Error Handling & D

Per Nordlöw

Machines, Software and Trust

- Why do *todays* programs crash?

Machines, Software and Trust

- Why do *todays* programs crash?
- Car Bulb Analogy

Machines, Software and Trust

- Why do *todays* programs crash?
- Car Bulb Analogy
- *Must* they?

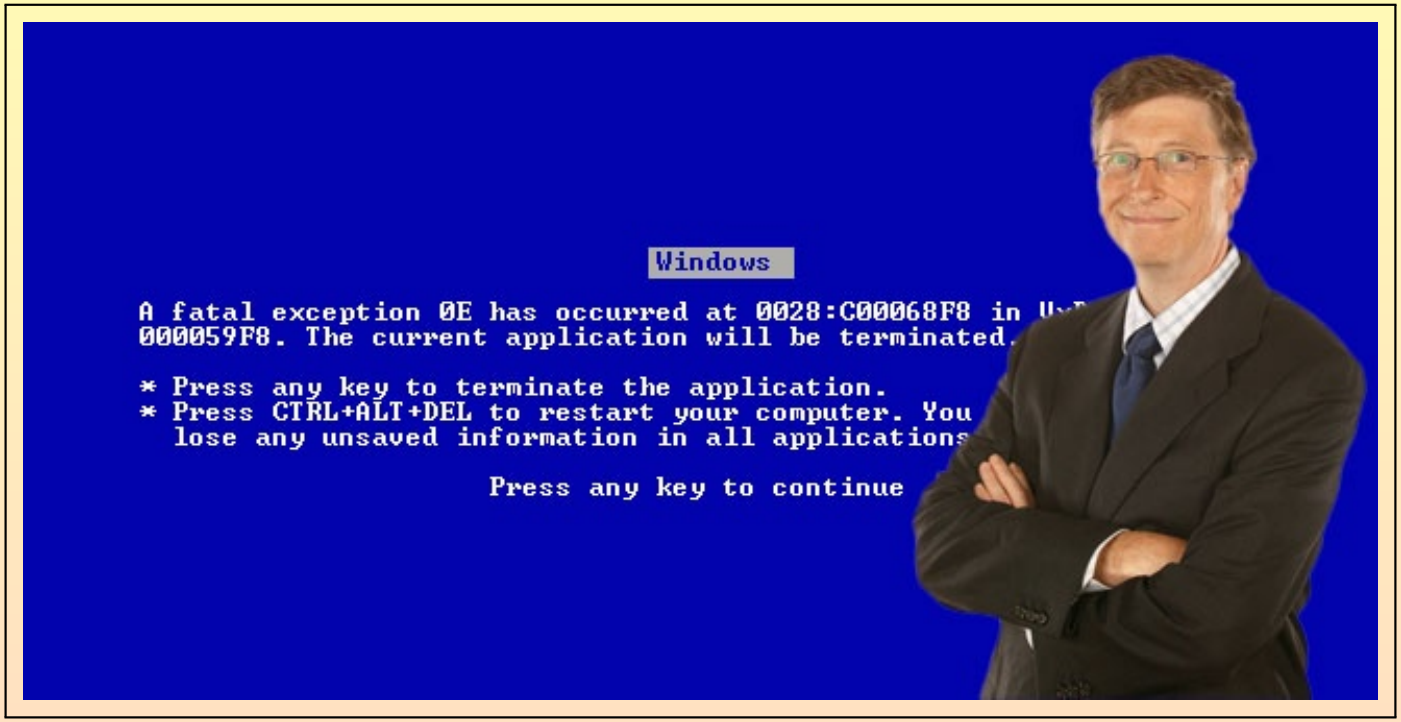
Machines, Software and Trust

- Why do *todays* programs crash?
- Car Bulb Analogy
- *Must* they?
- How to fix it

Machines, Software and Trust

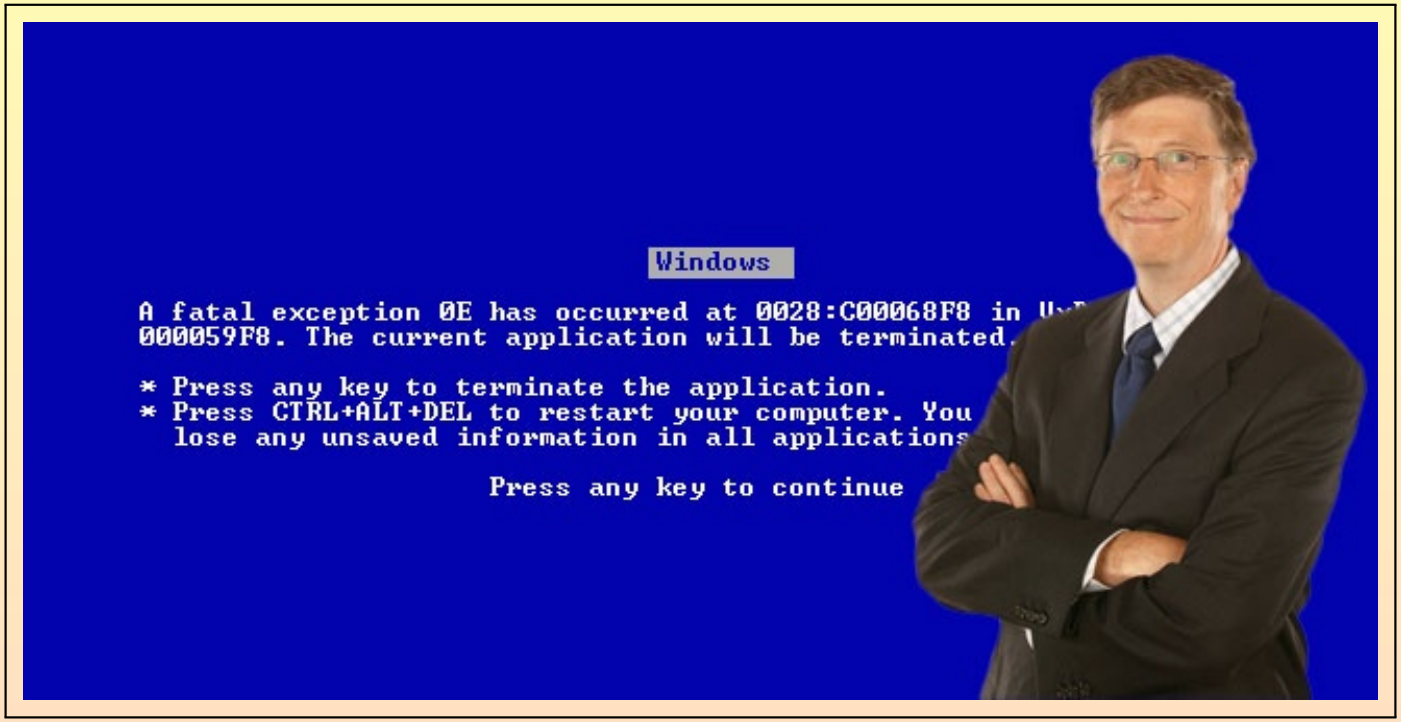
- Why do *todays* programs crash?
- Car Bulb Analogy
- *Must* they?
- How to fix it
- D is one step forward...

Trend: It's human to err!



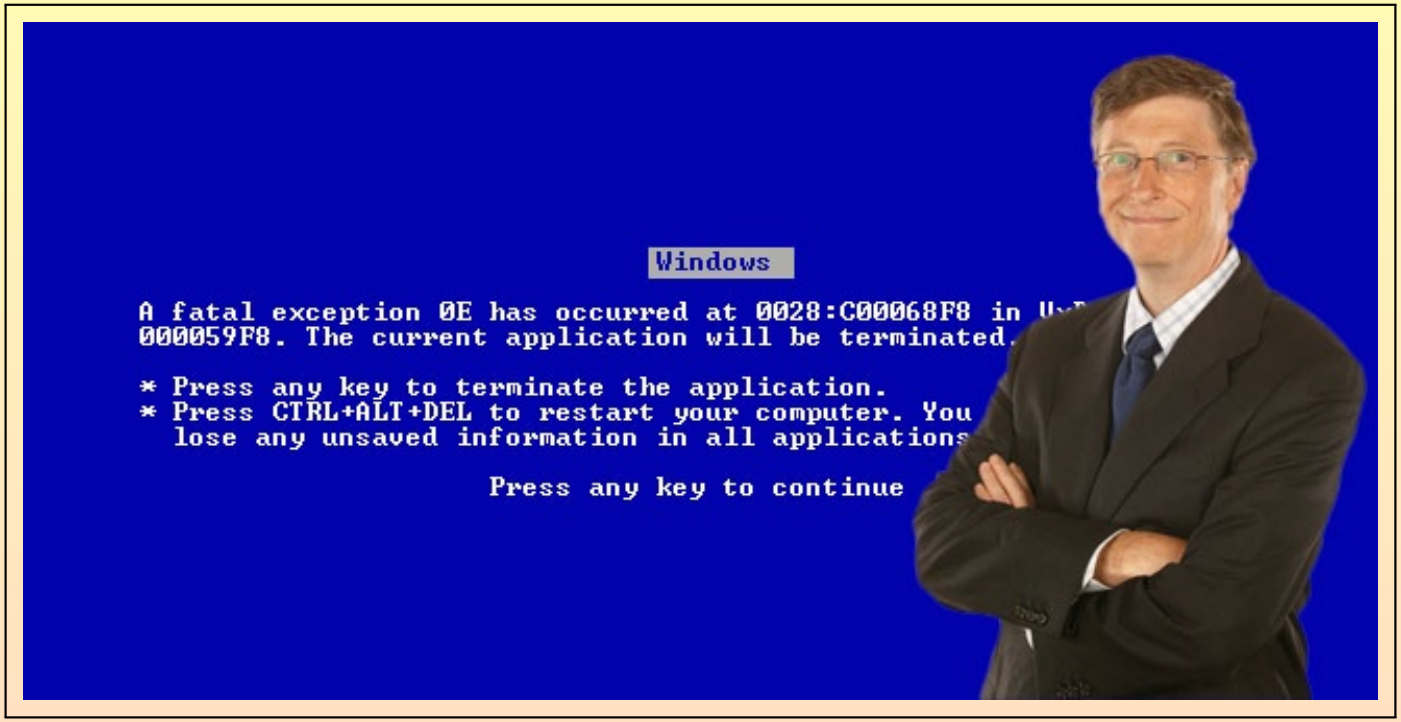
- Legacy: Humans don't err

Trend: It's human to err!



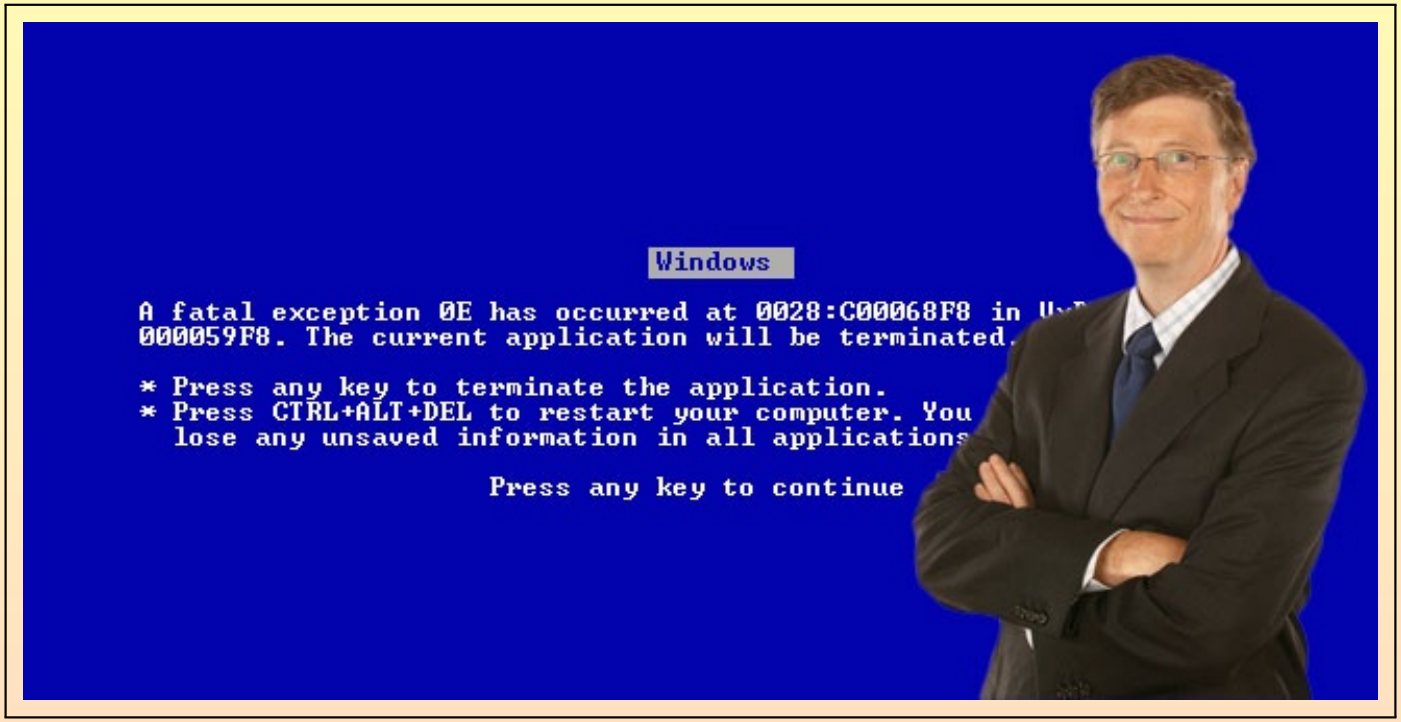
- Legacy: Humans don't err
- Changing Tides

Trend: It's human to err!



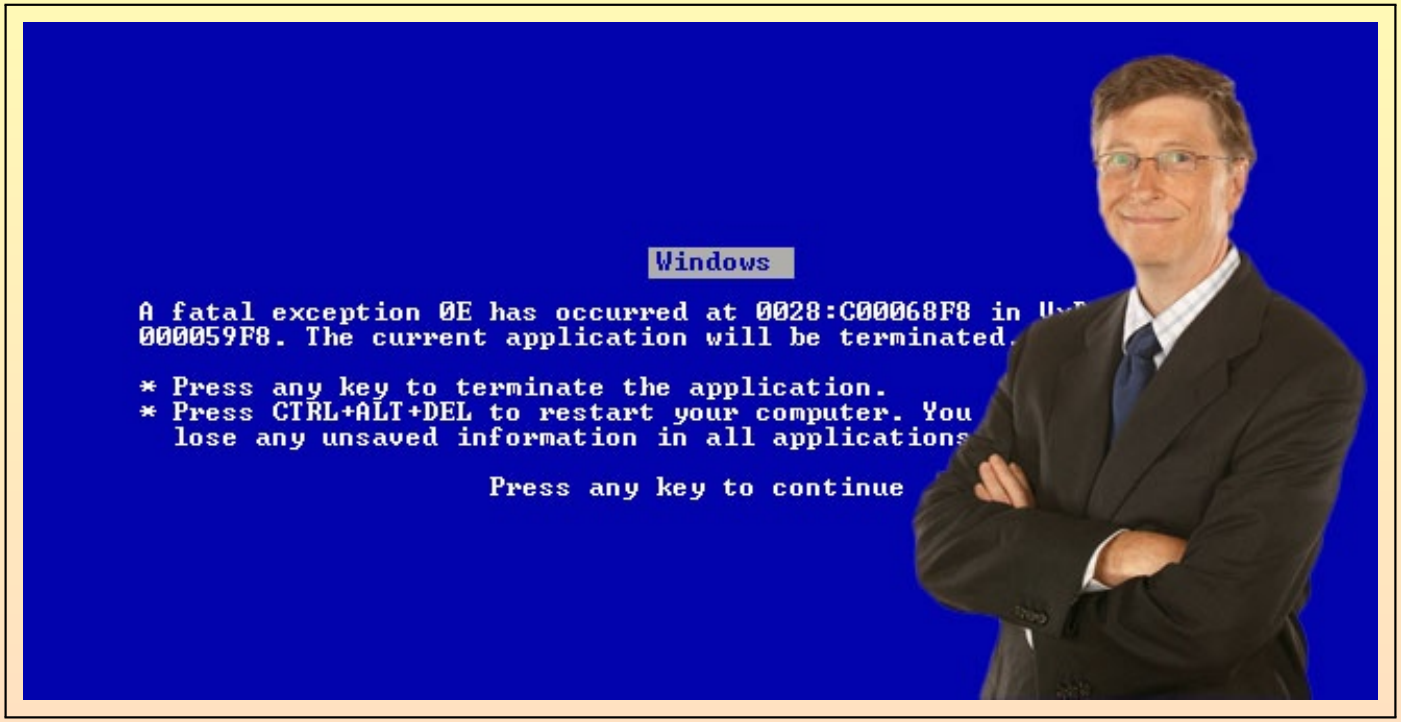
- Legacy: Humans don't err
- Changing Tides
  - Git: Mistakes Allowed: "Rewrite history"

Trend: It's human to err!



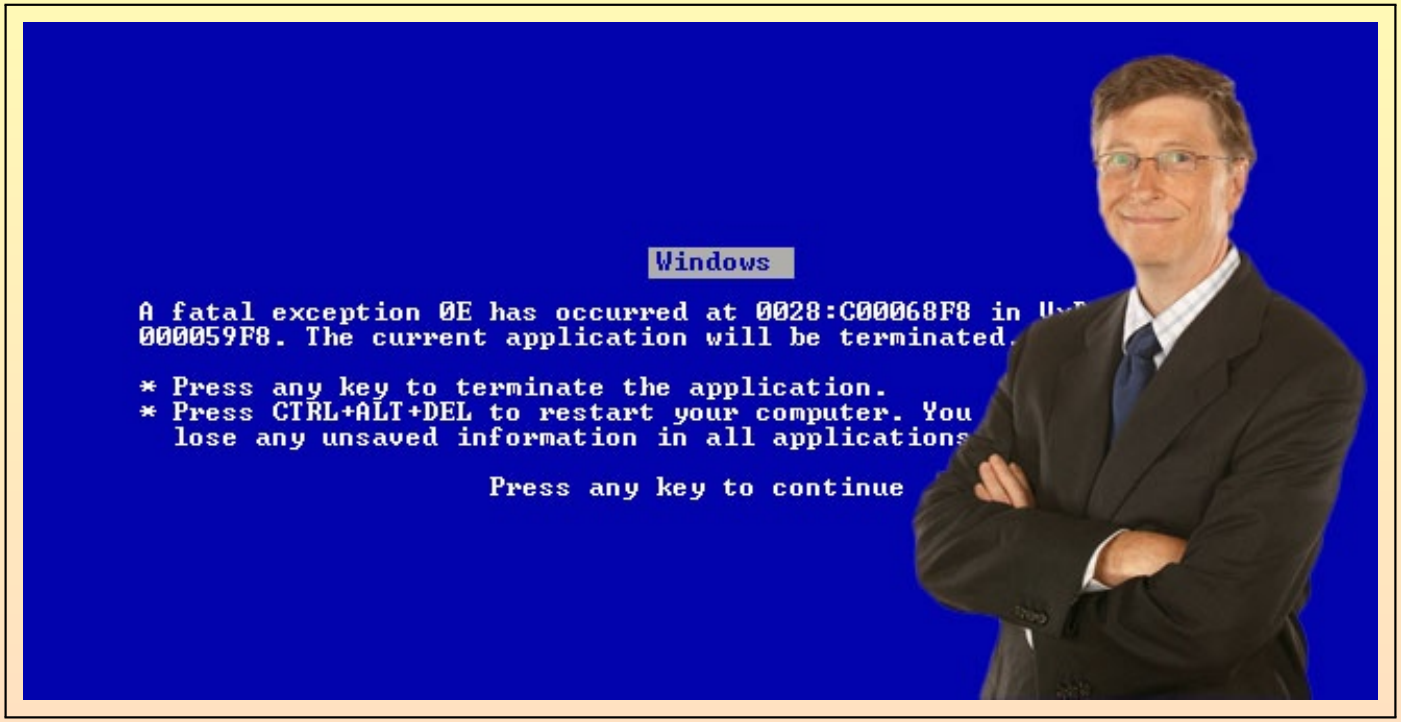
- Legacy: Humans don't err
- Changing Tides
  - Git: Mistakes Allowed: "Rewrite history"
  - No Programming Priesthood

Trend: It's human to err!



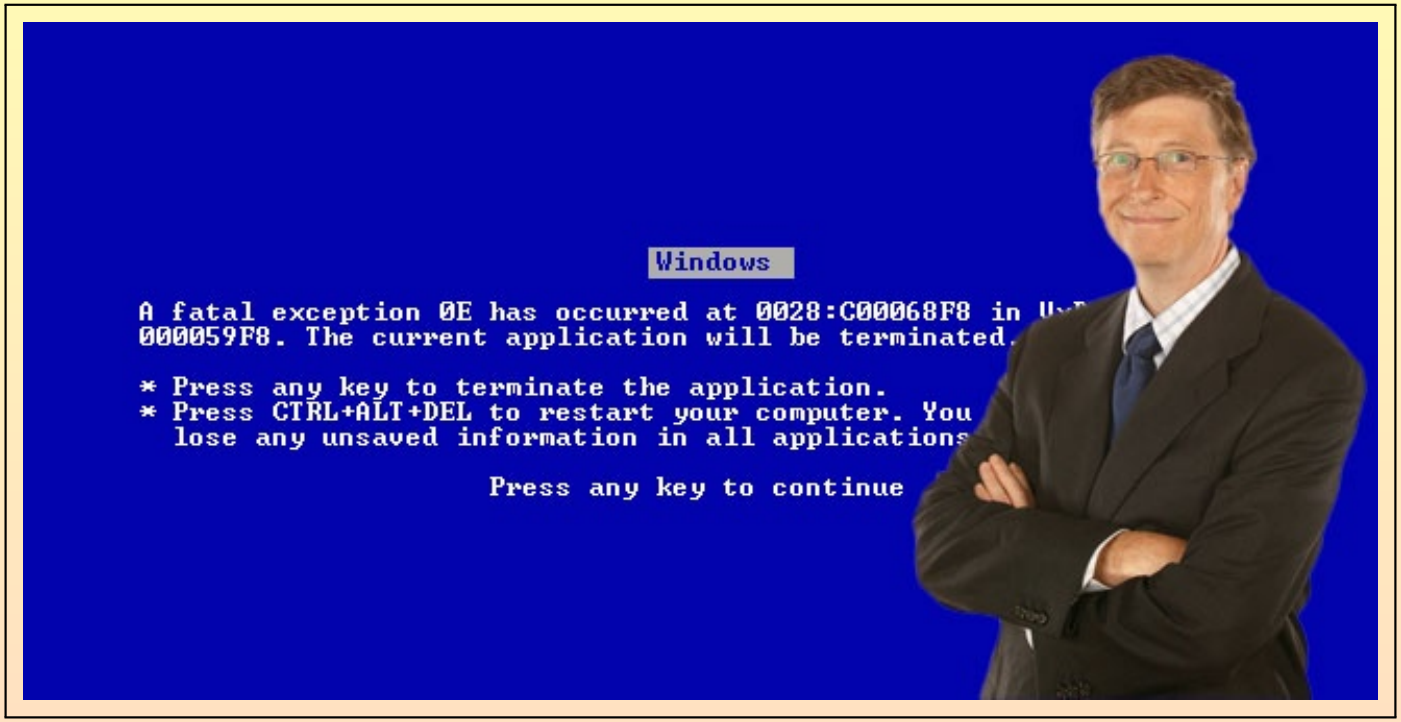
- Legacy: Humans don't err
- Changing Tides
  - Git: Mistakes Allowed: "Rewrite history"
  - No Programming Priesthood
- Humans & Machine Redundance

Trend: It's human to err!



- Legacy: Humans don't err
- Changing Tides
  - Git: Mistakes Allowed: "Rewrite history"
  - No Programming Priesthood
- Humans & Machine Redundance
- Error Correction

Trend: It's human to err!



- Legacy: Humans don't err
- Changing Tides
  - Git: Mistakes Allowed: "Rewrite history"
  - No Programming Priesthood
- Humans & Machine Redundance
- Error Correction
- Error Resilience

Software Process Goal: Detect Errors as *Early as* Possible

- **Build** (Compile, Link): Static Syntax & Semantic Checking

Software Process Goal: Detect Errors as *Early as* Possible

- **Build** (Compile, Link): Static Syntax & Semantic Checking
- **Test**: Dynamic Checking

Software Process Goal: Detect Errors as *Early as* Possible

- **Build** (Compile, Link): Static Syntax & Semantic Checking
- **Test**: Dynamic Checking
- **Run**: Dynamic Checking

Type System is the Key

- Information Propagation (DRY): Reflection

Type System is the Key

- Information Propagation (DRY): Reflection
- Rich Type Programming: Default, Range, Step, Saturation, Physical Unit

Type System is the Key

- Information Propagation (DRY): Reflection
- Rich Type Programming: Default, Range, Step, Saturation, Physical Unit
- Requires *Formal* Language to Capture its Algebra!

Type System is the Key

- Information Propagation (DRY): Reflection
- Rich Type Programming: Default, Range, Step, Saturation, Physical Unit
- Requires *Formal* Language to Capture its Algebra!
- Type Inference: Unit Analysis

Type System is the Key

- Information Propagation (DRY): Reflection
- Rich Type Programming: Default, Range, Step, Saturation, Physical Unit
- Requires *Formal* Language to Capture its Algebra!
- Type Inference: Unit Analysis
- *Type Safe Linking*: Should Change Everything! We just need to fixed GNU ld. GCC already puts types in debug information.

Type System is the Key

- Information Propagation (DRY): Reflection
- Rich Type Programming: Default, Range, Step, Saturation, Physical Unit
- Requires *Formal* Language to Capture its Algebra!
- Type Inference: Unit Analysis
- *Type Safe Linking*: Should Change Everything! We just need to fixed GNU ld. GCC already puts types in debug information.
- Trend: Static code inference can give guarantees on (concurrent) code!:
  Examples: Rust, Haskell, D, Scala, OCaml, F#, Julia partly C#

Error Handling Alternatives

- **Return Value**:

Error Handling Alternatives

- **Return Value**:
  - Nobody checks return of `printf` nor `malloc` in C.

Error Handling Alternatives

- **Return Value**:
  - Nobody checks return of `printf` nor `malloc` in C.
  - Try running programs on a system almost out of memory or disk.

Error Handling Alternatives

- **Return Value**:
  - Nobody checks return of `printf` nor `malloc` in C.
  - Try running programs on a system almost out of memory or disk.
- **Exceptions**:

Error Handling Alternatives

- **Return Value**:
  - Nobody checks return of `printf` nor `malloc` in C.
  - Try running programs on a system almost out of memory or disk.
- **Exceptions**:
  - Stroustrup on its performance

Error Handling Alternatives

- **Return Value**:
  - Nobody checks return of `printf` nor `malloc` in C.
  - Try running programs on a system almost out of memory or disk.
- **Exceptions**:
  - Stroustrup on its performance
  - Almost religious wars

Error Handling Alternatives

- **Return Value**:
  - Nobody checks return of `printf` nor `malloc` in C.
  - Try running programs on a system almost out of memory or disk.
- **Exceptions**:
  - Stroustrup on its performance
  - Almost religious wars
  - Polynomial Complexity

Error Handling Alternatives

- **Return Value**:
  - Nobody checks return of `printf` nor `malloc` in C.
  - Try running programs on a system almost out of memory or disk.
- **Exceptions**:
  - Stroustrup on its performance
  - Almost religious wars
  - Polynomial Complexity
  - Cleanup scope

Error Handling Alternatives

- **Return Value**:
  - Nobody checks return of `printf` nor `malloc` in C.
  - Try running programs on a system almost out of memory or disk.
- **Exceptions**:
  - Stroustrup on its performance
  - Almost religious wars
  - Polynomial Complexity
  - Cleanup scope
  - Exception safe constructors

Error Handling Alternatives

- **Return Value**:
  - Nobody checks return of `printf` nor `malloc` in C.
  - Try running programs on a system almost out of memory or disk.
- **Exceptions**:
  - Stroustrup on its performance
  - Almost religious wars
  - Polynomial Complexity
  - Cleanup scope
  - Exception safe constructors
- **Scope**:

Error Handling Alternatives

- **Return Value**:
  - Nobody checks return of `printf` nor `malloc` in C.
  - Try running programs on a system almost out of memory or disk.
- **Exceptions**:
  - Stroustrup on its performance
  - Almost religious wars
  - Polynomial Complexity
  - Cleanup scope
  - Exception safe constructors
- **Scope**:
  - D-only

Error Handling Alternatives

- **Return Value**:
  - Nobody checks return of `printf` nor `malloc` in C.
  - Try running programs on a system almost out of memory or disk.
- **Exceptions**:
  - Stroustrup on its performance
  - Almost religious wars
  - Polynomial Complexity
  - Cleanup scope
  - Exception safe constructors
- **Scope**:
  - D-only
  - Linear complexity

Error Handling Alternatives

- **Return Value**:
  - Nobody checks return of `printf` nor `malloc` in C.
  - Try running programs on a system almost out of memory or disk.
- **Exceptions**:
  - Stroustrup on its performance
  - Almost religious wars
  - Polynomial Complexity
  - Cleanup scope
  - Exception safe constructors
- **Scope**:
  - D-only
  - Linear complexity
  - Compiler generates nested try-catch expressions.

Redundancy

- Compactest code is random but impossible to understand and correct

Redundancy

- Compactest code is random but impossible to understand and correct
- Managers!: Computers have no common sense!

Redundancy

- Compactest code is random but impossible to understand and correct
- Managers!: Computers have no common sense!
- Not just numerics but

Redundancy

- Compactest code is random but impossible to understand and correct
- Managers!: Computers have no common sense!
- Not just numerics but
- Units, Default-Value, Range, Saturation, Ownership, Access, etc

Interaction with Structured Information

- Limit input to very specific structure in

Interaction with Structured Information

- Limit input to very specific structure in
- GUI/Webb Forms

Interaction with Structured Information

- Limit input to very specific structure in
- GUI/Webb Forms
- Example: Personal Codes, Dates, etc

Interaction with Structured Information

- Limit input to very specific structure in
- GUI/Webb Forms
- Example: Personal Codes, Dates, etc
- Lazy Programmers: Integrate pattern matching with UI component input logic and reuse!

Interaction with Structured Information

- Limit input to very specific structure in
- GUI/Webb Forms
- Example: Personal Codes, Dates, etc
- Lazy Programmers: Integrate pattern matching with UI component input logic and reuse!
- Leaders!: Investment always pays off in the long run!

General Tips

- Minimize Number of **Configurations**, especially in dynamic languages

General Tips

- Minimize Number of **Configurations**, especially in dynamic languages
- Minimize Number of **Code Paths**

General Tips

- Minimize Number of **Configurations**, especially in dynamic languages
- Minimize Number of **Code Paths**
- Learn & Use **Reflection** (Static & Dynamic) =>

General Tips

- Minimize Number of **Configurations**, especially in dynamic languages
- Minimize Number of **Code Paths**
- Learn & Use **Reflection** (Static & Dynamic) =>
- **Propagate Information Structure** to

General Tips

- Minimize Number of **Configurations**, especially in dynamic languages
- Minimize Number of **Code Paths**
- Learn & Use **Reflection** (Static & Dynamic) =>
- **Propagate Information Structure** to
  - Command Line

General Tips

- Minimize Number of **Configurations**, especially in dynamic languages
- Minimize Number of **Code Paths**
- Learn & Use **Reflection** (Static & Dynamic) =>
- **Propagate Information Structure** to
  - Command Line
  - GUIs

General Tips

- Minimize Number of **Configurations**, especially in dynamic languages
- Minimize Number of **Code Paths**
- Learn & Use **Reflection** (Static & Dynamic) =>
- **Propagate Information Structure** to
  - Command Line
  - GUIs
  - Protocols (showcase Boost.Serialization)

General Tips

- Minimize Number of **Configurations**, especially in dynamic languages
- Minimize Number of **Code Paths**
- Learn & Use **Reflection** (Static & Dynamic) =>
- **Propagate Information Structure** to
  - Command Line
  - GUIs
  - Protocols (showcase Boost.Serialization)
  - etc

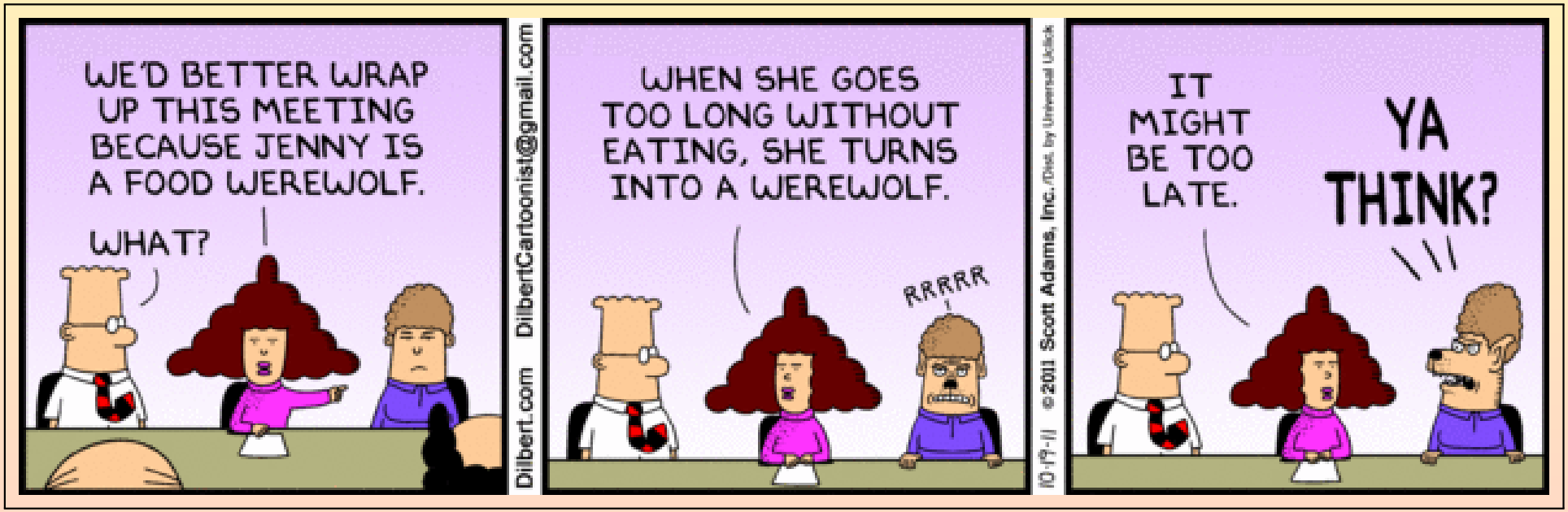
General Tips

- Minimize Number of **Configurations**, especially in dynamic languages
- Minimize Number of **Code Paths**
- Learn & Use **Reflection** (Static & Dynamic) =>
- **Propagate Information Structure** to
  - Command Line
  - GUIs
  - Protocols (showcase Boost.Serialization)
  - etc
- Make it impossible for user to enter incorrectly formatted data

General Tips

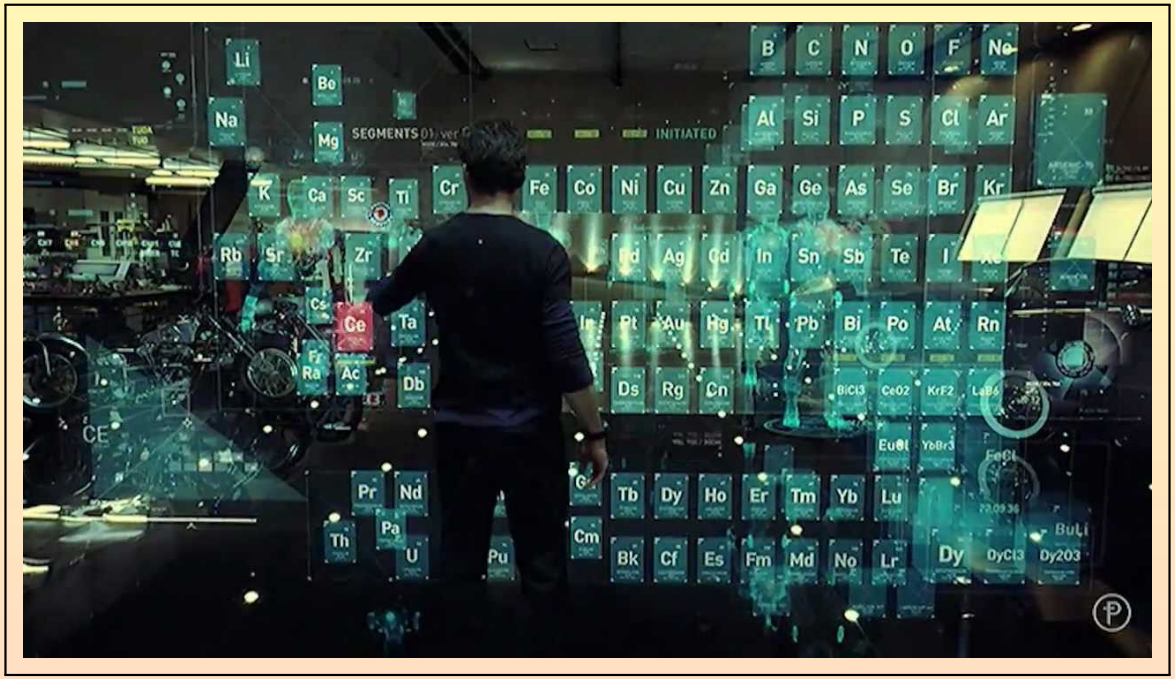
- Minimize Number of **Configurations**, especially in dynamic languages
- Minimize Number of **Code Paths**
- Learn & Use **Reflection** (Static & Dynamic) =>
- **Propagate Information Structure** to
  - Command Line
  - GUIs
  - Protocols (showcase Boost.Serialization)
  - etc
- Make it impossible for user to enter incorrectly formatted data
- Separate logic from presentation (CSS, LaTeX, Qt-XML)

Dinner!


WE'D BETTER WRAP UP THIS MEETING BECAUSE JENNY IS A FOOD WEREWOLF.
WHAT?
Dilbert.com DilbertCartoonist@gmail.com
WHEN SHE GOES TOO LONG WITHOUT EATING, SHE TURNS INTO A WEREWOLF.
RRRRR
10-19-11 ©2011 Scott Adams, Inc./Dist. by Universal Uclick
IT MIGHT BE TOO LATE.
YA THINK?

Iron Man like Systems and Interfaces?

How do we turn software engineers into Tony Starks? :)

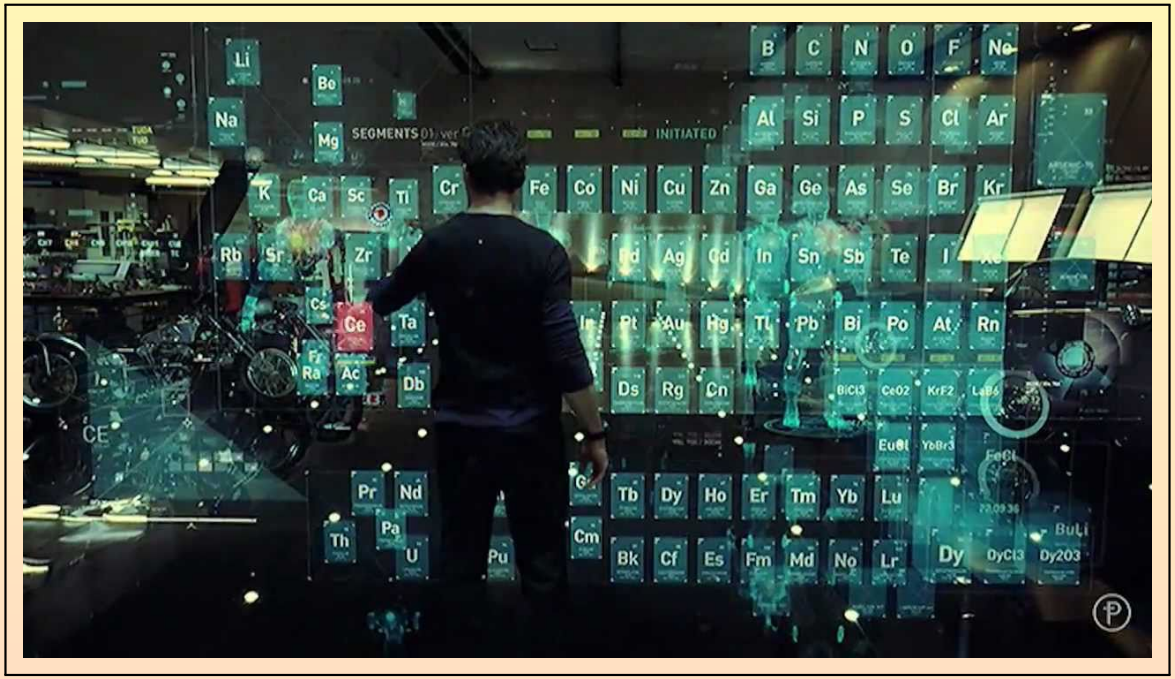


and what language shall we bootstrap this with!? We need it to be

- **Safe**: Provably Correct Memory Accesses

Iron Man like Systems and Interfaces?

How do we turn software engineers into Tony Starks? :)

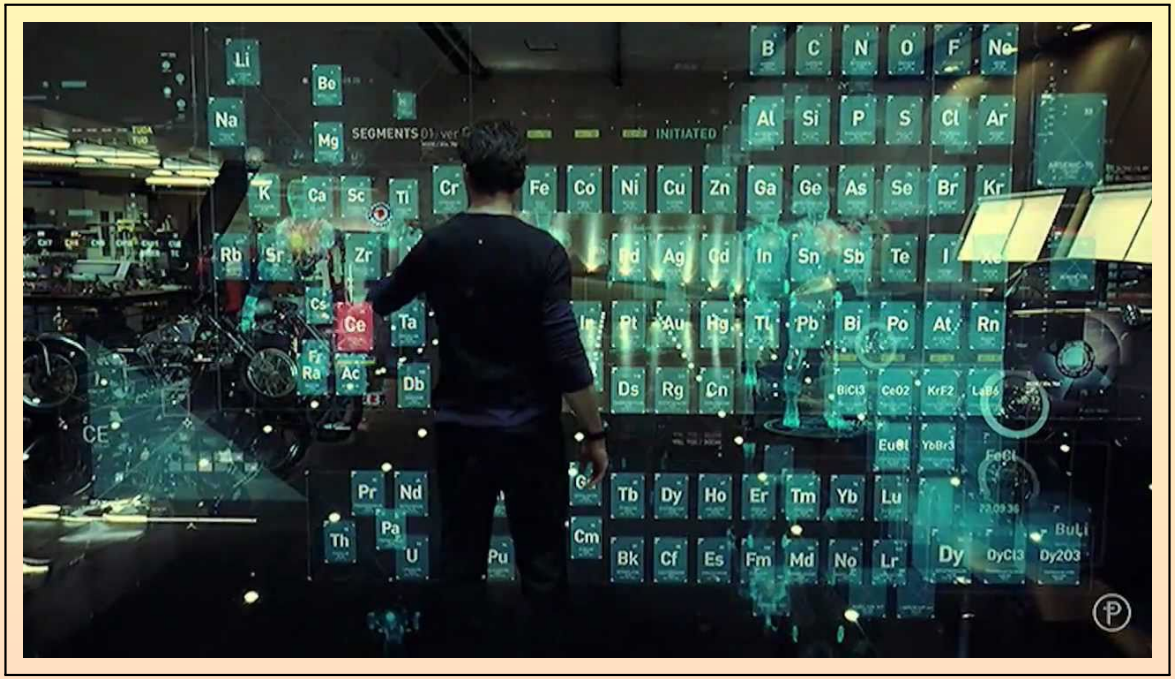


and what language shall we bootstrap this with!? We need it to be

- **Safe**: Provably Correct Memory Accesses
- **Performant**: (Fast Build and Run of Native Code)

Iron Man like Systems and Interfaces?

How do we turn software engineers into Tony Starks? :)

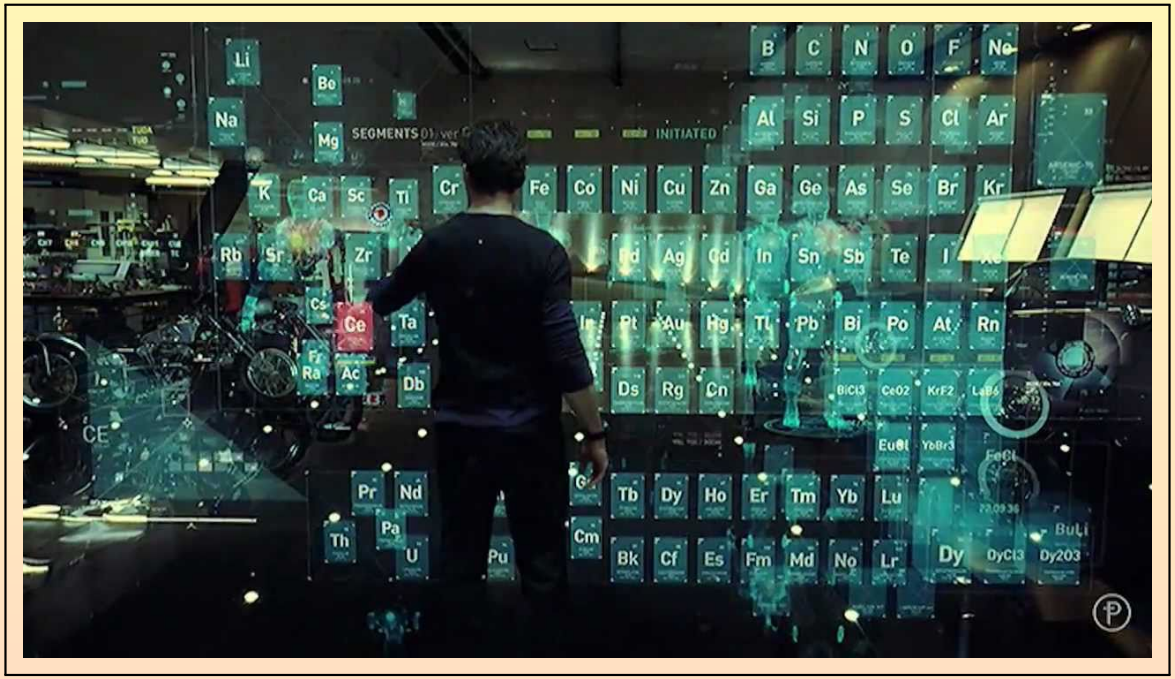


and what language shall we bootstrap this with!? We need it to be

- **Safe**: Provably Correct Memory Accesses
- **Performant**: (Fast Build and Run of Native Code)
- **Effective**: Easy, Expressive, Creative, Joyful

Iron Man like Systems and Interfaces?

How do we turn software engineers into Tony Starks? :)

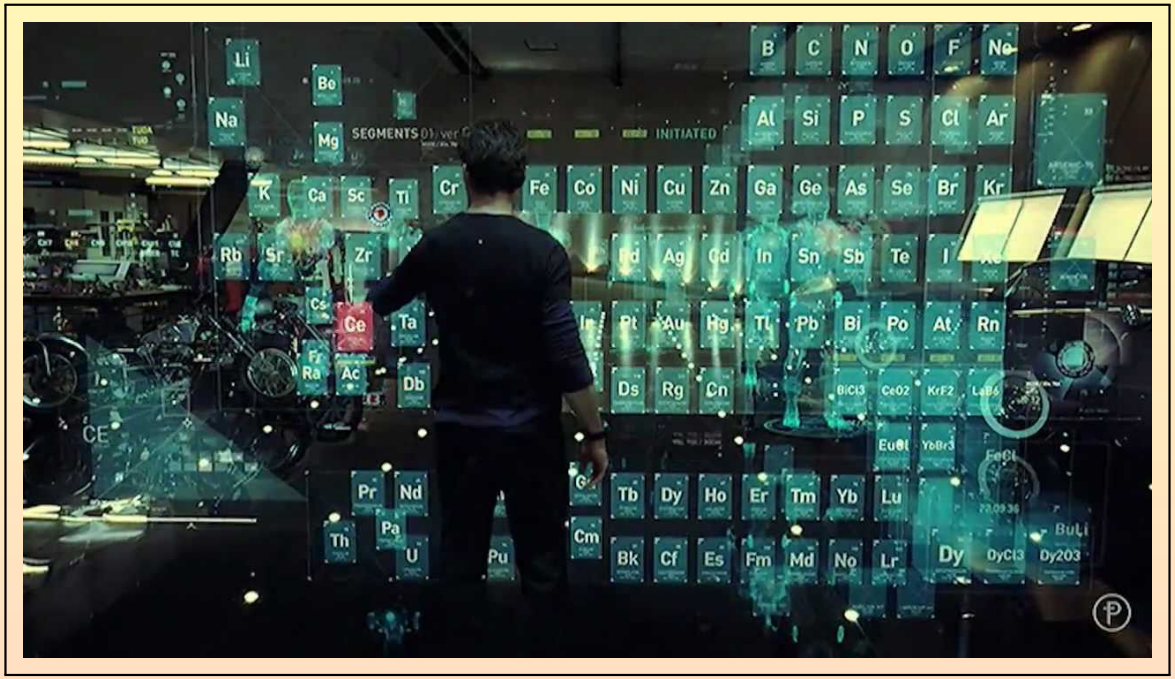


and what language shall we bootstrap this with!? We need it to be

- **Safe**: Provably Correct Memory Accesses
- **Performant**: (Fast Build and Run of Native Code)
- **Effective**: Easy, Expressive, Creative, Joyful
- **Other**: Only 2 out of 3

Iron Man like Systems and Interfaces?

How do we turn software engineers into Tony Starks? :)



and what language shall we bootstrap this with!? We need it to be

- **Safe**: Provably Correct Memory Accesses
- **Performant**: (Fast Build and Run of Native Code)
- **Effective**: Easy, Expressive, Creative, Joyful
- **Other**: Only 2 out of 3
- **D**: 3 out 3!

Leading Questions

- Why do systems *keep on* crashing?

Leading Questions

- Why do systems *keep on* crashing?
- Why so many languages: SAAB SimC uses C/C++, Fortran, Ada, Python, Shell, Make, . . . =>

Leading Questions

- Why do systems *keep on* crashing?
- Why so many languages: SAAB SimC uses C/C++, Fortran, Ada, Python, Shell, Make, . . . =>
- Restrictive languages, such as Ada, looses all its safety in the wrapping layer.

Leading Questions

- Why do systems *keep on* crashing?
- Why so many languages: SAAB SimC uses C/C++, Fortran, Ada, Python, Shell, Make, . . . =>
- Restrictive languages, such as Ada, looses all its safety in the wrapping layer.
- Why am I doing this?

Leading Questions

- Why do systems *keep on* crashing?
- Why so many languages: SAAB SimC uses C/C++, Fortran, Ada, Python, Shell, Make, . . . =>
- Restrictive languages, such as Ada, looses all its safety in the wrapping layer.
- Why am I doing this?
  - Lifes too short for bad code

Leading Questions

- Why do systems *keep on* crashing?
- Why so many languages: SAAB SimC uses C/C++, Fortran, Ada, Python, Shell, Make, . . . =>
- Restrictive languages, such as Ada, looses all its safety in the wrapping layer.
- Why am I doing this?
  - Lifes too short for bad code
- Either stick with it, or try to change it:

Language Trend

- Statically Inferred Languages

Language Trend

- Statically Inferred Languages
- Game Industry: “To few good C++ developers”. Why?

Language Trend

- Statically Inferred Languages
- Game Industry: "To few good C++ developers". Why?
- Which fault is it—developers or language designers?

Language Trend

- Statically Inferred Languages
- Game Industry: “To few good C++ developers”. Why?
- Which fault is it—developers or language designers?
- Mozilla: Servo Layout Engine using Rust

Language Trend

- Statically Inferred Languages
- Game Industry: “To few good C++ developers”. Why?
- Which fault is it—developers or language designers?
- Mozilla: Servo Layout Engine using Rust
- Microsoft Announcement: “We are planning a new language that is”

Language Trend

- Statically Inferred Languages
- Game Industry: “To few good C++ developers”. Why?
- Which fault is it—developers or language designers?
- Mozilla: Servo Layout Engine using Rust
- Microsoft Announcement: “We are planning a new language that is”
  - “as fast as C++” and

Language Trend

- Statically Inferred Languages
- Game Industry: “To few good C++ developers”. Why?
- Which fault is it—developers or language designers?
- Mozilla: Servo Layout Engine using Rust
- Microsoft Announcement: “We are planning a new language that is”
  - “as fast as C++” and
  - “as simple as C#”

Language Trend

- Statically Inferred Languages
- Game Industry: "To few good C++ developers". Why?
- Which fault is it—developers or language designers?
- Mozilla: Servo Layout Engine using Rust
- Microsoft Announcement: "We are planning a new language that is"
  - "as fast as C++" and
  - "as simple as C#"
- D already does this in company production code

Why should you learn about D?

Not Just Compacter Syntax but

- **Proving Ground** for Future C++

Why should you learn about D?

Not Just Compacter Syntax but

- **Proving Ground** for Future C++
- **Just like Market Consumers**:

Why should you learn about D?

Not Just Compacter Syntax but

- **Proving Ground** for Future C++
- **Just like Market Consumers**:
- **Developer have Responsibility**:

Why should you learn about D?

Not Just Compacter Syntax but

- **Proving Ground** for Future C++
- **Just like Market Consumers**:
- **Developer have Responsibility**:
  - If we aren't aware

Why should you learn about D?

Not Just Compacter Syntax but

- **Proving Ground** for Future C++
- **Just like Market Consumers**:
- **Developer have Responsibility**:
  - If we aren't aware
  - We won't ask for it and

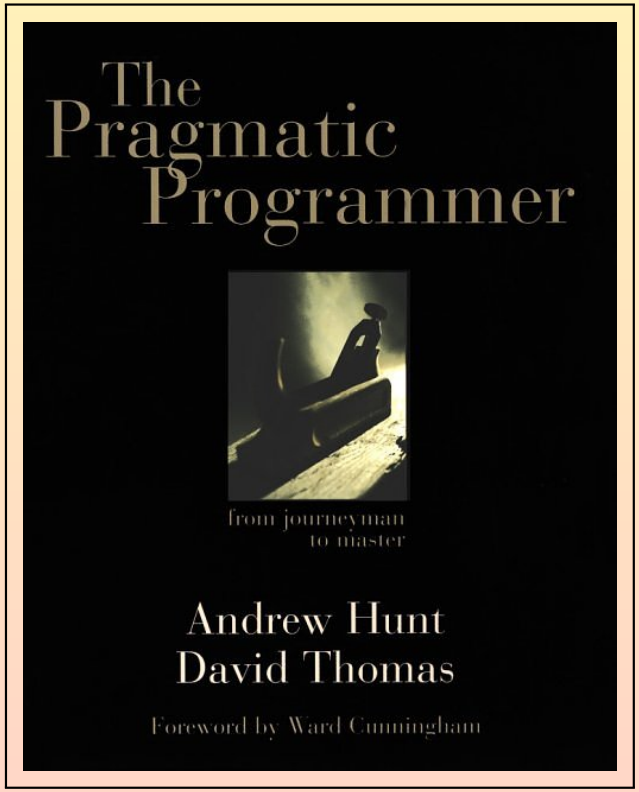
Why should you learn about D?

Not Just Compacter Syntax but

- **Proving Ground** for Future C++
- **Just like Market Consumers**:
- **Developer have Responsibility**:
  - If we aren't aware
  - We won't ask for it and
  - We will not get it (in languages in general)

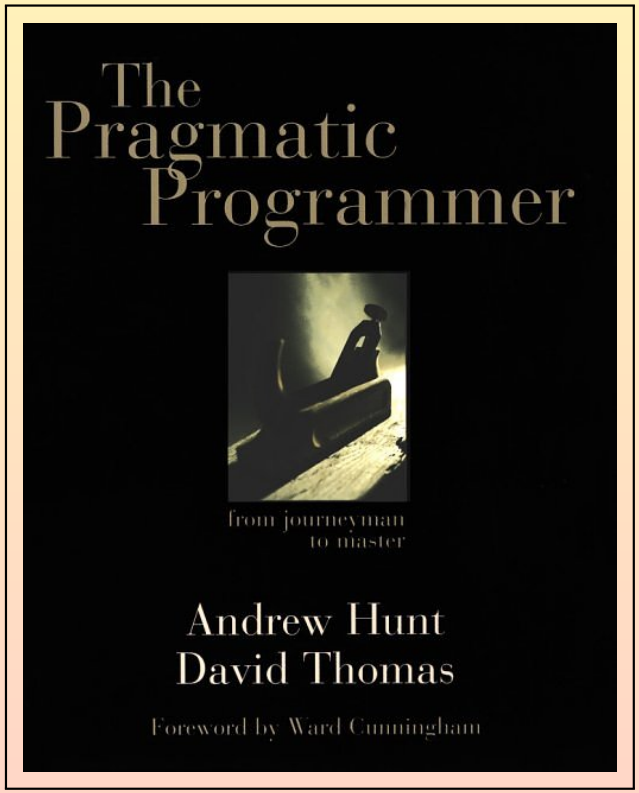
Democratic & Pragmatic Fundaments

- Gold is *not* everything that. . . — Big leaps are *not* made by the largest



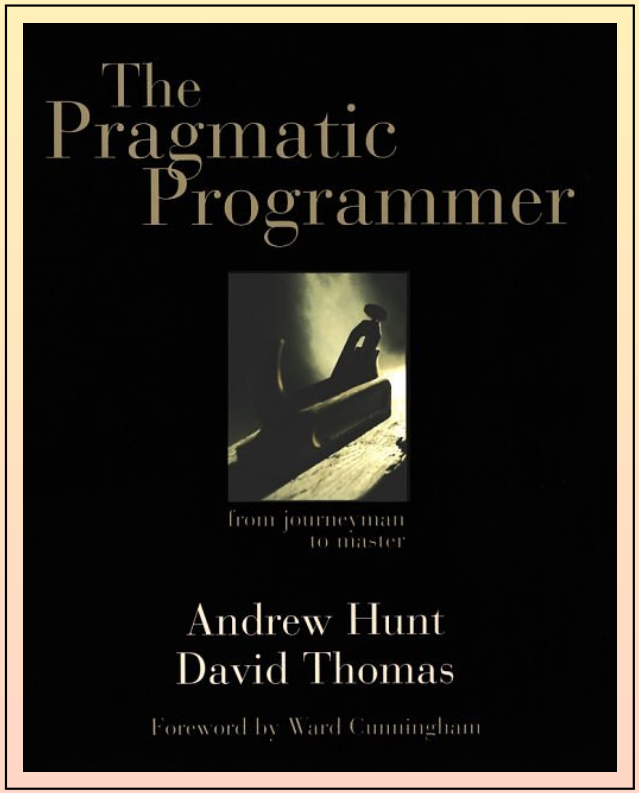
Democratic & Pragmatic Fundaments

- Gold is *not* everything that. . .— Big leaps are *not* made by the largest
- **Prestigeless Non-Company Controlled**: No more: *"Lets reinvent everything because we want to be different"*



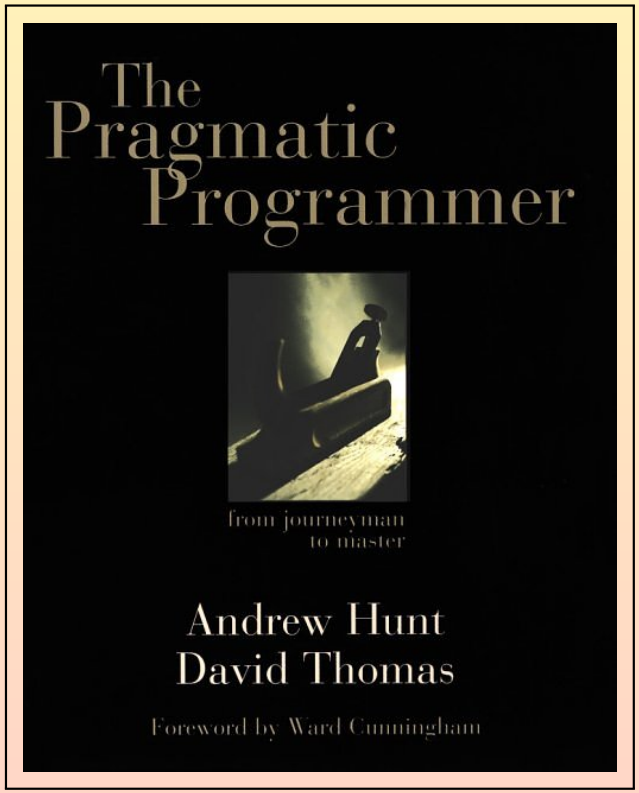
Democratic & Pragmatic Fundaments

- Gold is *not* everything that. . .— Big leaps are *not* made by the largest
- **Prestigeless Non-Company Controlled**: No more: *"Lets reinvent everything because we want to be different"*
- **No Priesthood** nor too Academic



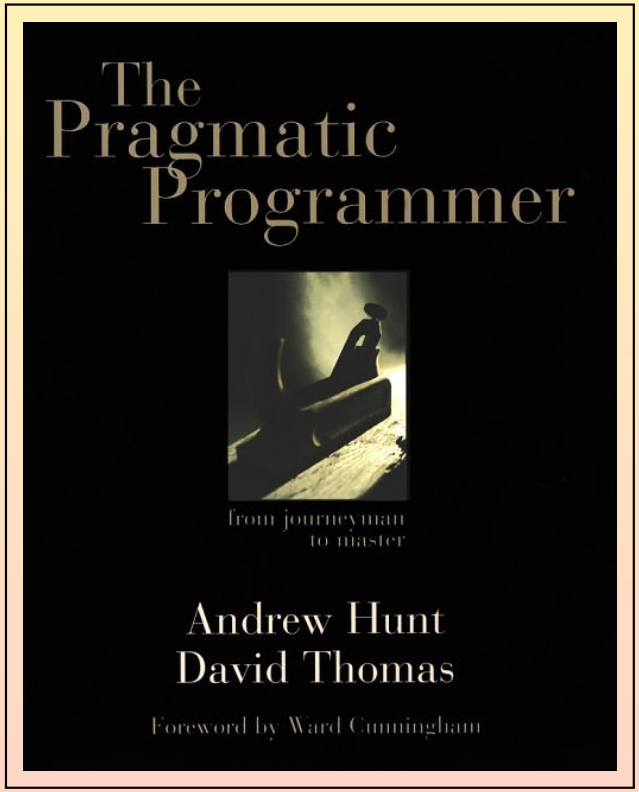
Democratic & Pragmatic Fundaments

- Gold is *not* everything that. . .— Big leaps are *not* made by the largest
- **Prestigeless Non-Company Controlled**: No more: *"Lets reinvent everything because we want to be different"*
- **No Priesthood** nor too Academic
- We have done our homework



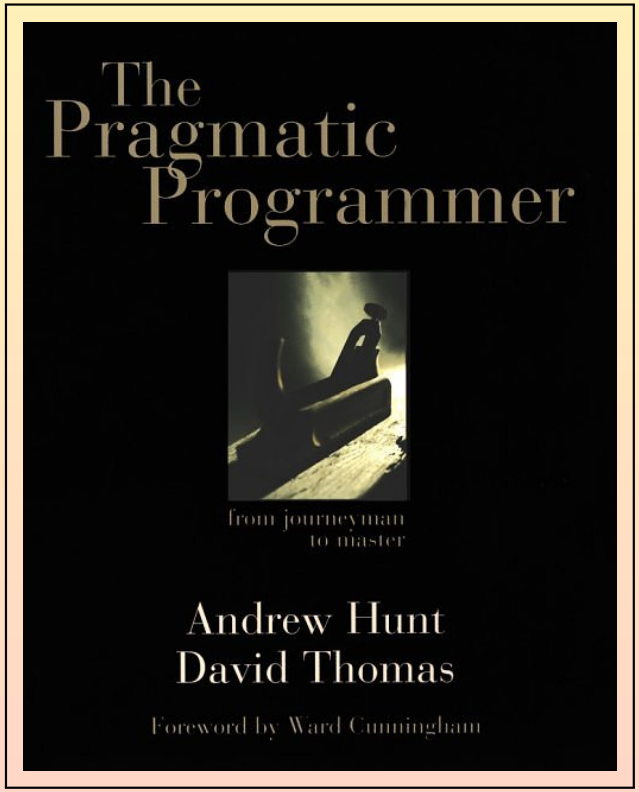
Democratic & Pragmatic Fundaments

- Gold is *not* everything that. . .— Big leaps are *not* made by the largest
- **Prestigeless Non-Company Controlled**: No more: *"Lets reinvent everything because we want to be different"*
- **No Priesthood** nor too Academic
- We have done our homework
- C/C++/C#-like to **ease transition**



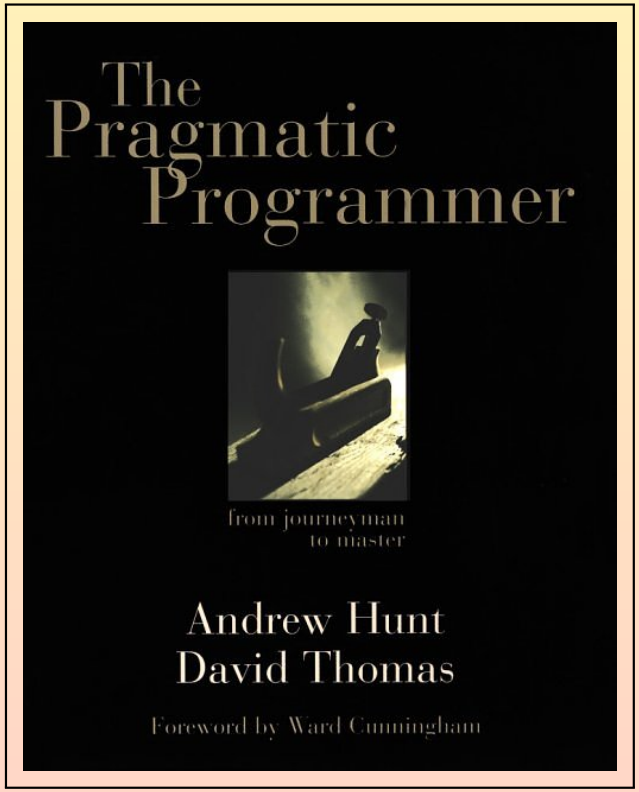
Democratic & Pragmatic Fundaments

- Gold is *not* everything that. . .— Big leaps are *not* made by the largest
- **Prestigeless Non-Company Controlled**: No more: *"Lets reinvent everything because we want to be different"*
- **No Priesthood** nor too Academic
- We have done our homework
- C/C++/C#-like to **ease transition**
- **Pragmatic**: Developers want to GTD



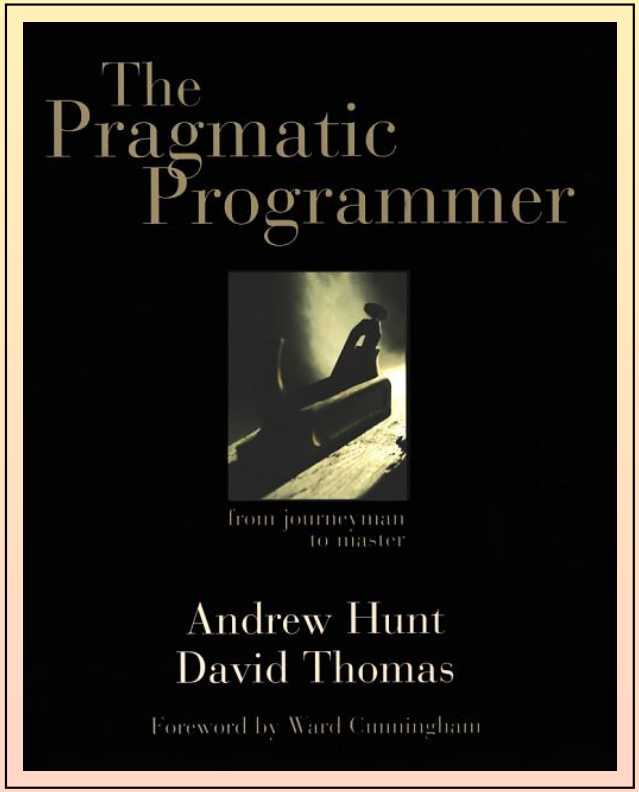
Democratic & Pragmatic Fundaments

- Gold is *not* everything that. . .— Big leaps are *not* made by the largest
- **Prestigeless Non-Company Controlled**: No more: *"Lets reinvent everything because we want to be different"*
- **No Priesthood** nor too Academic
- We have done our homework
- C/C++/C#-like to **ease transition**
- **Pragmatic**: Developers want to GTD
- **Democratic/Dynamic**: Key solutions voted for. All about *probabilities*.



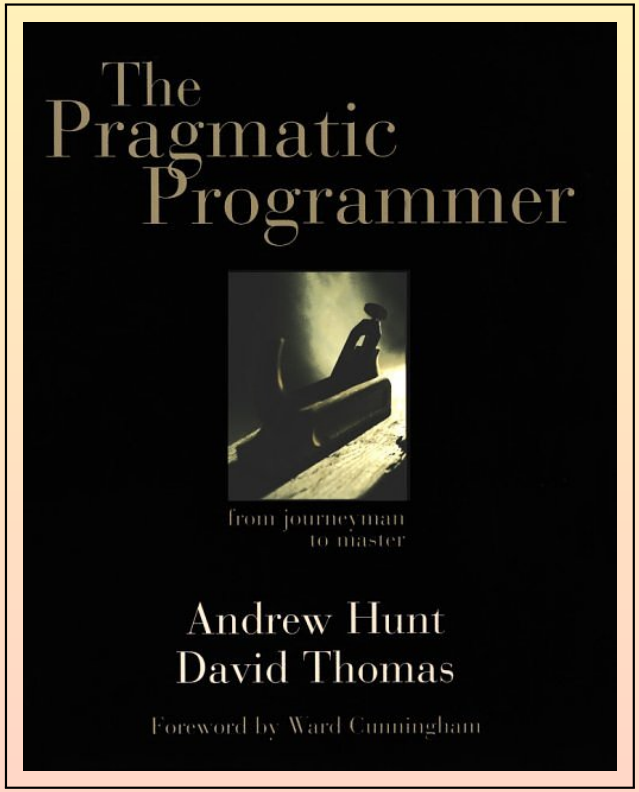
Democratic & Pragmatic Fundaments

- Gold is *not* everything that. . .— Big leaps are *not* made by the largest
- **Prestigeless Non-Company Controlled**: No more: *"Lets reinvent everything because we want to be different"*
- **No Priesthood** nor too Academic
- We have done our homework
- C/C++/C#-like to **ease transition**
- **Pragmatic**: Developers want to GTD
- **Democratic/Dynamic**: Key solutions voted for. All about *probabilities*.
- **Continuous Integration** on Github



Democratic & Pragmatic Fundaments

- Gold is *not* everything that. . .— Big leaps are *not* made by the largest
- **Prestigeless Non-Company Controlled**: No more: *"Lets reinvent everything because we want to be different"*
- **No Priesthood** nor too Academic
- We have done our homework
- C/C++/C#-like to **ease transition**
- **Pragmatic**: Developers want to GTD
- **Democratic/Dynamic**: Key solutions voted for. All about *probabilities*.
- **Continuous Integration** on Github
- Linux-style Deprecation Model



The Scientist and The Craftman — D's Magic Pair!

The Scientist and The Craftman — D's Magic Pair!

- History and Development of Walter Bright and D is very similar to that of Linus Torvalds and Linux.

Heart of D: Reuseable Software

Walter Bright: How can we

Heart of D: Reuseable Software

Walter Bright: How can we

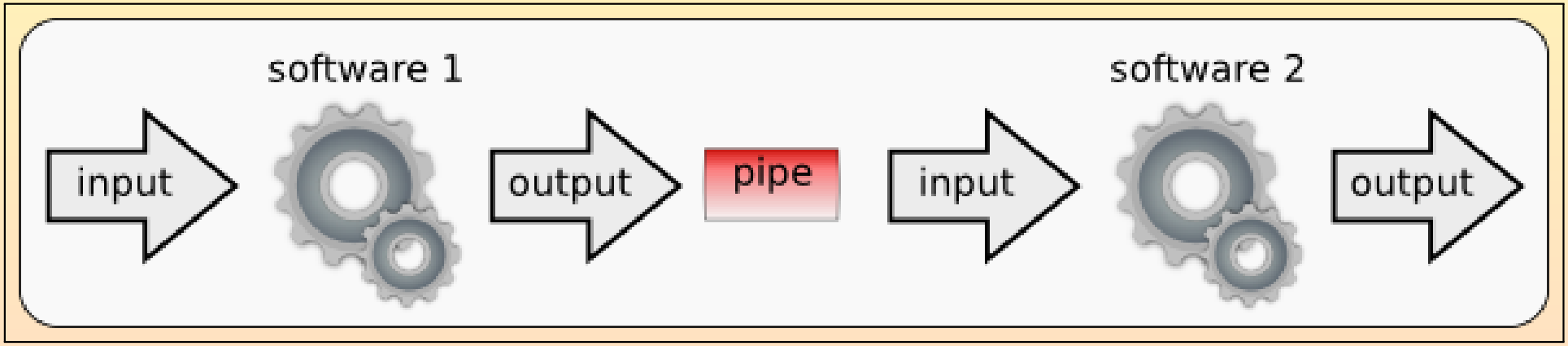
- Write truly reusable software components?

Heart of D: Reuseable Software

Walter Bright: How can we

- Write truly reusable software components?
- Escape “Whirlpool Programming”?

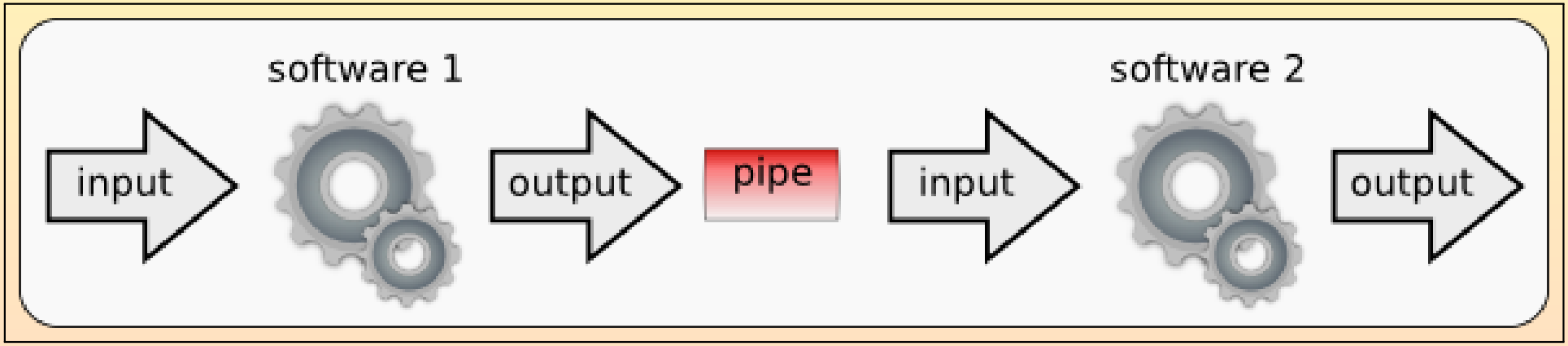
Heart of D: Component Programming Pipeline



Successful model in

- As Western text: Left-to-right

Heart of D: Component Programming Pipeline



Successful model in

- As Western text: Left-to-right
- Proof of Concept: UNIX Commands & Pipes

Heart of D: Component Programming Example

"Read lines, sort em and print em back":

```
import std.stdio, std.algorithm, std.range, std.array;
void main(string args[]) {
    Stdin.
        byLine(KeepTerminator.yes).
        map!(a => a.idup).
        array.
        sort.
        copy(stdout.lockingTextWriter());
}
```

as fast as handwritten C with nested loops and with manual error-checking but way simpler and more flexible

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**
  - Lazy/Delayed Evaluated means

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**
  - Lazy/Delayed Evaluated means
  - Return Calculation instead of Value

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**
  - Lazy/Delayed Evaluated means
  - Return Calculation instead of Value
  - Also called Higher-Order Range Pattern

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**
  - Lazy/Delayed Evaluated means
  - Return Calculation instead of Value
  - Also called Higher-Order Range Pattern
  - Compare with Monad

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**
  - Lazy/Delayed Evaluated means
  - Return Calculation instead of Value
  - Also called Higher-Order Range Pattern
  - Compare with Monad
  - Example:

```
writeln(x,y,z); // no heap
writeln(x~y~z); // use heap, revealed with -vgc
writeln(tuple(pair(x1,x2), y, z)); // composable, no heap
```

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**
  - Lazy/Delayed Evaluated means
  - Return Calculation instead of Value
  - Also called Higher-Order Range Pattern
  - Compare with Monad
  - Example:

```
writeln(x,y,z); // no heap
writeln(x~y~z); // use heap, revealed with -vgc
writeln(tuple(pair(x1,x2), y, z)); // composable, no heap
```

  - Concept easily confused with *value* ranges such as Ada's *Range Types*

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**
  - Lazy/Delayed Evaluated means
  - Return Calculation instead of Value
  - Also called Higher-Order Range Pattern
  - Compare with Monad
  - Example:

```
writeln(x,y,z); // no heap
writeln(x~y~z); // use heap, revealed with -vgc
writeln(tuple(pair(x1,x2), y, z)); // composable, no heap
```

  - Concept easily confused with *value* ranges such as Ada's *Range Types*
- **Arrays** and **Strings** Done Right

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**
  - Lazy/Delayed Evaluated means
  - Return Calculation instead of Value
  - Also called Higher-Order Range Pattern
  - Compare with Monad
  - Example:

```
writeln(x,y,z); // no heap
writeln(x~y~z); // use heap, revealed with -vgc
writeln(tuple(pair(x1,x2), y, z)); // composable, no heap
```

  - Concept easily confused with *value* ranges such as Ada's *Range Types*
- **Arrays** and **Strings** Done Right
- Uniform Call Syntax (**UFCS**)

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**
  - Lazy/Delayed Evaluated means
  - Return Calculation instead of Value
  - Also called Higher-Order Range Pattern
  - Compare with Monad
  - Example:

```
writeln(x,y,z); // no heap
writeln(x~y~z); // use heap, revealed with -vgc
writeln(tuple(pair(x1,x2), y, z)); // composable, no heap
```

  - Concept easily confused with *value* ranges such as Ada's *Range Types*
- **Arrays** and **Strings** Done Right
- Uniform Call Syntax (**UFCS**)
- Compile-Time Function Evaluation (**CTFE**)

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**
  - Lazy/Delayed Evaluated means
  - Return Calculation instead of Value
  - Also called Higher-Order Range Pattern
  - Compare with Monad
  - Example:

```
writeln(x,y,z); // no heap
writeln(x~y~z); // use heap, revealed with -vgc
writeln(tuple(pair(x1,x2), y, z)); // composable, no heap
```

  - Concept easily confused with *value* ranges such as Ada's *Range Types*
- **Arrays** and **Strings** Done Right
- Uniform Call Syntax (**UFCS**)
- Compile-Time Function Evaluation (**CTFE**)
- **Emptyness Propagation**: no if statments to check for null etc enables. Removes Error handling problem!

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**
  - Lazy/Delayed Evaluated means
  - Return Calculation instead of Value
  - Also called Higher-Order Range Pattern
  - Compare with Monad
  - Example:

```
writeln(x,y,z); // no heap
writeln(x~y~z); // use heap, revealed with -vgc
writeln(tuple(pair(x1,x2), y, z)); // composable, no heap
```

  - Concept easily confused with *value* ranges such as Ada's *Range Types*
- **Arrays** and **Strings** Done Right
- Uniform Call Syntax (**UFCS**)
- Compile-Time Function Evaluation (**CTFE**)
- **Emptyness Propagation**: no if statments to check for null etc enables. Removes Error handling problem!
- **Concepts**: (Type Predicates)

Heart of D: Component Programming realized by

- **Generics** (= reuse) made simple
- **Ranges**
  - Lazy/Delayed Evaluated means
  - Return Calculation instead of Value
  - Also called Higher-Order Range Pattern
  - Compare with Monad
  - Example:

```
writeln(x,y,z); // no heap
writeln(x~y~z); // use heap, revealed with -vgc
writeln(tuple(pair(x1,x2), y, z)); // composable, no heap
```

  - Concept easily confused with *value* ranges such as Ada's *Range Types*
- **Arrays** and **Strings** Done Right
- Uniform Call Syntax (**UFCS**)
- Compile-Time Function Evaluation (**CTFE**)
- **Emptyness Propagation**: no if statments to check for null etc enables. Removes Error handling problem!
- **Concepts**: (Type Predicates)
- **Inlineable** gives C/C++ performance

Uniform Call Syntax (UFCS)

- std.conv.to! makes API-integration transparent

Uniform Call Syntax (UFCS)

- std.conv.to! makes API-integration transparent
- `needle.findIn(haystack)` is more obvious than

Uniform Call Syntax (UFCS)

- std.conv.to! makes API-integration transparent
- `needle.findIn(haystack)` is more obvious than
- `findIn(needle, haystack)` and is used in

Uniform Call Syntax (UFCS)

- std.conv.to! makes API-integration transparent
- `needle.findIn(haystack)` is more obvious than
- `findIn(needle, haystack)` and is used in
- module `msgpack-d` as

Uniform Call Syntax (UFCS)

- std.conv.to! makes API-integration transparent
- `needle.findIn(haystack)` is more obvious than
- `findIn(needle, haystack)` and is used in
- module `msgpack-d` as
  - `value.pack`

Uniform Call Syntax (UFCS)

- std.conv.to! makes API-integration transparent
- `needle.findIn(haystack)` is more obvious than
- `findIn(needle, haystack)` and is used in
- module `msgpack-d` as
  - `value.pack`
  - `value.unpack`

Uniform Call Syntax (UFCS)

- std.conv.to! makes API-integration transparent
- `needle.findIn(haystack)` is more obvious than
- `findIn(needle, haystack)` and is used in
- module `msgpack-d` as
  - `value.pack`
  - `value.unpack`
- Chained Left-to-Right: `data.doThis.doThat;`

Uniform Call Syntax (UFCS)

- std.conv.to! makes API-integration transparent
- `needle.findIn(haystack)` is more obvious than
- `findIn(needle, haystack)` and is used in
- module `msgpack-d` as
  - `value.pack`
  - `value.unpack`
- Chained Left-to-Right: `data.doThis.doThat;`
- Ruby-Style: `"/bin/ls".isFile`

Concepts & Ranges

- AI The Movei Robotic Spare Arm Analog: Body is Range and Arm is Algorithm/Range

Concepts & Ranges

- AI The Movei Robotic Spare Arm Analog: Body is Range and Arm is Algorithm/Range
- Algorithms are the axioms and always come first

Concepts & Ranges

- AI The Movei Robotic Spare Arm Analog: Body is Range and Arm is Algorithm/Range
- Algorithms are the axioms and always come first
  - `reverse`

Concepts & Ranges

- AI The Movei Robotic Spare Arm Analog: Body is Range and Arm is Algorithm/Range
- Algorithms are the axioms and always come first
  - `reverse`
  - `editDistance`

Concepts & Ranges

- AI The Movei Robotic Spare Arm Analog: Body is Range and Arm is Algorithm/Range
- Algorithms are the axioms and always come first
  - `reverse`
  - `editDistance`
  - `binarySearch`, `quickSort`

Concepts & Ranges

- AI The Movei Robotic Spare Arm Analog: Body is Range and Arm is Algorithm/Range
- Algorithms are the axioms and always come first
  - `reverse`
  - `editDistance`
  - `binarySearch`, `quickSort`
- Concepts/Ranges (Minimal Data Interface Requirements) follow that groups

Concepts & Ranges

- AI The Movei Robotic Spare Arm Analog: Body is Range and Arm is Algorithm/Range
- Algorithms are the axioms and always come first
  - `reverse`
  - `editDistance`
  - `binarySearch`, `quickSort`
- Concepts/Ranges (Minimal Data Interface Requirements) follow that groups
- Containers

Concepts & Ranges

- AI The Movei Robotic Spare Arm Analog: Body is Range and Arm is Algorithm/Range
- Algorithms are the axioms and always come first
  - `reverse`
  - `editDistance`
  - `binarySearch`, `quickSort`
- Concepts/Ranges (Minimal Data Interface Requirements) follow that groups
- Containers
- Example: Check if a `range` is a **Palindrome**

```
bool isPalindrome(Range)(in Range x)
if (isBidirectionalRange!Range) {
    import std.algorithm: retro, equal;
    return x.retro.equal(x);
}
```

Concepts & Ranges

- Minimal set of requirements of interfaces to types gives

Concepts & Ranges

- Minimal set of requirements of interfaces to types gives
- Ranges and Interfaces.

Concepts & Ranges

- Minimal set of requirements of interfaces to types gives
- Ranges and Interfaces.
- Example `binarySearch` and `quickSort` works on anything that provides **RandomAccess**.

Concepts & Ranges

- Minimal set of requirements of interfaces to types gives
- Ranges and Interfaces.
- Example `binarySearch` and `quickSort` works on anything that provides **RandomAccess**.
- Distinguish between

Concepts & Ranges

- Minimal set of requirements of interfaces to types gives
- Ranges and Interfaces.
- Example `binarySearch` and `quickSort` works on anything that provides **RandomAccess**.
- Distinguish between
  - **std.range**: return calculations (like C++ expression templates but way simpler)

Concepts & Ranges

- Minimal set of requirements of interfaces to types gives
- Ranges and Interfaces.
- Example `binarySearch` and `quickSort` works on anything that provides **RandomAccess**.
- Distinguish between
  - **std.range**: return calculations (like C++ expression templates but way simpler)
  - **std.algorithm**: return values

Concepts & Ranges

- Minimal set of requirements of interfaces to types gives
- Ranges and Interfaces.
- Example `binarySearch` and `quickSort` works on anything that provides **RandomAccess**.
- Distinguish between
  - **std.range**: return calculations (like C++ expression templates but way simpler)
  - **std.algorithm**: return values
- A must need in Linear Algebra Packages

Concepts & Ranges

- More Compact/Efficient Encoding (Development) of Algorithms and their Variants (in Phobos)

Concepts & Ranges

- More Compact/Efficient Encoding (Development) of Algorithms and their Variants (in Phobos)
- Unique Feature of D: Ranges preserve powers when possible.

```
auto y = x.map("a*a")}.
```

If `x` is random access `y` also is

Concepts & Ranges

- More Compact/Efficient Encoding (Development) of Algorithms and their Variants (in Phobos)
- Unique Feature of D: Ranges preserve powers when possible.

```
auto y = x.map("a*a")}.
```

  If `x` is random access `y` also is

- `std.array`: `array` triggers computation if input is not RandomAccess

Concepts & Ranges

- More Compact/Efficient Encoding (Development) of Algorithms and their Variants (in Phobos)
- Unique Feature of D: Ranges preserve powers when possible.

```
auto y = x.map("a*a")}.
```

  If `x` is random access `y` also is

- `std.array`: `array` triggers computation if input is not RandomAccess
- See Andrei Alexandrescous article "Forward is Not Enough"

Concepts & Ranges

- More Compact/Efficient Encoding (Development) of Algorithms and their Variants (in Phobos)
- Unique Feature of D: Ranges preserve powers when possible.

```
auto y = x.map("a*a")}.
```

  If `x` is random access `y` also is

- `std.array`: `array` triggers computation if input is not RandomAccess
- See Andrei Alexandrescous article "Forward is Not Enough"
- Ranges and algorithms are more generic than others (STL). For example search/find can search for multiple (variadic) keys.

Concepts & Ranges

- More Compact/Efficient Encoding (Development) of Algorithms and their Variants (in Phobos)
- Unique Feature of D: Ranges preserve powers when possible.

```
auto y = x.map("a*a")}.
```

  If `x` is random access `y` also is

- `std.array`: `array` triggers computation if input is not RandomAccess
- See Andrei Alexandrescous article "Forward is Not Enough"
- Ranges and algorithms are more generic than others (STL). For example search/find can search for multiple (variadic) keys.
- TODO: Visualizations of behaviour would be nice

List of Range Creations 1

- `retro`: Iterates a bidirectional range *backwards*.

List of Range Creations 1

- **`retro`**: Iterates a bidirectional range *backwards*.
- **`stride`**: Iterates a range with *stride* `n`.

List of Range Creations 1

- **`retro`**: Iterates a bidirectional range *backwards*.
- **`stride`**: Iterates a range with *stride* `n`.
- **`chain`**: *Concatenates* several ranges into a single range.

List of Range Creations 1

- `retro`: Iterates a bidirectional range *backwards*.
- `stride`: Iterates a range with *stride* `n`.
- `chain`: *Concatenates* several ranges into a single range.
- `roundRobin`: Given n ranges, creates a new range that return the n first elements of each range, in turn, then the second element of each range, and so on, in a round-robin fashion.

List of Range Creations 1

- **`retro`**: Iterates a bidirectional range *backwards*.
- **`stride`**: Iterates a range with *stride* `n`.
- **`chain`**: *Concatenates* several ranges into a single range.
- **`roundRobin`**: Given n ranges, creates a new range that return the n first elements of each range, in turn, then the second element of each range, and so on, in a round-robin fashion.
- **`radial`**: Given a random-access range and a starting point, creates a range that alternately returns the next left and next right element to the starting point.

List of Range Creations 1

- **`retro`**: Iterates a bidirectional range *backwards*.
- **`stride`**: Iterates a range with *stride* `n`.
- **`chain`**: *Concatenates* several ranges into a single range.
- **`roundRobin`**: Given n ranges, creates a new range that return the n first elements of each range, in turn, then the second element of each range, and so on, in a round-robin fashion.
- **`radial`**: Given a random-access range and a starting point, creates a range that alternately returns the next left and next right element to the starting point.
- **`take`**: Creates a sub-range consisting of only up to the first n elements of the given range.

List of Range Creations 1

- **`retro`**: Iterates a bidirectional range *backwards*.
- **`stride`**: Iterates a range with *stride* `n`.
- **`chain`**: *Concatenates* several ranges into a single range.
- **`roundRobin`**: Given n ranges, creates a new range that return the n first elements of each range, in turn, then the second element of each range, and so on, in a round-robin fashion.
- **`radial`**: Given a random-access range and a starting point, creates a range that alternately returns the next left and next right element to the starting point.
- **`take`**: Creates a sub-range consisting of only up to the first n elements of the given range.
- **`takeExactly`**: Like take, but assumes the given range actually has n elements, and therefore also defines the length property.

List of Range Creations 1

- **`retro`**: Iterates a bidirectional range *backwards*.
- **`stride`**: Iterates a range with *stride* `n`.
- **`chain`**: *Concatenates* several ranges into a single range.
- **`roundRobin`**: Given n ranges, creates a new range that return the n first elements of each range, in turn, then the second element of each range, and so on, in a round-robin fashion.
- **`radial`**: Given a random-access range and a starting point, creates a range that alternately returns the next left and next right element to the starting point.
- **`take`**: Creates a sub-range consisting of only up to the first n elements of the given range.
- **`takeExactly`**: Like take, but assumes the given range actually has n elements, and therefore also defines the length property.
- **`takeOne`**: Creates a random-access range consisting of exactly the first element of the given range.

List of Range Creations 1

- **`retro`**: Iterates a bidirectional range *backwards*.
- **`stride`**: Iterates a range with *stride* `n`.
- **`chain`**: *Concatenates* several ranges into a single range.
- **`roundRobin`**: Given n ranges, creates a new range that return the n first elements of each range, in turn, then the second element of each range, and so on, in a round-robin fashion.
- **`radial`**: Given a random-access range and a starting point, creates a range that alternately returns the next left and next right element to the starting point.
- **`take`**: Creates a sub-range consisting of only up to the first n elements of the given range.
- **`takeExactly`**: Like take, but assumes the given range actually has n elements, and therefore also defines the length property.
- **`takeOne`**: Creates a random-access range consisting of exactly the first element of the given range.
- **`takeNone`**: Creates a random-access range consisting of zero elements of the given range.

List of Range Creations 2

- `drop`: Creates the range that results from discarding the first n elements from the given range.

List of Range Creations 2

- `drop`: Creates the range that results from discarding the first n elements from the given range.
- `dropExactly`: Creates the range that results from discarding exactly n of the first elements from the given range.

List of Range Creations 2

- `drop`: Creates the range that results from discarding the first n elements from the given range.
- `dropExactly`: Creates the range that results from discarding exactly n of the first elements from the given range.
- `dropOne`: Creates the range that results from discarding the first elements from the given range.

List of Range Creations 2

- `drop`: Creates the range that results from discarding the first n elements from the given range.
- `dropExactly`: Creates the range that results from discarding exactly n of the first elements from the given range.
- `dropOne`: Creates the range that results from discarding the first elements from the given range.
- `repeat`: Creates a range that consists of a single element repeated n times, or an infinite range repeating that element indefinitely.

List of Range Creations 2

- **`drop`**: Creates the range that results from discarding the first n elements from the given range.
- **`dropExactly`**: Creates the range that results from discarding exactly n of the first elements from the given range.
- **`dropOne`**: Creates the range that results from discarding the first elements from the given range.
- **`repeat`**: Creates a range that consists of a single element repeated n times, or an infinite range repeating that element indefinitely.
- **`cycle`**: Creates an infinite range that repeats the given forward range indefinitely. Look ma no types!:

```
import std.range;
auto b = [1,2,3,4]; // buffer
auto cb = b.cycle; // circular buffer
```

List of Range Creations 2

- **`drop`**: Creates the range that results from discarding the first n elements from the given range.
- **`dropExactly`**: Creates the range that results from discarding exactly n of the first elements from the given range.
- **`dropOne`**: Creates the range that results from discarding the first elements from the given range.
- **`repeat`**: Creates a range that consists of a single element repeated n times, or an infinite range repeating that element indefinitely.
- **`cycle`**: Creates an infinite range that repeats the given forward range indefinitely. Look ma no types!:

```
import std.range;
auto b = [1,2,3,4]; // buffer
auto cb = b.cycle; // circular buffer
```

- **`zip`**: Given n ranges, creates a range that successively returns a tuple of all the first elements, a tuple of all the second elements, etc.

List of Range Creations 2

- **`drop`**: Creates the range that results from discarding the first n elements from the given range.
- **`dropExactly`**: Creates the range that results from discarding exactly n of the first elements from the given range.
- **`dropOne`**: Creates the range that results from discarding the first elements from the given range.
- **`repeat`**: Creates a range that consists of a single element repeated n times, or an infinite range repeating that element indefinitely.
- **`cycle`**: Creates an infinite range that repeats the given forward range indefinitely. Look ma no types!:

```
import std.range;
auto b = [1,2,3,4]; // buffer
auto cb = b.cycle; // circular buffer
```

- **`zip`**: Given n ranges, creates a range that successively returns a tuple of all the first elements, a tuple of all the second elements, etc.
- **`lockstep`**: Iterates n ranges in lockstep, for use in a foreach loop. Similar to zip, except that lockstep is designed especially for foreach

List of Range Creations 2

- **`drop`**: Creates the range that results from discarding the first n elements from the given range.
- **`dropExactly`**: Creates the range that results from discarding exactly n of the first elements from the given range.
- **`dropOne`**: Creates the range that results from discarding the first elements from the given range.
- **`repeat`**: Creates a range that consists of a single element repeated n times, or an infinite range repeating that element indefinitely.
- **`cycle`**: Creates an infinite range that repeats the given forward range indefinitely. Look ma no types!:

```
import std.range;
auto b = [1,2,3,4]; // buffer
auto cb = b.cycle; // circular buffer
```

- **`zip`**: Given n ranges, creates a range that successively returns a tuple of all the first elements, a tuple of all the second elements, etc.
- **`lockstep`**: Iterates n ranges in lockstep, for use in a foreach loop. Similar to zip, except that lockstep is designed especially for foreach

List of Range Creations 3

- `sequence`: Similar to recurrence, except that a random-access range is created.

List of Range Creations 3

- `sequence`: Similar to recurrence, except that a random-access range is created.
- `iota`: Creates a range consisting of numbers between a starting point and ending point, spaced apart by a given interval.

List of Range Creations 3

- `sequence`: Similar to recurrence, except that a random-access range is created.
- `iota`: Creates a range consisting of numbers between a starting point and ending point, spaced apart by a given interval.
- `frontTransversal`: Creates a range that iterates over the first elements of the given ranges.

List of Range Creations 3

- `sequence`: Similar to recurrence, except that a random-access range is created.
- `iota`: Creates a range consisting of numbers between a starting point and ending point, spaced apart by a given interval.
- `frontTransversal`: Creates a range that iterates over the first elements of the given ranges.
- `transversal`: Creates a range that iterates over the n'th elements of the given random-access ranges.

List of Range Creations 3

- `sequence`: Similar to recurrence, except that a random-access range is created.
- `iota`: Creates a range consisting of numbers between a starting point and ending point, spaced apart by a given interval.
- `frontTransversal`: Creates a range that iterates over the first elements of the given ranges.
- `transversal`: Creates a range that iterates over the n'th elements of the given random-access ranges.
- `indexed`: Creates a range that offers a view of a given range as though its elements were reordered according to a given range of indices.

List of Range Creations 3

- `sequence`: Similar to recurrence, except that a random-access range is created.
- `iota`: Creates a range consisting of numbers between a starting point and ending point, spaced apart by a given interval.
- `frontTransversal`: Creates a range that iterates over the first elements of the given ranges.
- `transversal`: Creates a range that iterates over the n'th elements of the given random-access ranges.
- `indexed`: Creates a range that offers a view of a given range as though its elements were reordered according to a given range of indices.
- `chunks`: Creates a range that returns fixed-size chunks of the original range.

List of Range Creations 3

- `sequence`: Similar to recurrence, except that a random-access range is created.
- `iota`: Creates a range consisting of numbers between a starting point and ending point, spaced apart by a given interval.
- `frontTransversal`: Creates a range that iterates over the first elements of the given ranges.
- `transversal`: Creates a range that iterates over the n'th elements of the given random-access ranges.
- `indexed`: Creates a range that offers a view of a given range as though its elements were reordered according to a given range of indices.
- `chunks`: Creates a range that returns fixed-size chunks of the original range.
- `only`: Creates a range that iterates over a single value.

List of Upcoming Range Creations

- `enumerate`: (Just merged)

List of Upcoming Range Creations

- `enumerate`: (Just merged)
- `cache`: Inject caching of (in-pipeline) calculations of intermediate results. (Just merged)

List of Upcoming Range Creations

- `enumerate`: (Just merged)
- `cache`: Inject caching of (in-pipeline) calculations of intermediate results. (Just merged)
- `each`: (PR on review)

List of Upcoming Range Creations

- `enumerate`: (Just merged)
- `cache`: Inject caching of (in-pipeline) calculations of intermediate results. (Just merged)
- `each`: (PR on review)
- `chunkBy`: (PR on review)

Data Semantics

- Like in C#

Data Semantics

- Like in C#
- Clear Distingushing between

Data Semantics

- Like in C#
- Clear Distingushing between
  - *Value* Semantics: `struct`

Data Semantics

- Like in C#
- Clear Distingushing between
  - *Value* Semantics: `struct`
  - *Reference* Semantics: `class`

Data Semantics

- Like in C#
- Clear Distingushing between
  - *Value* Semantics: `struct`
  - *Reference* Semantics: `class`
- Removes need for C++'s pointer member access operator: `->`

Safer and Easier Class Polymorphism

- *All* members functions by default virtual

Safer and Easier Class Polymorphism

- *All* members functions by default virtual
- No `virtual` keyword needed

Safer and Easier Class Polymorphism

- *All* members functions by default virtual
- No `virtual` keyword needed
- Overriding must be explicit through `override`

Safer and Easier Class Polymorphism

- *All* members functions by default virtual
- No `virtual` keyword needed
- Overriding must be explicit through `override`
- Member qualifier `final` makes them non-virtual

Safer and Easier Class Polymorphism

- *All* members functions by default virtual
- No `virtual` keyword needed
- Overriding must be explicit through `override`
- Member qualifier `final` makes them non-virtual
- Final members can't be overridden in sub-classes

Basic Needs Builtin

- Strings: `string`

Basic Needs Builtin

- Strings: `string`
- Arrays: `ubyte[]`

Basic Needs Builtin

- Strings: `string`
- Arrays: `ubyte[]`
- Maps (Associative Arrays): `ubyte[string]`

Arrays Done Right fixes “C’s Largest Mistake”:

Arrays are “Fat” Pointers (Pointer + Length):

Arrays Done Right fixes "C's Largest Mistake":

Arrays are "Fat" Pointers (Pointer + Length):

- **Static** (Stack) Compile-Time Checked with *Value*-Semantics:

```
int[2] a;
int[3] b;
ubyte[\$] b = [1,2,3]; // type given, length inferred (v2.066)
a[] = b[]; // mismatched array lengths, 2 and 3
```

Arrays Done Right fixes "C's Largest Mistake":

Arrays are "Fat" Pointers (Pointer + Length):

- **Static** (Stack) Compile-Time Checked with *Value*-Semantics:

```
int[2] a;
int[3] b;
ubyte[\$] b = [1,2,3]; // type given, length inferred (v2.066)
a[] = b[]; // mismatched array lengths, 2 and 3
```

- **Dynamic** (GC-Heap) Run-Time Checked with *Reference*-Semantics:

```
int[2] a;
auto b = [1,2,3];
a[] = b[]; // lengths don't match for array copy, 2 = 3
```

  GC-escapeable from function calls makes use easy and robust.

Arrays Done Right fixes "C's Largest Mistake":

Arrays are "Fat" Pointers (Pointer + Length):

- **Static** (Stack) Compile-Time Checked with *Value*-Semantics:

```
int[2] a;
int[3] b;
ubyte[\$] b = [1,2,3]; // type given, length inferred (v2.066)
a[] = b[]; // mismatched array lengths, 2 and 3
```

- **Dynamic** (GC-Heap) Run-Time Checked with *Reference*-Semantics:

```
int[2] a;
auto b = [1,2,3];
a[] = b[]; // lengths don't match for array copy, 2 = 3
```

  GC-escapeable from function calls makes use easy and robust.

- Dynamic **Scoped** (Heap):

```
scoped auto a = [1,2];
```

Arrays Builtin & Done Right

- Easier Transition from C/C++ with:

Arrays Builtin & Done Right

- Easier Transition from C/C++ with:
  - **C-Style Arrays Compatible**: `int a[2];`

Arrays Builtin & Done Right

- Easier Transition from C/C++ with:
  - **C-Style Arrays Compatible**: `int a[2];`
  - **Clever Leading Principle**: "If it compiles its behaviour will be equal to C (and C++)".

Arrays Builtin & Done Right

- Easier Transition from C/C++ with:
  - **C-Style Arrays Compatible**: `int a[2];`
  - **Clever Leading Principle**: "If it compiles its behaviour will be equal to C (and C++)".
- Compare with C/C++11/C++14's Complexity: `T[]`, C99 VLA, `std::vector`, `std::array`, `std::valarray`, `std::dynarray`, etc.

Arrays Builtin & Done Right

- Easier Transition from C/C++ with:
  - **C-Style Arrays Compatible**: `int a[2];`
  - **Clever Leading Principle**: "If it compiles its behaviour will be equal to C (and C++)".
- Compare with C/C++11/C++14's Complexity: `T[]`, C99 VLA, `std::vector`, `std::array`, `std::valarray`, `std::dynarray`, etc.
- Safe by default (even in release mode): 90 % of (browser) security holes are memory overruns.

Pragmatic Array Slice Semantics

Eventhough this, at first sight, seem inconsisent this is *really* what we want in pattern (regexp) matching algorithms.

- Pointer part decide "**null**ness" and Bool Conversion

```
static assert([].ptr != null);
static assert("ab"[$..$].ptr == null);
static assert(![]);
static assert(!null);
```

Pragmatic Array Slice Semantics

Eventhough this, at first sight, seem inconsisent this is *really* what we want in pattern (regexp) matching algorithms.

- Pointer part decide "**null**ness" and Bool Conversion

```
static assert([].ptr != null);
static assert("ab"[$..$].ptr == null);
static assert(![]);
static assert(!null);
```

- Zero-length slices are not null

```
static assert("ab"[0..0] == []);
static assert("ab"[$..$] == []);
static assert("ab"[$..$] !is null);
static assert("ab"[$..$] == null);
static assert("ab"[0..0] == "ab"[$..$]);
static assert("ab"[0..0]); // contextual hit (BOL)
static assert("a\nb"[2..2]); // contextual hit (beginning of
    second line)
static assert("ab"[$..$]); // contextual hit (EOL)
```

Associative Arrays (Maps) Builtin & Done Right

No need for separate construction of map values:

```
string[][int] tags;
tags[13] ~= "Alice";
```

Strings Builtin & Done Right

- Unicode Builtin: UTF-8/16/32 = `string` `wstring` `dstring` are by default `immutable` ranges. This enable for example

Strings Builtin & Done Right

- Unicode Builtin: UTF-8/16/32 = `string` `wstring` `dstring` are by default `immutable` ranges. This enable for example
- ASTs to refer directly to GC:ed source string (slice) instead of a heap-allocated copy

Strings Builtin & Done Right

- Unicode Builtin: UTF-8/16/32 = `string` `wstring` `dstring` are by default `immutable` ranges. This enable for example
- ASTs to refer directly to GC:ed source string (slice) instead of a heap-allocated copy
- No heap allocation of read-only copies

Strings Builtin & Done Right

- Unicode Builtin: UTF-8/16/32 = `string` `wstring` `dstring` are by default `immutable` ranges. This enable for example
- ASTs to refer directly to GC:ed source string (slice) instead of a heap-allocated copy
- No heap allocation of read-only copies
- D XML-parsers fastest in the world

String types are aware of their limits

- Distinction Between

String types are aware of their limits

- Distinction Between
  - String `string` and Character `char`: and

String types are aware of their limits

- Distinction Between
  - String `string` and Character `char`: and
  - Array of Bytes: `ubyte[]`

String types are aware of their limits

- Distinction Between
  - String `string` and Character `char`: and
  - Array of Bytes: `ubyte[]`
- String Overview

Letter Type	*String Type*	*Code Length*	*Range Concept*	*Algorithms*
`char`, `wchar`	`string`, `wstring`	Variable	BiDirectional	Levenshtein Distance
`dchar`	`dstring`	Fixed	RandomAccess	Quick Sort

Iteration Done Right: foreach

- Like C++11 Range-Based for-loop but without need for `auto`:

Iteration Done Right: foreach

- Like C++11 Range-Based for-loop but without need for `auto`:
  - By Value:

```
foreach(element; range) {}
```

Iteration Done Right: foreach

- Like C++11 Range-Based for-loop but without need for `auto`:
  - By Value:

```
foreach(element; range) {}
```

  - By Reference:

```
foreach(ref element; range) {}
```

Iteration Done Right: foreach

- Like C++11 Range-Based for-loop but without need for `auto`:
  - By Value:

```
foreach(element; range) {}
```

  - By Reference:

```
foreach(ref element; range) {}
```

- Optional Statically Inferred Index (`size_t`):

```
foreach(index, element; range) {}
```

Iteration Done Right: foreach

- Like C++11 Range-Based for-loop but without need for `auto`:
  - By Value:

```
foreach(element; range) {}
```

  - By Reference:

```
foreach(ref element; range) {}
```

- Optional Statically Inferred Index (`size_t`):

```
foreach(index, element; range) {}
```

- Tuple Unpacking

```
foreach(index, tupleElementA, tupleElementB; rangeOfTuples) {}
```

Iteration Done Right: foreach

- Like C++11 Range-Based for-loop but without need for `auto`:
  - By Value:

```
foreach(element; range) {}
```

  - By Reference:

```
foreach(ref element; range) {}
```

- Optional Statically Inferred Index (`size_t`):

```
foreach(index, element; range) {}
```

- Tuple Unpacking

```
foreach(index, tupleElementA, tupleElementB; rangeOfTuples) {}
```

- To date – not a single classic `for` loop

Programming Styles Overview: Which pill to take?

- *Imperative (Blue)* Style (Side-Effects):

```
foreach (elt; range) {
    elt.doSideEffect;
}
```

Programming Styles Overview: Which pill to take?

- *Imperative (Blue)* Style (Side-Effects):

```
foreach (elt; range) {
    elt.doSideEffect;
}
```

- *Functional (Red)* Style (no Side-Effects):

```
range.map!"a*a";
```

Programming Styles Overview: Which pill to take?

- *Imperative (Blue)* Style (Side-Effects):

```
foreach (elt; range) {
    elt.doSideEffect;
}
```

- *Functional (Red)* Style (no Side-Effects):

```
range.map!"a*a";
```

- *Combined (Magenta)* Style:

```
foreach (elt; range.map!"a*a") {
    elt.doSideEffect;
}
```

Programming Styles Overview: Which pill to take?

- *Imperative (Blue)* Style (Side-Effects):

```
foreach (elt; range) {
    elt.doSideEffect;
}
```

- *Functional (Red)* Style (no Side-Effects):

```
range.map!"a*a";
```

- *Combined (Magenta)* Style:

```
foreach (elt; range.map!"a*a") {
    elt.doSideEffect;
}
```

- *Functional-Lambda* (White?) Style:

```
10.times!({ doSideEffect; });
```

Correctness: `unittest` builtin everywhere

- Simplicity!:

```
unittest { assert(something); }
```

Correctness: `unittest` builtin everywhere

- Simplicity!:

```
unittest { assert(something); }
```

- This unittest is the executed before main if flag `-unittest` is given to the compiler.

Correctness: `unittest` builtin everywhere

- Simplicity!:

```
unittest { assert(something); }
```

- This unittest is the executed before main if flag `-unittest` is given to the compiler.
- Number of tests scale a magnitude

Correctness: `unittest` builtin everywhere

- Simplicity!:

```
unittest { assert(something); }
```

- This unittest is the executed before main if flag `-unittest` is given to the compiler.
- Number of tests scale a magnitude
- Can be placed everywhere they belong including generic structs and classes!

Correctness: `unittest` builtin everywhere

- Simplicity!:

```
unittest { assert(something); }
```

- This unittest is the executed before main if flag `-unittest` is given to the compiler.
- Number of tests scale a magnitude
- Can be placed everywhere they belong including generic structs and classes!
- Great Leap in Stability/Correctness

Ada-Style Value Range Propagation

Thanks to built in static analysis of type ranges

- Compiler follows range propagation values and assumes worst-case propagation: For example the sum of two 3-bit unsigned integers fits in a 4-bit unsigned integer.

Ada-Style Value Range Propagation

Thanks to built in static analysis of type ranges

- Compiler follows range propagation values and assumes worst-case propagation: For example the sum of two 3-bit unsigned integers fits in a 4-bit unsigned integer.
- Removes need for dumb casts in place such as

```
auto i = 254;
ubyte ub = i+1; // no cast needed
```

Ada-Style Value Range Propagation

Thanks to built in static analysis of type ranges

- Compiler follows range propagation values and assumes worst-case propagation: For example the sum of two 3-bit unsigned integers fits in a 4-bit unsigned integer.
- Removes need for dumb casts in place such as

```
auto i = 254;
ubyte ub = i+1; // no cast needed
```

- and in places such as

```
const i = readInRunTime!ubyte;
const j = readInRunTime!ubyte;
const ij = i*j;
ushort y = ij; // no cast needed
```

Ada-Style Type Strictness

Implementing an Ada-Style Range Type is a no-brainer thanks

- Template Powers

Ada-Style Type Strictness

Implementing an Ada-Style Range Type is a no-brainer thanks

- Template Powers
- More flexible operator overloading

Ada-Style Type Strictness

Implementing an Ada-Style Range Type is a no-brainer thanks

- Template Powers
- More flexible operator overloading
- CTFE

Ada-Style Type Strictness

Implementing an Ada-Style Range Type is a no-brainer thanks

- Template Powers
- More flexible operator overloading
- CTFE
- struct/class data `invariant` code plus

Ada-Style Type Strictness

Implementing an Ada-Style Range Type is a no-brainer thanks

- Template Powers
- More flexible operator overloading
- CTFE
- struct/class data `invariant` code plus
- *Contract*-Based Programming using

```
in {} out (result) {} body {}
```

Provably not Probably Correct

- No more Faith-Based Programming

Provably not Probably Correct

- No more Faith-Based Programming
- C#-Designer: “Billion-Dollar mistake”

Provably not Probably Correct

- No more Faith-Based Programming
- C#-Designer: “Billion-Dollar mistake”
- Safety-Critical: Aviation

Provably not Probably Correct

- No more Faith-Based Programming
- C#-Designer: "Billion-Dollar mistake"
- Safety-Critical: Aviation
- How do we prove that a code never throws an exception when trying to deref a zero (null) reference? Solutions:

Provably not Probably Correct

- No more Faith-Based Programming
- C#-Designer: "Billion-Dollar mistake"
- Safety-Critical: Aviation
- How do we prove that a code never throws an exception when trying to deref a zero (null) reference? Solutions:
  - **Haskell**: `Maybe`

Provably not Probably Correct

- No more Faith-Based Programming
- C#-Designer: "Billion-Dollar mistake"
- Safety-Critical: Aviation
- How do we prove that a code never throws an exception when trying to deref a zero (null) reference? Solutions:
  - **Haskell**: `Maybe`
  - **Ada 2005**: `not null`

Provably not Probably Correct

- No more Faith-Based Programming
- C#-Designer: "Billion-Dollar mistake"
- Safety-Critical: Aviation
- How do we prove that a code never throws an exception when trying to deref a zero (null) reference? Solutions:
  - **Haskell**: `Maybe`
  - **Ada 2005**: `not null`
  - **D**: Library: `NotNull`, Maybe Language: `@nullable` required in `@safe` D code with extra compiler switch

Provably not Probably Correct

- No more Faith-Based Programming
- C#-Designer: "Billion-Dollar mistake"
- Safety-Critical: Aviation
- How do we prove that a code never throws an exception when trying to deref a zero (null) reference? Solutions:
  - **Haskell**: `Maybe`
  - **Ada 2005**: `not null`
  - **D**: Library: `NotNull`, Maybe Language: `@nullable` required in `@safe` D code with extra compiler switch
  - **Java 8**: Type annotation `@NotNull`

Provably not Probably Correct

- No more Faith-Based Programming
- C#-Designer: "Billion-Dollar mistake"
- Safety-Critical: Aviation
- How do we prove that a code never throws an exception when trying to deref a zero (null) reference? Solutions:
  - **Haskell**: `Maybe`
  - **Ada 2005**: `not null`
  - **D**: Library: `NotNull`, Maybe Language: `@nullable` required in `@safe` D code with extra compiler switch
  - **Java 8**: Type annotation `@NotNull`
- Provably Correct:

Provably not Probably Correct

- No more Faith-Based Programming
- C#-Designer: "Billion-Dollar mistake"
- Safety-Critical: Aviation
- How do we prove that a code never throws an exception when trying to deref a zero (null) reference? Solutions:
  - **Haskell**: `Maybe`
  - **Ada 2005**: `not null`
  - **D**: Library: `NotNull`, Maybe Language: `@nullable` required in `@safe` D code with extra compiler switch
  - **Java 8**: Type annotation `@NotNull`
- Provably Correct:
  - Safety: @safe, @trusted, @system

Provably not Probably Correct

- No more Faith-Based Programming
- C#-Designer: "Billion-Dollar mistake"
- Safety-Critical: Aviation
- How do we prove that a code never throws an exception when trying to deref a zero (null) reference? Solutions:
  - **Haskell**: `Maybe`
  - **Ada 2005**: `not null`
  - **D**: Library: `NotNull`, Maybe Language: `@nullable` required in `@safe` D code with extra compiler switch
  - **Java 8**: Type annotation `@NotNull`
- Provably Correct:
  - Safety: @safe, @trusted, @system
  - Data: immutable, const. Axiom: immutable is transitive: The only reasonable way to do it. Glass-Cage/Locker Analogy.

Provably not Probably Correct

- No more Faith-Based Programming
- C#-Designer: "Billion-Dollar mistake"
- Safety-Critical: Aviation
- How do we prove that a code never throws an exception when trying to deref a zero (null) reference? Solutions:
  - **Haskell**: `Maybe`
  - **Ada 2005**: `not null`
  - **D**: Library: `NotNull`, Maybe Language: `@nullable` required in `@safe` D code with extra compiler switch
  - **Java 8**: Type annotation `@NotNull`
- Provably Correct:
  - Safety: @safe, @trusted, @system
  - Data: immutable, const. Axiom: immutable is transitive: The only reasonable way to do it. Glass-Cage/Locker Analogy.
  - Code: pure, in, out, body, invariant, unittest + assert

Transactional Part 1: scope statement

Makes writing a *safe* cp in D a no-brainer:

```
import std.exception, std.file;
void main(string[] args)
{
    enforce(args.length == 3,
            "Usage: trcopy file1 file2");
    auto tmp = args[2] ~ ".messedup";
    scope(failure) {
        if (exists(tmp))
            remove(tmp);
    }
    copy(args[1], tmp);
    rename(tmp, args[2]);
}
```

Transactional Part 2: Exception-Safe Constructors

- D uses Copy-And-Swap Principle

Transactional Part 2: Exception-Safe Constructors

- D uses Copy-And-Swap Principle
- C++ does not => Not Exception-Safe!

Build Tool Builtin

- **`rdmd hello.d`** when `hello.d` contains the main function

Build Tool Builtin

- `rdmd hello.d` when `hello.d` contains the main function
- UNIX "shebang": **"As-a-Script-Execution"** using `#!/usr/bin/env rdmd`

Build Tool Builtin

- `rdmd hello.d` when `hello.d` contains the main function
- UNIX "shebang": **"As-a-Script-Execution"** using `#!/usr/bin/env rdmd`
- For larger projects **SCons** supports D

Compilation Performance Facts

- Context-Free Non-Ambiguous Grammar

Compilation Performance Facts

- Context-Free Non-Ambiguous Grammar
- *Single*-Pass (vs C++ 3-7 Passes) makes builds

Compilation Performance Facts

- Context-Free Non-Ambiguous Grammar
- *Single*-Pass (vs C++ 3-7 Passes) makes builds
- Fastest Compiled Language

Compilation Performance Facts

- Context-Free Non-Ambiguous Grammar
- *Single*-Pass (vs C++ 3-7 Passes) makes builds
- Fastest Compiled Language
- $>$ 10x faster than C++

Compilation Performance Facts

- Context-Free Non-Ambiguous Grammar
- *Single*-Pass (vs C++ 3-7 Passes) makes builds
- Fastest Compiled Language
- $>$ 10x faster than C++
- Phobos builds and runs all its unittests in 2 minutes!

Compilation Performance Facts

- Context-Free Non-Ambiguous Grammar
- *Single*-Pass (vs C++ 3-7 Passes) makes builds
- Fastest Compiled Language
- $>$ 10x faster than C++
- Phobos builds and runs all its unittests in 2 minutes!
- **Quick Iterations**: A Github Pull Request is checked on 10 configurations within half an hour!

Compilation Performance Facts

- Context-Free Non-Ambiguous Grammar
- *Single*-Pass (vs C++ 3-7 Passes) makes builds
- Fastest Compiled Language
- $>$ 10x faster than C++
- Phobos builds and runs all its unittests in 2 minutes!
- **Quick Iterations**: A Github Pull Request is checked on 10 configurations within half an hour!
- Multi-threading Unittesting will give further speedups

Compilation Speed Consequences 1

Enables use of new tools such as

- **dustmite** - Code Minimizer: Automatically isolates bug generating a specific error message by minimizing its parenting source code. Built into recent distribution of DMD.

Compilation Speed Consequences 1

Enables use of new tools such as

- **dustmite** - Code Minimizer: Automatically isolates bug generating a specific error message by minimizing its parenting source code. Built into recent distribution of DMD.
- Instant (1s) Syntactic and Semantic Analys for Project around 20k lines

Compilation Speed Consequences 1

Enables use of new tools such as

- **dustmite** - Code Minimizer: Automatically isolates bug generating a specific error message by minimizing its parenting source code. Built into recent distribution of DMD.
- Instant (1s) Syntactic and Semantic Analys for Project around 20k lines
- **Type-Rich Programming** finally becomes *practically* doable in *real* large projects, not just toy examples (such as in `Boost.Units`). Library error-messages can also be made understandable thanks to *alias*es and string template arguments. Having (physical) **unit analysis** checked by the compiler is another quantum leap code correctness.

Compilation Speed Consequences 2

"As-a-Script-Execution/Evaluation" using

- `#!/usr/bin/env rdmd` removes need for build step and tool for small applications and enables

Compilation Speed Consequences 2

"As-a-Script-Execution/Evaluation" using

- `#!/usr/bin/env rdmd` removes need for build step and tool for small applications and enables
- Module-Executable-Hybrid Behaviour á lá Python, MATLAB, etc.

The Big Picture

- Fastest Parser

The Big Picture

- Fastest Parser
- Templates everywhere

The Big Picture

- Fastest Parser
- Templates everywhere
- **Compiler sees whole program not just single function**

The Big Picture

- Fastest Parser
- Templates everywhere
- **Compiler sees whole program not just single function**
- **Unique Feature**: *Infers* of purity and Safety.

The Big Picture

- Fastest Parser
- Templates everywhere
- **Compiler sees whole program not just single function**
- **Unique Feature**: *Infers* of purity and Safety.
- Better optimizations, safety checks, static analysis, etc

Safe Concurrency by default

- "Globals" Variables **TLS** by default

Safe Concurrency by default

- "Globals" Variables **TLS** by default
- No default race-conditions

Safe Concurrency by default

- "Globals" Variables **TLS** by default
- No default race-conditions
- Fundament: Better

Safe Concurrency by default

- "Globals" Variables **TLS** by default
- No default race-conditions
- Fundament: Better
  - Safe with 3 instructions than

Safe Concurrency by default

- "Globals" Variables **TLS** by default
- No default race-conditions
- Fundament: Better
  - Safe with 3 instructions than
  - Incorrect with 1 instructions

Safe Concurrency by default

- "Globals" Variables **TLS** by default
- No default race-conditions
- Fundament: Better
  - Safe with 3 instructions than
  - Incorrect with 1 instructions
- In PIC-code difference is unnoticeble

Safe Concurrency by default

- "Globals" Variables **TLS** by default
- No default race-conditions
- Fundament: Better
  - Safe with 3 instructions than
  - Incorrect with 1 instructions
- In PIC-code difference is unnoticeble
- Thread-Global must be tagged with `__shared`

Safe Concurrency by default

- "Globals" Variables **TLS** by default
- No default race-conditions
- Fundament: Better
  - Safe with 3 instructions than
  - Incorrect with 1 instructions
- In PIC-code difference is unnoticeble
- Thread-Global must be tagged with `__shared`
- Compare with OS Protected Memory

Safe Concurrency by default

- "Globals" Variables **TLS** by default
- No default race-conditions
- Fundament: Better
  - Safe with 3 instructions than
  - Incorrect with 1 instructions
- In PIC-code difference is unnoticeble
- Thread-Global must be tagged with `__shared`
- Compare with OS Protected Memory
- Threads become *isolated* LWP

Safe Concurrency: Example

```
import std.concurrency: spawn;

alias T = string[];

void useArgs(const T x)
{
    import std.stdio: writeln;
    writeln("x: ", x);
}

void main(T args)
{
    useArgs(args); // ok to call in same thread
    auto f1 = spawn(&useArgs, args.idup); // ok
    auto f3 = spawn(&useArgs, args); // errors as: "Aliases to
        mutable thread-local data not allowed."
}
```

Message Passing

- Unique Feature: Messages are automatically dynamically typed and can be pattern matched on like in dynamic languages! Enabled by close interaction with compiler, type mangling (serialization) etc.

```
import std.concurrency;
void f(Tid tid)
{
    receive(
        (int i) { /* do something with i */ }
    );
    send(tid, true);
}
void main()
{
    auto tid = spawn(&f, thisTid); // async call f
    send(tid, 42);
    const ok = receiveOnly!(bool);
    assert(ok);
}
```

Parallelism

- Fibers and Threads

Parallelism

- Fibers and Threads
- Inside `std.parallelism`

Parallelism

- Fibers and Threads
- Inside `std.parallelism`
- Task-Based Scheduling & Stealing

Parallelism

- Fibers and Threads
- Inside `std.parallelism`
- Task-Based Scheduling & Stealing
- Builtin (no costly 3:rd part such as C++'s Intel TBB)

Parallelism

- Fibers and Threads
- Inside `std.parallelism`
- Task-Based Scheduling & Stealing
- Builtin (no costly 3:rd part such as C++'s Intel TBB)
- Builtin `TaskPool` Singleton

Parallelism

- Fibers and Threads
- Inside `std.parallelism`
- Task-Based Scheduling & Stealing
- Builtin (no costly 3:rd part such as C++'s Intel TBB)
- Builtin `TaskPool` Singleton
- Showcase is vibe.d

Parallelism

- Fibers and Threads
- Inside `std.parallelism`
- Task-Based Scheduling & Stealing
- Builtin (no costly 3:rd part such as C++'s Intel TBB)
- Builtin `TaskPool` Singleton
- Showcase is vibe.d
- Example: Close to optimal speedup:

```
import std.algorithm, std.parallelism, std.range;
void main() {
    immutable n = 1_000_000_000;
    immutable delta = 1.0 / n;
    real getTerm(int i) {
        immutable x = ( i - 0.5 ) * delta;
        return delta / ( 1.0 + x * x ) ;
    }
    immutable pi = 4.0 * taskPool.reduce!"a+b"(
        std.algorithm.map!getTerm(iota(n))
    );
}
```

Parallelism

- Fibers and Threads
- Inside `std.parallelism`
- Task-Based Scheduling & Stealing
- Builtin (no costly 3:rd part such as C++'s Intel TBB)
- Builtin `TaskPool` Singleton
- Showcase is vibe.d
- Example: Close to optimal speedup:
  - `TaskPool.reduce`: 12.170 s

```
import std.algorithm, std.parallelism, std.range;
void main() {
    immutable n = 1_000_000_000;
    immutable delta = 1.0 / n;
    real getTerm(int i) {
        immutable x = ( i - 0.5 ) * delta;
        return delta / ( 1.0 + x * x ) ;
    }
    immutable pi = 4.0 * taskPool.reduce!"a+b"(
        std.algorithm.map!getTerm(iota(n))
    );
}
```

Parallelism

- Fibers and Threads
- Inside `std.parallelism`
- Task-Based Scheduling & Stealing
- Builtin (no costly 3:rd part such as C++'s Intel TBB)
- Builtin `TaskPool` Singleton
- Showcase is vibe.d
- Example: Close to optimal speedup:
  - `TaskPool.reduce`: 12.170 s
  - `std.algorithm.reduce`: 24.065 s

```
import std.algorithm, std.parallelism, std.range;
void main() {
    immutable n = 1_000_000_000;
    immutable delta = 1.0 / n;
    real getTerm(int i) {
        immutable x = ( i - 0.5 ) * delta;
        return delta / ( 1.0 + x * x ) ;
    }
    immutable pi = 4.0 * taskPool.reduce!"a+b"(
        std.algorithm.map!getTerm(iota(n))
    );
}
```

Floating-Point Correctness

- Defaulted To **NaN** (Non-A-Number) making it *obvious* that a value is undefined (zero is not) when code switches maintainer.

Floating-Point Correctness

- Defaulted To **NaN** (Non-A-Number) making it *obvious* that a value is undefined (zero is not) when code switches maintainer.
- NaNs propagate through system without bringing the system down but still indiciates something is wrong.

Strict Control

- Concurrency: **shared**

enables optimization and auto-parallelization

Strict Control

- Concurrency: **shared**
- Ownership, Data-flow Analysis: **immutable**, **const**

enables optimization and auto-parallelization

Strict Control

- Concurrency: **shared**
- Ownership, Data-flow Analysis: **immutable**, **const**
- Exception: **nothrow**

enables optimization and auto-parallelization

Strict Control

- Concurrency: **shared**
- Ownership, Data-flow Analysis: **immutable**, **const**
- Exception: **nothrow**
- Code Access: **pure**

enables optimization and auto-parallelization

Strict Control

- Concurrency: **shared**
- Ownership, Data-flow Analysis: **immutable**, **const**
- Exception: **nothrow**
- Code Access: **pure**
- Memory Safety: **@safe**

enables optimization and auto-parallelization

Strict Control

- Concurrency: **shared**
- Ownership, Data-flow Analysis: **immutable**, **const**
- Exception: **nothrow**
- Code Access: **pure**
- Memory Safety: **@safe**
- GC Heap Allocation: **@nogc** and compiler flag **-vgc**

enables optimization and auto-parallelization

Flexible Memory Management System

- Builtin `class` Garbage Collection (GC) by default or

Flexible Memory Management System

- Builtin `class` Garbage Collection (GC) by default or
- Roll your own class Constructors/Destructors using `new` and `delete`

Flexible Memory Management System

- Builtin `class` Garbage Collection (GC) by default or
- Roll your own class Constructors/Destructors using `new` and `delete`
- `struct` Copy Constructors more Efficient than C++ thanks to **bitblit**.

Flexible Memory Management System

- Builtin `class` Garbage Collection (GC) by default or
- Roll your own class Constructors/Destructors using `new` and `delete`
- `struct` Copy Constructors more Efficient than C++ thanks to **bitblit**.
- Move-Semantics Built-in for `struct` thanks to "postblit"

Flexible Memory Management System

- Builtin `class` Garbage Collection (GC) by default or
- Roll your own class Constructors/Destructors using `new` and `delete`
- `struct` Copy Constructors more Efficient than C++ thanks to **bitblit**.
- Move-Semantics Built-in for `struct` thanks to "postblit"
- Upcoming Custom Allocators in Phobos: `std.allocator`

All Aspect of Source Code Accessible to Developer

Programmer can access everything compiler "sees":

- Static (Compile-Time) Reflection

All Aspect of Source Code Accessible to Developer

Programmer can access everything compiler "sees":

- Static (Compile-Time) Reflection
  - Members of `struct` and `class` via `tupleof` can be iterated with `foreach` makes reflection in for example serialization trivial

All Aspect of Source Code Accessible to Developer

Programmer can access everything compiler "sees":

- Static (Compile-Time) Reflection
  - Members of `struct` and `class` via `tupleof` can be iterated with `foreach` makes reflection in for example serialization trivial
  - Enumerators, along with most other builtins, (de)serialized using

```
enumInstance.to!string
```

and

```
stringInstance.to!EnumName
```

given (`import std.conv: to;`) like in Ada.

All Aspect of Source Code Accessible to Developer

Programmer can access everything compiler "sees":

- Static (Compile-Time) Reflection
  - Members of `struct` and `class` via `tupleof` can be iterated with `foreach` makes reflection in for example serialization trivial
  - Enumerators, along with most other builtins, (de)serialized using

```
enumInstance.to!string
```

and

```
stringInstance.to!EnumName
```

given (`import std.conv: to;`) like in Ada.

  - Type Names using `T.stringof`

All Aspect of Source Code Accessible to Developer

Programmer can access everything compiler "sees":

- Static (Compile-Time) Reflection
  - Members of `struct` and `class` via `tupleof` can be iterated with `foreach` makes reflection in for example serialization trivial
  - Enumerators, along with most other builtins, (de)serialized using

```
enumInstance.to!string
```

    and

```
stringInstance.to!EnumName
```

    given (`import std.conv: to;`) like in Ada.
  - Type Names using `T.stringof`
  - Gives more informative error messages

All Aspect of Source Code Accessible to Developer

Programmer can access everything compiler "sees":

- Static (Compile-Time) Reflection
  - Members of `struct` and `class` via `tupleof` can be iterated with `foreach` makes reflection in for example serialization trivial
  - Enumerators, along with most other builtins, (de)serialized using

```
enumInstance.to!string
```

    and

```
stringInstance.to!EnumName
```

    given (`import std.conv: to;`) like in Ada.
  - Type Names using `T.stringof`
  - Gives more informative error messages
- Propagate Static to Dynamic Reflection

All Aspect of Source Code Accessible to Developer

Programmer can access everything compiler "sees":

- Static (Compile-Time) Reflection
  - Members of `struct` and `class` via `tupleof` can be iterated with `foreach` makes reflection in for example serialization trivial
  - Enumerators, along with most other builtins, (de)serialized using

```
enumInstance.to!string
```

    and

```
stringInstance.to!EnumName
```

    given (`import std.conv: to;`) like in Ada.
  - Type Names using `T.stringof`
  - Gives more informative error messages
- Propagate Static to Dynamic Reflection
  - Auto-Generate

All Aspect of Source Code Accessible to Developer

Programmer can access everything compiler "sees":

- Static (Compile-Time) Reflection
  - Members of `struct` and `class` via `tupleof` can be iterated with `foreach` makes reflection in for example serialization trivial
  - Enumerators, along with most other builtins, (de)serialized using

```
enumInstance.to!string
```

    and

```
stringInstance.to!EnumName
```

    given (`import std.conv: to;`) like in Ada.
  - Type Names using `T.stringof`
  - Gives more informative error messages
- Propagate Static to Dynamic Reflection
  - Auto-Generate
  - Command-Line Interface and

All Aspect of Source Code Accessible to Developer

Programmer can access everything compiler "sees":

- Static (Compile-Time) Reflection
  - Members of `struct` and `class` via `tupleof` can be iterated with `foreach` makes reflection in for example serialization trivial
  - Enumerators, along with most other builtins, (de)serialized using

```
enumInstance.to!string
```

    and

```
stringInstance.to!EnumName
```

    given (`import std.conv: to;`) like in Ada.
  - Type Names using `T.stringof`
  - Gives more informative error messages
- Propagate Static to Dynamic Reflection
  - Auto-Generate
  - Command-Line Interface and
  - GUI-Look-and-Feel from

More Inference than in C++11

Along with type inference through `auto`, also infers

- Accessibility (`auto`, `const` or `immutable`) using **inout** parameter and return qualifier and

More Inference than in C++11

Along with type inference through `auto`, also infers

- Accessibility (`auto`, `const` or `immutable`) using **inout** parameter and return qualifier and
- Return by Value or Reference Semantics using **auto ref**

More Inference than in C++11

Along with type inference through `auto`, also infers

- Accessibility (`auto`, `const` or `immutable`) using **inout** parameter and return qualifier and
- Return by Value or Reference Semantics using **auto ref**
- These thanks to D strict memory model — compiler are aware of what code

More Inference than in C++11

Along with type inference through `auto`, also infers

- Accessibility (`auto`, `const` or `immutable`) using **inout** parameter and return qualifier and
- Return by Value or Reference Semantics using **auto ref**
- These thanks to D strict memory model — compiler are aware of what code
- This reduces code bloat more than *any* other *imperative* (non Haskell-like) language

Expressiveness: Meta-Programming

Compile-Time-Function-Evaluation (CTFE) ,

realized by the Combo: (**`pure`** & **`immutable`**) + **`static if`**, enables

- `Safety`: Detect Errors before deployment

Expressiveness: Meta-Programming

Compile-Time-Function-Evaluation (CTFE) ,

realized by the Combo: (`pure` & `immutable`) + `static if`, enables

- `Safety`: Detect Errors before deployment
- **Declarative Programming** generates D code in same compilation pass as regular D code:

Expressiveness: Meta-Programming

Compile-Time-Function-Evaluation (CTFE) ,

realized by the Combo: (`pure` & `immutable`) + `static if`, enables

- `Safety`: Detect Errors before deployment
- **Declarative Programming** generates D code in same compilation pass as regular D code:
  - `Pegged` on Github replaces external parsing tools

Expressiveness: Meta-Programming

Compile-Time-Function-Evaluation (CTFE) ,

realized by the Combo: (**`pure`** & **`immutable`**) + **`static if`**, enables

- `Safety`: Detect Errors before deployment
- **Declarative Programming** generates D code in same compilation pass as regular D code:
  - `Pegged` on Github replaces external parsing tools
  - `std.regex.ctRegex` fastest in the world!

When is CTFE Triggered?

CTFEable functions (**`pure`** with whole body visible) can be either

- Evaluated at Compile-time:

```
enum e = f();
```

or

When is CTFE Triggered?

CTFEable functions (**`pure`** with whole body visible) can be either

- Evaluated at Compile-time:

```
enum e = f();
```

or

- Executed at Run-time:

```
auto a = f();
const c = f();
immutable i = f();
```

CTFE + Mixin Magic: Pegged

```
import pegged.peg;
import pegged.grammar;
pragma(lib, "pegged");

mixin(grammar(`
Arithmetic:
    Term     < Factor (Add / Sub)*
    Add      < "+" Factor
    Sub      < "-" Factor
    Factor   < Primary (Mul / Div)*
    Mul      < "*" Primary
    Div      < "/" Primary
    Primary  < Parens / Neg / Number / Variable
    Parens   < "(" Term ")"
    Neg      < "-" Primary
    Number   < ~([0-9]+)
    Variable <- identifier`));

void main(string[] args) {
    enum parseTree1 = Arithmetic("1 + 2 - (3*x-5)*6");
    assert(parseTree1.matches == ["1", "+", "2", "-",
           "(", "3", "*", "x", "-", "5", ")", "*", "6"]);
}
```

Concise and Uniform Syntax

```
ubyte someByte = 13;
string someString = "alpha";
auto someNumber = 42;
enum someConstant = 1;
const iPromiseToNeverChangeThisValue = 1;
immutable thisValueCanNeverBeChangedByAnyone = 1;
alias goodName = badName; // potentially many overloads
```

Easy, Flexible & Safe Module System

- As easy as the easiest languages such as Python and Matlab, but typesafe

Easy, Flexible & Safe Module System

- As easy as the easiest languages such as Python and Matlab, but typesafe
- No need for separate `.h` and `.c` to keep in sync by default...but

Easy, Flexible & Safe Module System

- As easy as the easiest languages such as Python and Matlab, but typesafe
- No need for separate `.h` and `.c` to keep in sync by default...but
- **Unique Feature**: Compiler can **auto-generate interface files** if needed!

Easy, Flexible & Safe Module System

- As easy as the easiest languages such as Python and Matlab, but typesafe
- No need for separate `.h` and `.c` to keep in sync by default...but
- **Unique Feature**: Compiler can **auto-generate interface files** if needed!
- Consequence: Many github projects are one D file!

Easy, Flexible & Safe Module System

- **Hijack-Safe**

Easy, Flexible & Safe Module System

- **Hijack-Safe**
- Imports automatically "used" (like C++'s `using namespace`)

Easy, Flexible & Safe Module System

- **Hijack-Safe**
- Imports automatically "used" (like C++'s `using namespace`)
- No *need* to qualify symbols like in Python's `os.path.join()`

Easy, Flexible & Safe Module System

- **Hijack-Safe**
- Imports automatically "used" (like C++'s `using namespace`)
- No *need* to qualify symbols like in Python's `os.path.join()`
- Because of advanced type system overloads very seldom collide

Easy, Flexible & Safe Module System

- **Hijack-Safe**
- Imports automatically "used" (like C++'s `using namespace`)
- No *need* to qualify symbols like in Python's `os.path.join()`
- Because of advanced type system overloads very seldom collide
- This is not possible in dynamic languages like Python and Ruby

Easy, Flexible & Safe Module System

- **Hijack-Safe**
- Imports automatically "used" (like C++'s `using namespace`)
- No *need* to qualify symbols like in Python's `os.path.join()`
- Because of advanced type system overloads very seldom collide
- This is not possible in dynamic languages like Python and Ruby
- When they *do* collide uniquify with scope

Easy, Flexible & Safe Module System

- **Hijack-Safe**
- Imports automatically "used" (like C++'s `using namespace`)
- No *need* to qualify symbols like in Python's `os.path.join()`
- Because of advanced type system overloads very seldom collide
- This is not possible in dynamic languages like Python and Ruby
- When they *do* collide uniquify with scope
- **Scoped** Imports (inside structs, classes, functions, templates, etc)

Easy, Flexible & Safe Module System

- **Hijack-Safe**
- Imports automatically "used" (like C++'s `using namespace`)
- No *need* to qualify symbols like in Python's `os.path.join()`
- Because of advanced type system overloads very seldom collide
- This is not possible in dynamic languages like Python and Ruby
- When they *do* collide uniquify with scope
- **Scoped** Imports (inside structs, classes, functions, templates, etc)
- **Cyclic** Imports (Dependencies) just works, compared to C/C++, Python.

Easy, Flexible & Safe Module System

- **Hijack-Safe**
- Imports automatically "used" (like C++'s `using namespace`)
- No *need* to qualify symbols like in Python's `os.path.join()`
- Because of advanced type system overloads very seldom collide
- This is not possible in dynamic languages like Python and Ruby
- When they *do* collide uniquify with scope
- **Scoped** Imports (inside structs, classes, functions, templates, etc)
- **Cyclic** Imports (Dependencies) just works, compared to C/C++, Python.
- This makes it trivial to break out classes and functions into new separate files.

Safe Integer Literals

- No need to ULL-suffix 64-bit Constants which C++ needs:

```
0x0000_1111_2222_3333ULL
```

Safe Integer Literals

- No need to ULL-suffix 64-bit Constants which C++ needs:

```
0x0000_1111_2222_3333ULL
```

- Static Type Analysis of Constants for Implicit Casts

Safe Integer Literals

- No need to ULL-suffix 64-bit Constants which C++ needs:

```
0x0000_1111_2222_3333ULL
```

- Static Type Analysis of Constants for Implicit Casts
  - Ok:

```
int x = cast(ulong)0x0000_1111;
```

Safe Integer Literals

- No need to ULL-suffix 64-bit Constants which C++ needs:

```
0x0000_1111_2222_3333ULL
```

- Static Type Analysis of Constants for Implicit Casts
  - Ok:

```
int x = cast(ulong)0x0000_1111;
```

  - Error:

```
int y = 0x0000_1111_2222_3333;
```

Safe Integer Literals

- No need to ULL-suffix 64-bit Constants which C++ needs:

```
0x0000_1111_2222_3333ULL
```

- Static Type Analysis of Constants for Implicit Casts
  - Ok:

```
int x = cast(ulong)0x0000_1111;
```

  - Error:

```
int y = 0x0000_1111_2222_3333;
```

  - Ok:

```
int z = cast(int) 0x0000_1111_2222_3333;
```

Expressiveness: Type Construction

Haskell-Like Elegance in Type Construction:

```
alias Pair(T) = Tuple!(T, T);
```

Type-Safe Command Line Argument Parsing

- Type-Aware Builtin (`std.getopt`)

Type-Safe Command Line Argument Parsing

- Type-Aware Builtin (`std.getopt`)
- Just define typed variable and give it as reference into a type aware overloaded `registerArgument(type)`.

Type-Safe Command Line Argument Parsing

- Type-Aware Builtin (`std.getopt`)
- Just define typed variable and give it as reference into a type aware overloaded `registerArgument(type)`.
- Type information infers to interface generation

Type-Safe Command Line Argument Parsing

- Type-Aware Builtin (`std.getopt`)
- Just define typed variable and give it as reference into a type aware overloaded `registerArgument(type)`.
- Type information infers to interface generation
- DRY!

Documentation Generation Builtin

- DDoc Builtin to the compiler

Documentation Generation Builtin

- DDoc Builtin to the compiler
- Runs During Compilation

Documentation Generation Builtin

- DDoc Builtin to the compiler
- Runs During Compilation
- Unnoticable overhead

Documentation Generation Builtin

- DDoc Builtin to the compiler
- Runs During Compilation
- Unnoticable overhead
- Generates HTML, PDF

Documentation Generation Builtin

- DDoc Builtin to the compiler
- Runs During Compilation
- Unnoticable overhead
- Generates HTML, PDF
- D also supported by Doxygen

Code Security/Safety Levels

Classification of code eases bug isolation:

- `@safe` can call `@safe` and in turn

Code Security/Safety Levels

Classification of code eases bug isolation:

- `@safe` can call `@safe` and in turn
- `@trusted` can call `@trusted` and in turn

Code Security/Safety Levels

Classification of code eases bug isolation:

- `@safe` can call `@safe` and in turn
- `@trusted` can call `@trusted` and in turn
- `@system` can call `@system` and C code

Safe Code

`@safe` functions can

- unrefer pointers but

Safe Code

`@safe` functions can

- unrefer pointers but
- *not* null-initialize nor do arithmetic on them

Safe Code

`@safe` functions can

- unrefer pointers but
- *not* null-initialize nor do arithmetic on them
- *not* do any memory recast dribblings

Safe Code

`@safe` functions can

- unrefer pointers but
- *not* null-initialize nor do arithmetic on them
- *not* do any memory recast dribblings
- *not* do unsafe `cast`

Compatibility with C/C++

- ABI Compatible with C, most of C++ and soon Objective-C (Apple IOS) through `extern`

Compatibility with C/C++

- ABI Compatible with C, most of C++ and soon Objective-C (Apple IOS) through `extern`
- No Surprises when porting: C constructs that compiles in D should behave the same as in C

Compatibility with C/C++

- ABI Compatible with C, most of C++ and soon Objective-C (Apple IOS) through `extern`
- No Surprises when porting: C constructs that compiles in D should behave the same as in C
- @safeness of templated functions are automatically deduced by compiler

Commercial Successes

- **Sociomantic**: Profitable D-only company with no initial funding/backing bought for $100M.

Commercial Successes

- **Sociomantic**: Profitable D-only company with no initial funding/backing bought for $100M.
- **Remedy Games Engine**: replace scripting languages

Commercial Successes

- **Sociomantic**: Profitable D-only company with no initial funding/backing bought for $100M.
- **Remedy Games Engine**: replace scripting languages
- **Facebook**!: Andrei, Proof-of-Concept, Bounties

Commercial Successes

- **Sociomantic**: Profitable D-only company with no initial funding/backing bought for $100M.
- **Remedy Games Engine**: replace scripting languages
- **Facebook**!: Andrei, Proof-of-Concept, Bounties
- **EMSI**: Economic Modeling Specialists Intl

Commercial Successes

- **Sociomantic**: Profitable D-only company with no initial funding/backing bought for $100M.
- **Remedy Games Engine**: replace scripting languages
- **Facebook**!: Andrei, Proof-of-Concept, Bounties
- **EMSI**: Economic Modeling Specialists Intl
- **STACK4**: Gameserver Backend written in D 2

Commercial Successes

- **Sociomantic**: Profitable D-only company with no initial funding/backing bought for $100M.
- **Remedy Games Engine**: replace scripting languages
- **Facebook**!: Andrei, Proof-of-Concept, Bounties
- **EMSI**: Economic Modeling Specialists Intl
- **STACK4**: Gameserver Backend written in D 2
- **SR Labs**: Eye Tracking Systems Integrator

Commercial Successes

- **Sociomantic**: Profitable D-only company with no initial funding/backing bought for $100M.
- **Remedy Games Engine**: replace scripting languages
- **Facebook**!: Andrei, Proof-of-Concept, Bounties
- **EMSI**: Economic Modeling Specialists Intl
- **STACK4**: Gameserver Backend written in D 2
- **SR Labs**: Eye Tracking Systems Integrator
- **Funatics Software GmbH**: MMO Game Developer

Commercial Successes

- **Sociomantic**: Profitable D-only company with no initial funding/backing bought for $100M.
- **Remedy Games Engine**: replace scripting languages
- **Facebook**!: Andrei, Proof-of-Concept, Bounties
- **EMSI**: Economic Modeling Specialists Intl
- **STACK4**: Gameserver Backend written in D 2
- **SR Labs**: Eye Tracking Systems Integrator
- **Funatics Software GmbH**: MMO Game Developer
- **RandomStorm**: network security products and services

Other Improvements

- **Multi-Dim Slicing** a la Matlab och Fortran 90+: `x[0..m, 0..n]`. Just merged into DMD 2.066 (git master).

Other Improvements

- **Multi-Dim Slicing** a la Matlab och Fortran 90+: `x[0..m, 0..n]`. Just merged into DMD 2.066 (git master).
- Compacter generics using `static if`

Other Improvements

- **Multi-Dim Slicing** a la Matlab och Fortran 90+: `x[0..m, 0..n]`. Just merged into DMD 2.066 (git master).
- Compacter generics using `static if`
- No darn semicolons after classes/structs

Other Improvements

- **Multi-Dim Slicing** a la Matlab och Fortran 90+: `x[0..m, 0..n]`. Just merged into DMD 2.066 (git master).
- Compacter generics using `static if`
- No darn semicolons after classes/structs
- **Nested functions**: Capture scope of parent

Compilers

- **DMD**: Reference (Recent Features), Fast Compile

Compilers

- **DMD**: Reference (Recent Features), Fast Compile
- **GDC**: GCC, Fast Code

Compilers

- **DMD**: Reference (Recent Features), Fast Compile
- **GDC**: GCC, Fast Code
- **LDC**: LLVM (ca 1 month behing DMD), Fast Code

Compilers

- **DMD**: Reference (Recent Features), Fast Compile
- **GDC**: GCC, Fast Code
- **LDC**: LLVM (ca 1 month behing DMD), Fast Code
- **DDMD**: coming “soon”.

IDEs

- **Visual D**: (Visual Studio):

IDEs

- **Visual D**: (Visual Studio):
- **Eclipse DDT**:

IDEs

- **Visual D**: (Visual Studio):
- **Eclipse DDT**:
- **Emacs FlyCheck**: Including visual feedback overlays of unittests!

IDEs

- **Visual D**: (Visual Studio):
- **Eclipse DDT**:
- **Emacs FlyCheck**: Including visual feedback overlays of unittests!
- **DCD**: D Completion Daemon: Server-Client! Frontends for Emacs, Vi, Sublime Text, etc.

Learning Curve & Cons

- When to use `x[]`?

Learning Curve & Cons

- When to use `x[]`?
- Compiler Messages are better than C++ but not yet perfect

Learning Curve & Cons

- When to use `x[]`?
- Compiler Messages are better than C++ but not yet perfect
- Template Instance Syntax: `Tuple!(S,T,U)`

Learning Curve & Cons

- When to use `x[]`?
- Compiler Messages are better than C++ but not yet perfect
- Template Instance Syntax: `Tuple!(S,T,U)`
- Compile-Time Constants (like C's `define`):

```
enum thisConstantCanNeverChange = 42;
```

Learning Curve & Cons

- When to use `x[]`?
- Compiler Messages are better than C++ but not yet perfect
- Template Instance Syntax: `Tuple!(S,T,U)`
- Compile-Time Constants (like C's `define`):

```
enum thisConstantCanNeverChange = 42;
```

- CTFE is slow for complex usage such as **Pegged**.

Trend: Safe Memory-Models

- Not by default in C/C++,

Trend: Safe Memory-Models

- Not by default in C/C++,
- and compilers optimize away it by default, because

Trend: Safe Memory-Models

- Not by default in C/C++,
- and compilers optimize away it by default, because
- C,C++ is not built to provide ownership control and safe memory slicing, so range checking gives a significant performance hit

Trend: Safe Memory-Models

- Not by default in C/C++,
- and compilers optimize away it by default, because
- C,C++ is not built to provide ownership control and safe memory slicing, so range checking gives a significant performance hit
- Firefox & Rust: 95 % of browser security holes are buffer overflows

Trend: Safe Memory-Models

- Not by default in C/C++,
- and compilers optimize away it by default, because
- C,C++ is not built to provide ownership control and safe memory slicing, so range checking gives a significant performance hit
- Firefox & Rust: 95 % of browser security holes are buffer overflows
- D has thought of this—DMD's `-release` builds keeps range checking

Trend: Safe Memory-Models

- Not by default in C/C++,
- and compilers optimize away it by default, because
- C,C++ is not built to provide ownership control and safe memory slicing, so range checking gives a significant performance hit
- Firefox & Rust: 95 % of browser security holes are buffer overflows
- D has thought of this—DMD's `-release` builds keeps range checking
- Further, DMD is preparing for *Data-flow and Escape Analysis*

Trend: Safe Memory-Models

- Not by default in C/C++,
- and compilers optimize away it by default, because
- C,C++ is not built to provide ownership control and safe memory slicing, so range checking gives a significant performance hit
- Firefox & Rust: 95 % of browser security holes are buffer overflows
- D has thought of this—DMD's `-release` builds keeps range checking
- Further, DMD is preparing for *Data-flow and Escape Analysis*
- Inhibit slice checking in Divide-&-Conquer Algorithms.

Trend: Safe Memory-Models

- Not by default in C/C++,
- and compilers optimize away it by default, because
- C,C++ is not built to provide ownership control and safe memory slicing, so range checking gives a significant performance hit
- Firefox & Rust: 95 % of browser security holes are buffer overflows
- D has thought of this—DMD's `-release` builds keeps range checking
- Further, DMD is preparing for *Data-flow and Escape Analysis*
- Inhibit slice checking in Divide-&-Conquer Algorithms.
- Strongly-Typed "Nullness" á lá Haskell's `Maybe`: `NotNull` or `@nullable`: No more "null-pointer-exceptions".

What's Next?

- **Allocators**: `std.allocator`

What's Next?

- **Allocators**: `std.allocator`
- **Reference Counted String**: `rcstring`

What's Next?

- **Allocators**: `std.allocator`
- **Reference Counted String**: `rcstring`
- **D Parsing**: `std.lexer`

Conclusion: Component Programming

Features come together in an elegant and concise harmonyReusable Software with native performance!

Conclusion: Productivity and Performance

- *Fastest* Compilation of *Fast* Code gives

Conclusion: Productivity and Performance

- *Fastest* Compilation of *Fast* Code gives
- New level of productivity

Conclusion: Productivity and Performance

- *Fastest* Compilation of *Fast* Code gives
- New level of productivity
- And statically Checked gives

Conclusion: Productivity and Performance

- *Fastest* Compilation of *Fast* Code gives
- New level of productivity
- And statically Checked gives
- New level of correctness an in turn safety

Conclusion: Productivity and Performance

- *Fastest* Compilation of *Fast* Code gives
- New level of productivity
- And statically Checked gives
- New level of correctness an in turn safety
- This is a Game-Changer

Conclusion: Productivity and Performance

- *Fastest* Compilation of *Fast* Code gives
- New level of productivity
- And statically Checked gives
- New level of correctness an in turn safety
- This is a Game-Changer
- Flexibility: Fulfils more needs than any other language

Final Words

- Computing: Extension of the Intellect

Final Words

- Computing: Extension of the Intellect
- It's human to err so...

Final Words

- Computing: Extension of the Intellect
- It's human to err so...
- Let Compilers help correct us and

Final Words

- Computing: Extension of the Intellect
- It's human to err so...
- Let Compilers help correct us and
- Ask not..
  what D can do for you but...
  what you can do for D!

The End

Let's make the best candidate win!