.8K views • 1 year ago

ACCU Conference

The C++ language has a reputation of being a very powerful, fast, and expressive, but it also has a reputation...

Abstract | Who Am I | How the Language Scales | What's the Greatest... 26 moments

Some Anecdotal Findings (My Own)

- Simply working in a language with different defaults, may be a top factor in improving a programmers capabilities
  - e.g. (In DLang: initialized memory, struct vs class, thread-local storage)
- Simply working in a language with a different standard library may improve your programming capabilities
  - If anyone wants to run this type of study -- I’m around at my academic job with some interest in studying this problem (in theory forever)

Goal

Programming Languages - First Impressions

Mike Shah

Public

22 videos 2,345 views Last updated on Mar 24, 2024

This is a playlist where I download and try out programming languages in about

[Programming Languages] Episode 19 - First Impression - dlang (FOSDEM 2024 Talk)

Mike Shah • 812 views • 2 months ago

[Programming Languages] Episode 20 - First Impression - Fortran

Mike Shah • 750 views • 1 month ago

[Programming Languages] Episode 21 - First Impression - Elm

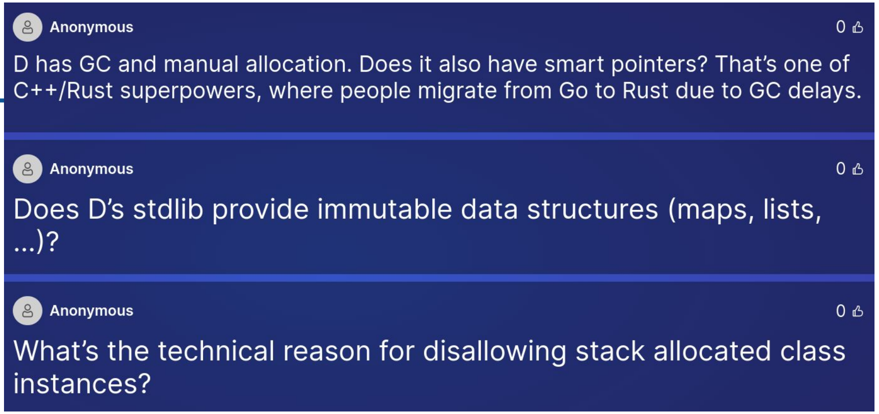
Mike Shah • 492 views • 3 weeks ago

But...If you remember just one thing after this talk:

1. **Set a timer for one hour**
2. **Go to https://tour.dlang.org/**
3. **Try out the D Language**

Apologies -- I missed these questions entered into slido after the talk

1. SafeRefCounted may be what you want for a ‘smart pointer’
   a. I believe there are other packages in ‘dub repository’ that provide C++ like smart pointers that are also thread-safe.
2. D’s stdlib has built-in types -- data structures marked as immutable (e.g. dynamic array) means the data referred to cannot be changed.
   a. You can otherwise have a dynamic array that can add unchanging data (i.e. dynamic array storing immutable values)
3. D does allow stack allocated classes using ‘scoped’ (programmer now is responsible for deallocation in correct scope)


Anonymous 0

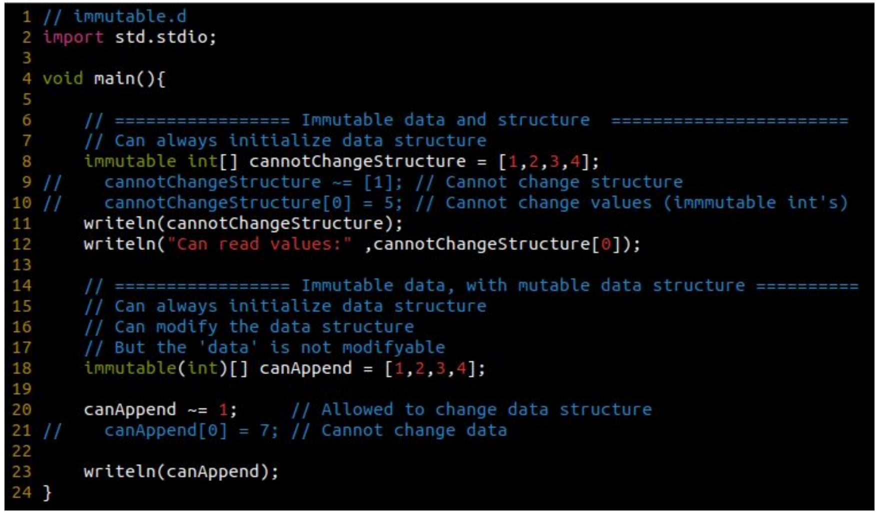
D has GC and manual allocation. Does it also have smart pointers? That’s one of C++/Rust superpowers, where people migrate from Go to Rust due to GC delays.

Anonymous 0

Does D’s stdlib provide immutable data structures (maps, lists, ...)?

Anonymous 0

What’s the technical reason for disallowing stack allocated class instances?


```
// immutable.d
import std.stdio;

void main(){

    // ================= Immutable data and structure  =====================
    // Can always initialize data structure
    immutable int[] cannotChangeStructure = [1,2,3,4];
//    cannotChangeStructure ~= [1]; // Cannot change structure
//    cannotChangeStructure[0] = 5; // Cannot change values (immmutable int's)
    writeln(cannotChangeStructure);
    writeln("Can read values:" ,cannotChangeStructure[0]);

    // ================= Immutable data, with mutable data structure =========
    // Can always initialize data structure
    // Can modify the data structure
    // But the 'data' is not modifyable
    immutable(int)[] canAppend = [1,2,3,4];

    canAppend ~= 1;     // Allowed to change data structure
//    canAppend[0] = 7; // Cannot change data

    writeln(canAppend);
}
```

Thank you ACCU 2024!

How DLang Improves My Modern C++ and Vice Versa

Mike Shah

11:00 - 12:30 BST Wed. April 17, 2024

75 minutes + 15 minute Q&A After
Audience: For all and all skill levels!

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube
www.youtube.com/c/MikeShah
http://tinyurl.com/mike-talks

Thank you!