Helpful D Techniques

Ali Çehreli

DConf Online 2020

November 21

The speaker

With D since 2009

- Love at first sight: Created a Turkish D site[1], translated Andrei Alexandrescu's "The Case for D"[2] article to Turkish[3]

[1] http://ddili.org
[2] https://www.drdobbs.com/parallel/the-case-for-d/217801225
[3] http://ddili.org/makale/neden_d.html

The speaker

With D since 2009

- Love at first sight: Created a Turkish D site[1], translated Andrei Alexandrescu's "The Case for D"[2] article to Turkish[3]
- Known for the free book "Programming in D"[4]
  - "A happy accident"[5]
  - Recently available on Educative.io as an *interactive course*:
    - First part[6]
    - Second part[7]

[1] http://ddili.org
[2] https://www.drdobbs.com/parallel/the-case-for-d/217801225
[3] http://ddili.org/makale/neden_d.html
[4] http://ddili.org/ders/d.en/index.html
[5] https://dlang.org/blog/2016/06/29/programming-in-d-a-happy-accident/
[6] https://www.educative.io/courses/programming-in-d-ultimate-guide
[7] https://www.educative.io/collection/10370001/5620751206973440

The speaker (continued)

Currently at Mercedes-Benz Research and Development, North America

- Using D for ROS Bag File Manipulation for Autonomous Driving[1]

[1] https://dconf.org/2019/talks/cehreli.html

The speaker (continued)

Currently at Mercedes-Benz Research and Development, North America

- Using D for ROS Bag File Manipulation for Autonomous Driving[1]
- A project by Daimler and Bosch, a "happy place"

[1] https://dconf.org/2019/talks/cehreli.html

Contents

- Introduction
- Engineering with D
- Mini experience report since DConf 2019
- Various productive features of D
    - Parallelism
    - Concurrency
    - More ...

Contents

- Introduction
- Engineering with D
- Mini experience report since DConf 2019
- Various productive features of D
  - Parallelism
  - Concurrency
  - More ...

```
  Clicks,
not slides
    ▼
    ▼
    ▼
```

Engineering with D

- Involves few bugs
- Is very productive
- Is a lot of fun

Engineering with D

- Involves few bugs
- Is very productive
- Is a lot of fun

Subjectively, D makes a better engineer:

- Less perfectionist

Engineering with D

- Involves few bugs
- Is very productive
- Is a lot of fun

Subjectively, D makes a better engineer:

- Less perfectionist
- More pragmatic

Engineering with D

- Involves few bugs
- Is very productive
- Is a lot of fun

Subjectively, D makes a better engineer:

- Less perfectionist
- More pragmatic
- Acknowledges organic growth (e.g. `@nogc` vs. `pure` is just fine)

Engineering with D

- Involves few bugs
- Is very productive
- Is a lot of fun

Subjectively, D makes a better engineer:

- Less perfectionist
- More pragmatic
- Acknowledges organic growth (e.g. `@nogc` vs. `pure` is just fine)
- Can afford to be less principled because
  - D is both a prototype language and a production language
  - D provides plasticity

See: Presentations by Liran Zvibel[1] and Laeeth Isharc[2]

[1] http://dconf.org/2018/talks/zvibel.html
[2] http://dconf.org/2019/talks/isharc.html

`unittest` pragmatism

One of the most useful features of D, ingrained in D coding:

```d
int halved(int value) {
  return value / 2;
}

unittest {
  assert(42.halved == 21);
}
```

Note: Thanks to UFCS (universal function call syntax) ***`42.halved`*** *is the same as* ***`halved(42)`****.*

unittest pragmatism

One of the most useful features of D, ingrained in D coding:

```d
int halved(int value) {
  return value / 2;
}

unittest {
  assert(42.halved == 21);
}
```

Note: Thanks to UFCS (universal function call syntax) ***`42.halved`*** *is the same as* ***`halved(42)`***.

D's **`unittest`** blocks are as underpowered as it gets:

- No test suites, fixtures, mocks, fakes, etc.
- Nothing but **`assert`** (and **`assertThrown`** and friends)

unittest pragmatism

One of the most useful features of D, ingrained in D coding:

```
int halved(int value) {
  return value / 2;
}

unittest {
  assert(42.halved == 21);
}
```

Note: Thanks to UFCS (universal function call syntax) ***`42.halved`*** *is the same as* ***`halved(42)`****.*

D's **`unittest`** blocks are as underpowered as it gets:

- No test suites, fixtures, mocks, fakes, etc.
- Nothing but **`assert`** (and **`assertThrown`** and friends)

It would be a huge loss if **`unittest`** disappeared.

static if pragmatism

A very useful feature that has been needed[1] for years:

```
struct S(T) {
  static if (isIntegral!T) {
    int i;    // Injected member
              // (good kind of magic, almost cheating)
    // ...

  } else {
    // ...
  }
}
```

[1] https://en.wikipedia.org/wiki/Modern_C%2B%2B_Design

static if pragmatism

A very useful feature that has been needed[1] for years:

```
struct S(T) {
  static if (isIntegral!T) {
    int i;    // Injected member
              // (good kind of magic, almost cheating)
    // ...

  } else {
    // ...
  }
}
```

"Considered"[2] to be "harder to read and understand", "provides ample opportunities for confution[sic] and mistakes", etc.

[1] https://en.wikipedia.org/wiki/Modern_C%2B%2B_Design

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

static if pragmatism

A very useful feature that has been needed[1] for years:

```
struct S(T) {
  static if (isIntegral!T) {
    int i;    // Injected member
              // (good kind of magic, almost cheating)
    // ...

  } else {
    // ...
  }
}
```

"Considered"[2] to be "harder to read and understand", "provides ample opportunities for confution[sic] and mistakes", etc.

It would be a huge loss if **`static if`** disappeared.

[1] https://en.wikipedia.org/wiki/Modern_C%2B%2B_Design

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

Range pragmatism

Ranges are very useful and ubiquitous in modern D code:

- Lazy evaluations
- Minimal memory usage
- Component programming
- Pipeline programming
- Reduced loops
- etc.

```
/*     A range algorithm that generates 0, 1, 2, 3, and 4
       |
       |       A range algorithm that applies 'foo(i)' to each of those
       |       |
       ▼       ▼
*/
     5.iota.map!foo  // ← A range
```

Range pragmatism (continued)

There are issues:

- Some algorithms like **`find()`** naturally return one thing not a range of elements.

Range pragmatism (continued)

There are issues:

- Some algorithms like **`find()`** naturally return one thing not a range of elements.
- If some algorithms like **`partition()`** returned an iterator, the programmers could make a range from the left-hand side or the right-hand side. *(See **`std.algorithm.partition3`**.)*

Range pragmatism (continued)

There are issues:

- Some algorithms like `find()` naturally return one thing not a range of elements.
- If some algorithms like `partition()` returned an iterator, the programmers could make a range from the left-hand side or the right-hand side. *(See `std.algorithm.partition3`.)*

It would be a huge loss if ranges disappeared.

Statistics

D code at Mercedes-Benz Research and Development, North America:

Structure	**2019**	**2020**
programs	3	7
files	25	50
lines	4600	12000

Statistics

D code at Mercedes-Benz Research and Development, North America:

Structure	2019	2020
programs	3	7
files	25	50
lines	4600	12000

1 of every 25 lines contains a format string literal.

Function	2019	2020
format	182	381
writefln	54	134
writef	5	13

Format string literals

Should be considered to be a "bug":

```
format("Hello, %s. Today is %s.", name, day)
```

Format string literals

Should be considered to be a "bug":

```
format("Hello, %s. Today is %s.", name, day)
```

Indispensably useful but "not good enough":

```
format!"Hello, %s. Today is %s."(name, day)
```

Format string literals

Should be considered to be a "bug":

```
format("Hello, %s. Today is %s.", name, day)
```

Indispensably useful but "not good enough":

```
format!"Hello, %s. Today is %s."(name, day)
```

Enter DIP 1036 -- Formatted String Tuple Literals[1]:

Thank you, **Adam D. Ruppe** and **Steven Schveighoffer**. (Based on past work by Walter Bright, Jason Helson, Jonathon Marler, and others.)

```
format!(i"Hello, $name. Today is $day.")
```

[1] https://github.com/dlang/DIPs/blob/master/DIPs/DIP1036.md

Garbage collector statistics

Run your program with a special command line option:

```
$ my_program "--DRT-gcopt=profile:1" <arguments to my_program>
```

See: Garbage Collection specification[1] for other D runtime command line options.

[1] https://dlang.org/spec/garbage.html

Garbage collector statistics

Run your program with a special command line option:

```
$ my_program "--DRT-gcopt=profile:1" <arguments to my_program>
```

See: Garbage Collection specification[1] for other D runtime command line options.

Program	real time	user time	GC memory	GC count	GC time	GC pause total	GC pause max
prg1 using 60G file	3m40s	15m35s	4G	220	3.7s	3.4s	54ms
prg1 using 0.3G file	0m2s	0m6s	0.8G	12	0.2s	0.2s	50ms
prg2 using 2 x 60G files	1m7s	0m46s	19G	64	0.3s	0.2s	13ms
prg2 using 2 x 0.044G files	0m1s	0m1s	6G	6	0.016s	0.015s	4ms

[1] https://dlang.org/spec/garbage.html

Profiling memory allocations

Compile your program with the **`--profile=gc`** switch:

```
$ dmd --profile=gc my_program.d
```

Profiling memory allocations

Compile your program with the **`--profile=gc`** switch:

```
$ dmd --profile=gc my_program.d
```

```
$ ./my_program

$ cat profilegc.log
bytes allocated, allocations, type, function, file:line
    704     4        core.thread.osthread.Thread std.concurrency._spawn!()void [...]
    704     4        int[] my_program.main.__lambda1 my_program.d:23
    704     4        std.concurrency.MessageBox std.concurrency._spawn!()void [...]
    384     4        std.concurrency.LinkTerminated std.concurrency.MessageBox [...]
    256     4        closure std.concurrency._spawn!()void function()int, shared [...]
     16     1        closure D main my_program.d:19
```

Reducing memory allocations

Remove premature pessimization:

```
int[] outer;

while (a) {
  int[] inner;

  while (b) {
    inner ~= e;         // Line 8
  }

  outer ~= bar(inner);  // Line 11
}
```

Reducing memory allocations

Remove premature pessimization:

```
int[] outer;

while (a) {
  int[] inner;

  while (b) {
    inner ~= e;         // Line 8
  }

  outer ~= bar(inner);  // Line 11
}
```

```
bytes allocated, allocations, type, function, file:line
      18000000                259000        int[] deneme.foo deneme.d:8
      11040000                 15000        int[] deneme.foo deneme.d:11
```

Reducing memory allocations (continued)

Reuse the same array for all loop iterations:

```
int[] outer;
int[] inner;

while (a) {
  inner.length = 0;        // Treat as empty
  inner.assumeSafeAppend;  // Reuse existing memory
  // (DON'T DO THOSE. FOR DEMONSTRATION PURPOSES ONLY.)

  while (b) {
    inner ~= e;         // Line 10
  }

  outer ~= bar(inner);  // Line 13
}
```

Reducing memory allocations (continued)

Reuse the same array for all loop iterations:

```
int[] outer;
int[] inner;

while (a) {
  inner.length = 0;        // Treat as empty
  inner.assumeSafeAppend;  // Reuse existing memory
  // (DON'T DO THOSE. FOR DEMONSTRATION PURPOSES ONLY.)

  while (b) {
    inner ~= e;         // Line 10
  }

  outer ~= bar(inner);  // Line 13
}
```

```
bytes allocated, allocations, type, function, file:line
       11040000                  15000 int[] deneme.foo deneme.d:13
         816000  ← was 18M       8000 int[] deneme.foo deneme.d:10
```

Reducing memory allocations (continued)

Use **`static Appender`**:

```
// Remember: These are thread-local
static Appender!(int[]) outer;
static Appender!(int[]) inner;

outer.clear();                  // Clear state from last call

while (a) {
  inner.clear();                // Clear state from last iteration

  while (b) {
    inner ~= e;
  }

  outer ~= bar(inner.data);
}
```

***Warning**: : Thread-safe but non-reentrant.*

Reducing memory allocations (continued)

Use **`static Appender`**:

```
// Remember: These are thread-local
static Appender!(int[]) outer;
static Appender!(int[]) inner;

outer.clear();                  // Clear state from last call

while (a) {
  inner.clear();                // Clear state from last iteration

  while (b) {
    inner ~= e;
  }

  outer ~= bar(inner.data);
}
```

Warning: *: Thread-safe but non-reentrant.*

```
bytes allocated, allocations, type, function, file:line
             64                     2        std.array.Appender!(int[]) [...]
```

Various Productive D Features

Range format specifiers

(Also known as *compound* format specifier and *grouping* format specifier.)

```
5.iota.writefln!"%(%s%)";      // prints 01234
```

Range format specifiers

(Also known as *compound* format specifier and *grouping* format specifier.)

```
5.iota.writefln!"%(%s%)";      // prints 01234
```

- **%(** Opening specifier

Range format specifiers

(Also known as *compound* format specifier and *grouping* format specifier.)

```
5.iota.writefln!"%(%s%)";       // prints 01234
```

- **%(** Opening specifier
- **%)** Closing specifier

Range format specifiers

(Also known as *compound* format specifier and *grouping* format specifier.)

```
5.iota.writefln!"%(%s%)";      // prints 01234
```

- **%(** Opening specifier
- **%)** Closing specifier
- Anything in between is *per element* (e.g. **%s** above)

Range format specifiers

(Also known as *compound* format specifier and *grouping* format specifier.)

```
5.iota.writefln!"%(%s%)";      // prints 01234
```

- **%(** Opening specifier
- **%)** Closing specifier
- Anything in between is *per element* (e.g. **%s** above)

Anything "after the element specifier" is element separator:

```
5.iota.writefln!"%(%s, %)";    // 0, 1, 2, 3, 4
                                                  good: not printed here
```

Range format specifiers (continued)

Too much can be missing:

```
5.iota.writefln!"%(<%s>\n%)";
```

Range format specifiers (continued)

Too much can be missing:

```
5.iota.writefln!"%(<%s>\n%)";
```

```
<0>
<1>
<2>
<3>
<4          '>' is not printed
```

Range format specifiers (continued)

Too much can be missing:

```
5.iota.writefln!"%(<%s>\n%)";
```

```
<0>
<1>
<2>
<3>
<4          '>' is not printed
```

%| specifies where the actual separator starts:

```
5.iota.writefln!"%(<%s>%|\n%)";
```

Range format specifiers (continued)

Too much can be missing:

```
5.iota.writefln!"%(<%s>\n%)";
```

```
<0>
<1>
<2>
<3>
<4          '>' is not printed
```

%| specifies where the actual separator starts:

```
5.iota.writefln!"%(<%s>%|\n%)";
```

```
<0>
<1>
<2>
<3>
<4>         '>' is now a part of all elements
```

Range format specifiers (continued)

Strings are double-quoted (and characters are single-quoted) by default:

```
["monday", "tuesday"].writefln!"%(%s, %)";  // "monday", "tuesday"
```

Range format specifiers (continued)

Strings are double-quoted (and characters are single-quoted) by default:

```
["monday", "tuesday"].writefln!"%(%s, %)";  // "monday", "tuesday"
```

If not desired, open with **%-(**:

```
["monday", "tuesday"].writefln!"%-(%s, %)";  // monday, tuesday
```

Range format specifiers (continued)

Can be nested:

```
5.iota.map!(i => i.iota).writefln!"%(%(%s, %)\n%)";
```

Range format specifiers (continued)

Can be nested:

```
5.iota.map!(i => i.iota).writefln!"%(%(%s, %)\n%)";
```

```
    ← (The range for outer 0 is empty.)
0
0, 1
0, 1, 2
0, 1, 2, 3
```

Range format specifiers (continued)

Can be nested:

```
5.iota.map!(i => i.iota).writefln!"%(%(%s, %)\n%)";
```

```
        ← (The range for outer 0 is empty.)
0
0, 1
0, 1, 2
0, 1, 2, 3
```

For associative arrays, the first specifier is for the key and the second specifier is for the value.

```
auto aa = [ "a" : "one", "b" : "two" ];
aa.writefln!"%-(%s is %s\n%)";
```

```
b is two
a is one
```

Decimal place separator

`%,` is for decimal place separator:

- 3 decimal places by default
- Comma by default

```
writefln!"%,s"(123456789);            // 123,456,789
```

Decimal place separator

%, is for decimal place separator:

- 3 decimal places by default
- Comma by default

```
writefln!"%,s"(123456789);              // 123,456,789
```

```
writefln!"%,*s"(6, 123456789);          // 123,456789
```

Decimal place separator

%, is for decimal place separator:

- 3 decimal places by default
- Comma by default

```
writefln!"%,s"(123456789);            // 123,456,789
```

```
writefln!"%,*s"(6, 123456789);        // 123,456789
```

```
writefln!"%,?s"('·', 123456789);      // 123·456·789
```

Decimal place separator

%, is for decimal place separator:

- 3 decimal places by default
- Comma by default

```
writefln!"%,s"(123456789);          // 123,456,789
```

```
writefln!"%,*s"(6, 123456789);      // 123,456789
```

```
writefln!"%,?s"('·', 123456789);    // 123·456·789
```

```
writefln!"%,*?s"(2, '`', 123456789); // 1`23`45`67`89
```

`std.parallelism.parallel`

One of the most impressive parts of the D standard library.

std.parallelism.parallel

One of the most impressive parts of the D standard library.

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

```
foreach (e; elements) {
  // ...
}
```

`std.parallelism.parallel`

One of the most impressive parts of the D standard library.

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

```
foreach (e; elements) {
  // ...
}
```

The following takes 1 second on 4 cores:

```
foreach (e; elements.parallel) {
  // ...
}
```

`std.parallelism.parallel` (continued)

Impressive because `parallel` is not a language feature:

`std.parallelism.parallel` (continued)

Impressive because `parallel` is not a language feature:

- A function that returns an object,

`std.parallelism.parallel` (continued)

Impressive because **`parallel`** is not a language feature:

- A function that returns an object,
- which defines **`opApply`** to support **`foreach`** iteration,

`std.parallelism.parallel` (continued)

Impressive because `parallel` is not a language feature:

- A function that returns an object,
- which defines `opApply` to support `foreach` iteration,
- which distributes the loop body to a thread pool,

`std.parallelism.parallel` (continued)

Impressive because **`parallel`** is not a language feature:

- A function that returns an object,
- which defines **`opApply`** to support **`foreach`** iteration,
- which distributes the loop body to a thread pool,
- and waits for their completion.

`std.parallelism.parallel` (continued)

Impressive because `parallel` is not a language feature:

- A function that returns an object,
- which defines `opApply` to support `foreach` iteration,
- which distributes the loop body to a thread pool,
- and waits for their completion.

Impressive also because the guideline list is short:

1. Make sure loop body is independent for each element.

std.parallelism.parallel (continued)

```
int[] results;

foreach (e; elements.parallel) {
  results ~= process(e);        // ← BUG
  reportProgress(/* ... */);    // ← Questionable
}
```

std.parallelism.parallel (continued)

```
int[] results;

foreach (e; elements.parallel) {
  results ~= process(e);         // ← BUG
  reportProgress(/* ... */);     // ← Questionable
}
```

One way of fixing the bug:

```
auto results = new int[elements.length];  // Separate result per element

foreach (i, e; elements.parallel) {
  results[i] = process(e);
  // ...
}
```

`std.parallelism.parallel` (continued)

```
int[] results;

foreach (e; elements.parallel) {
  results ~= process(e);        // ← BUG
  reportProgress(/* ... */);    // ← Questionable
}
```

One way of fixing the bug:

```
auto results = new int[elements.length];  // Separate result per element

foreach (i, e; elements.parallel) {
  results[i] = process(e);
  // ...
}
```

Warning*: See "false sharing", which may hurt performance here.*

`std.parallelism.parallel` (continued)

One way of reporting progress correctly:

```
size_t completed = 0;

foreach (i, e; elements.parallel) {
  // ...
  synchronized {    // ← QUESTIONABLE
    completed++;
    reportProgress(completed, elements.length);
  }
}
```

`std.parallelism.parallel` (continued)

One way of reporting progress correctly:

```
size_t completed = 0;

foreach (i, e; elements.parallel) {
  // ...
  synchronized {    // ← QUESTIONABLE
    completed++;
    reportProgress(completed, elements.length);
  }
}
```

Perhaps, needing **`reportProgress()`** is proof that **`process(e)`** takes a long time anyway and **`synchronized`** is affordable? Only you can decide...

`std.parallelism.parallel` (continued)

Two configuration points:

1. *Thread count*: **`parallel`** distributes to **`totalCPUs`** number of threads by default. To change:
    - Create a **`TaskPool`** with desired thread count, which you must **`finish()`**.

`std.parallelism.parallel` (continued)

Two configuration points:

1. *Thread count*: **`parallel`** distributes to **`totalCPUs`** number of threads by default. To change:
   - Create a **`TaskPool`** with desired thread count, which you must **`finish()`**.
2. *Work unit size*: Each thread grabs execution of 100 elements by default. To change:
   - Specify a work unit size (e.g. **`1`** for loop bodies that take a long time).

std.parallelism.parallel (continued)

Two configuration points:

1. *Thread count*: **parallel** distributes to **totalCPUs** number of threads by default. To change:
    - Create a **TaskPool** with desired thread count, which you must **finish()**.
2. *Work unit size*: Each thread grabs execution of 100 elements by default. To change:
    - Specify a work unit size (e.g. **1** for loop bodies that take a long time).

```
auto tp = new TaskPool(totalCPUs / 2);     // 1. Thread count

foreach (e; tp.parallel(elements, 1)) {    // 2. Work unit size
  // ...
}

tp.finish();                               // Don't forget
```

Experiment with different combinations for best performance for your loop.

`std.concurrency`

Message passing concurrency is

- The right kind of concurrency for many programs
- More complicated than parallelism

My recipe follows...

`std.concurrency`

Message passing concurrency is

- The right kind of concurrency for many programs
- More complicated than parallelism

My recipe follows...

Start a thread with **`spawnLinked`**:

```
  auto workers = 4.iota
                 .map!(i => spawnLinked(&workerThread))
                 .array;

// ...

void workerThread() {
  // ...
}
```

std.concurrency

Message passing concurrency is

- The right kind of concurrency for many programs
- More complicated than parallelism

My recipe follows...

Start a thread with **spawnLinked**:

```
  auto workers = 4.iota
                 .map!(i => spawnLinked(&workerThread))
                 .array;

// ...

void workerThread() {
  // ...
}
```

- Send messages with **send**
- Wait for messages with **receive** (or **receiveTimeout**)

`std.concurrency` (continued)

Detect thread termination with a **`LinkTerminated`** message:

```
  size_t completed = 0;

  while (completed < workers.length) {
    receive(
      (const(LinkTerminated) msg) {
        completed++;
      },
      // ...
    );
  }
}
```

*Note: There is also **`OwnerTerminated`**.*

`std.concurrency` (continued)

Threads have separate function call stacks[1].

- Each worker must **`catch`** and communicate its exceptions.

[1] http://dconf.org/2016/talks/cehreli.html

std.concurrency (continued)

Threads have separate function call stacks[1].

- Each worker must **catch** and communicate its exceptions.

```
void workerThread() {
  try {
    workerThreadImpl();  // Dispatch to the implementation
  }
  catch /* ... */
}

void workerThreadImpl() {
  // ...
}
```

[1] http://dconf.org/2016/talks/cehreli.html

Exception kinds

```
         Throwable (do not catch)
         ↗       ↖
 Exception       Error (do not catch)
  ↗    ↖         ↗    ↖
...    ...     ...    ...
```

Exception kinds

```
         Throwable (do not catch)
         ↗       ↖
 Exception       Error (do not catch)
  ↗    ↖         ↗    ↖
...    ...     ...    ...
```

`Exception`: Something bad happened but the program is in a recoverable state.

```
enforce(!name.empty, "Name cannot be empty.");
```

- May **`catch`** and continue

std.concurrency (continued)

Reporting **Exception**:

```
struct WorkerError {
  int id;
  immutable(Exception) exc;
}
```

std.concurrency (continued)

Reporting **Exception**:

```
struct WorkerError {
  int id;
  immutable(Exception) exc;
}
```

```
void workerThread() {
  try /* ... */

  catch (Exception exc) {
    ownerTid.send(WorkerError(id, cast(immutable)exc));
  }
// ...
}
```

std.concurrency (continued)

Reporting **Exception**:

```
struct WorkerError {
  int id;
  immutable(Exception) exc;
}
```

```
void workerThread() {
  try /* ... */

  catch (Exception exc) {
    ownerTid.send(WorkerError(id, cast(immutable)exc));
  }
// ...
}
```

```
    receive(
      (const(WorkerError) msg) {
        // ...
      },
      // ...
    );
```

Error

The program is in an illegal state.

```
assert(name.length == 42, format!"Wrong name: %s"(name));
```

- Should not **`catch()`** (in theory)
- Should not **`format()`** (in theory)
- Should not **`abort()`** (in theory)
- Should not do anything (in theory)

Error

The program is in an illegal state.

```
assert(name.length == 42, format!"Wrong name: %s"(name));
```

- Should not **`catch()`** (in theory)
- Should not **`format()`** (in theory)
- Should not **`abort()`** (in theory)
- Should not do anything (in theory)

One practical approach applied by D runtime for the main thread:

1. Catch
2. Report
3. Abort

*See: **`rt_trapExceptions`** and **`--DRT-trapExceptions=0`**[1] for changing the behavior of the main thread.*

[1] http://arsdnet.net/this-week-in-d/2016-aug-07.html

std.concurrency (continued)

Reporting **Error**:

```
void workerThread() {
  try /* ... */

  catch (Error err) {       // Contrary to theory

    stderr.writeln(err);    // Wishful thinking: Does stderr even exist?

    import core.stdc.stdlib;
    abort();
  }
}
```

`std.concurrency` (continued)

Passing mutable data between threads:

```
auto workers =
  4.iota
  .map!(i => spawnLinked(&workerThread,
                         cast(shared)new int[42]))
  .array;
```

*Note: **`immutable`** data is implicitly **`shared`** (e.g. **`string`**).*

std.concurrency (continued)

Passing mutable data between threads:

```
auto workers =
  4.iota
  .map!(i => spawnLinked(&workerThread,
                         cast(shared)new int[42]))
  .array;
```

Note: ***immutable*** *data is implicitly* ***shared*** *(e.g.* ***string****).*

Worker thread must take **shared** and likely cast it away:

```
void workerThread(shared(int[]) data) {  // Take shared
  try {
    workerThreadImpl(cast(int[])data);   // Cast shared away
  }
  // ...
}

void workerThreadImpl(int[] data) {      // Non-shared happily ever after
  // ...
}
```

`std.concurrency` (continued)

Passing mutable data between threads:

```
auto workers =
  4.iota
  .map!(i => spawnLinked(&workerThread,
                         cast(shared)new int[42]))
  .array;
```

Note: ***`immutable`*** *data is implicitly* ***`shared`*** *(e.g.* ***`string`****).*

Worker thread must take **`shared`** and likely cast it away:

```
void workerThread(shared(int[]) data) {  // Take shared
  try {
    workerThreadImpl(cast(int[])data);   // Cast shared away
  }
  // ...
}

void workerThreadImpl(int[] data) {      // Non-shared happily ever after
  // ...
}
```

- Warning: Do not actually *share* this data between threads!

`std.concurrency` (continued)

Single-slide example. :o) Each worker thread either succeeds or fails with either **`Exception`** or **`Error`**.

```
import std;    // Importing the entire package for terseness.

void main() {
  auto workers = 4.iota
                 .map!(id => spawnLinked(&workerThread,
                                         id,
                                         cast(shared)new int[42]))
                 .array;

  size_t completed = 0;
  while (completed != workers.length) {
    receive(
      (const(LinkTerminated) msg) {
        completed++;
      },

      (const(WorkerError) msg) {
        writefln!"Worker %s failed: %s"(msg.id, msg.exc.msg);
      },

      (const(WorkerReport) msg) {
        writefln!"Worker %s finished successfully with %s."(msg.id, msg.data);
      },
    );
  }
}

struct WorkerError {
  int id;
  immutable(Exception) exc;
}

void workerThread(int id, shared(int[]) data) {
  try {
    workerThreadImpl(id, cast(int[])data);  // Dispatch to the implementation

  } catch (Exception exc) {
    ownerTid.send(WorkerError(id, cast(immutable)exc));

  } catch (Error err) {
    stderr.writeln(err);
    import core.stdc.stdlib : abort;
    abort();
  }
}
```

```
struct WorkerReport {
  int id;
  int data;
}

void workerThreadImpl(int id, int[] data) {
  foreach (d; data) {
    // We will fail with some probability
    failMaybe(id, data.length);
  }

  // Survived without an error; send report.
  ownerTid.send(WorkerReport(id, 42));
}

// This function simulates an operation that may fail
void failMaybe(int id, size_t length) {
  auto msg(string kind) {
    return format!"Worker %s is throwing %s."(id, kind);
  }

  // Succeeds most of the time
  final switch (dice(length * 5, 1, 1)) {
  case 0:
    break;

  case 1:
    enforce(false, msg("Exception"));
    break;

  case 2:
    assert(false, msg("Error"));
    break;
  }
}
```

Nested functions

```
void foo() {
  foreach (i; 0 .. n) {
    if (a[i].p.q.r.color == "red" &&
        b[i].p.q.r.color == "green") {
      // ...
      enforce(c, format!"illegal: %s"(a[i].p.q.r.color));
    }
  }
}
```

Nested functions

```
void foo() {
  foreach (i; 0 .. n) {
    if (a[i].p.q.r.color == "red" &&
        b[i].p.q.r.color == "green") {
      // ...
      enforce(c, format!"illegal: %s"(a[i].p.q.r.color));
    }
  }
}
```

Nested function for reducing code duplication and readability:

```
void foo() {
  foreach (i; 0 .. n) {
    auto color(S[] arr) {           // Nested function
      return arr[i].p.q.r.color;    // Using 'i' from the enclosing scope
    }

    if (color(a) == "red" && color(b) == "green") {
      // ...
      enforce(c, format!"illegal: %s"(color(a)));
    }
  }
}
```

Nested functions (continued)

```
struct RGB {
  ubyte red;
  ubyte green;
  ubyte blue;

  this(uint value)
    ubyte popLowByte() {
      ubyte b = value & 0xff;    // Uses 'value' from the enclosing scope
      value >>= 8;
      return b;
    }

    this.blue = popLowByte();
    this.green = popLowByte();
    this.red = popLowByte();
  }
}
```

Nested functions (continued)

```
void foo() {
  // The message is evaluated lazily: GOOD
  enforce(a, format!"illegal: %s"(x));
  // Code duplication: BAD
  enforce(b, format!"illegal: %s"(x));
}
```

Nested functions (continued)

```
void foo() {
  // The message is evaluated lazily: GOOD
  enforce(a, format!"illegal: %s"(x));
  // Code duplication: BAD
  enforce(b, format!"illegal: %s"(x));
}
```

Not good enough:

```
  const msg = format!"illegal: %s"(x);  // Evaluated eagerly: BAD
  enforce(a, msg);
  enforce(b, msg);                      // No code duplication: GOOD
```

Nested functions (continued)

```
void foo() {
  // The message is evaluated lazily: GOOD
  enforce(a, format!"illegal: %s"(x));
  // Code duplication: BAD
  enforce(b, format!"illegal: %s"(x));
}
```

Not good enough:

```
  const msg = format!"illegal: %s"(x);  // Evaluated eagerly: BAD
  enforce(a, msg);
  enforce(b, msg);                      // No code duplication: GOOD
```

Nested function for lazy evaluation:

```
  auto msg() {
    return format!"illegal: %s"(x);
  }

  enforce(a, msg);
  enforce(b, msg);
```

Unmentionable types of range objects

Can't spell out *unmentionable* types:

```
struct S {
  ??? r;

  this(string fileName) {
    this.r = File(fileName).byLine;
  }
}
```

Unmentionable types of range objects

Can't spell out *unmentionable* types:

```
struct S {
  ??? r;

  this(string fileName) {
    this.r = File(fileName).byLine;
  }
}
```

One solution is to return the expression from a function:

```
auto makeRange(string fileName = null)    // ← Defaulted for convenience
in (!fileName.empty) {                    // ← Checked against null
  return File(fileName).byLine;
}
```

Unmentionable types of range objects

Can't spell out *unmentionable* types:

```
struct S {
  ??? r;

  this(string fileName) {
    this.r = File(fileName).byLine;
  }
}
```

One solution is to return the expression from a function:

```
auto makeRange(string fileName = null)    // ← Defaulted for convenience
in (!fileName.empty) {                    // ← Checked against null
  return File(fileName).byLine;
}
```

```
struct S {
  typeof(makeRange()) r;

  this(string fileName) {
    this.r = makeRange(fileName);
  }
}
```

Initializing a non-mutable variable

A known technique from other languages; nothing special about D here:

```
auto a = someValue();

if (condition) {
  doSomethingElse();
  a = someOtherValue();
}
```

Trouble: **`a`** cannot be **`const`** or **`immutable`**.

Initializing a non-mutable variable

A known technique from other languages; nothing special about D here:

```
auto a = someValue();

if (condition) {
  doSomethingElse();
  a = someOtherValue();
}
```

Trouble: **a** cannot be **const** or **immutable**.

Putting the whole logic into a lambda is a solution:

```
const a = {
  if (condition) {
    doSomethingElse();
    return someOtherValue();
  }
  return someValue();
}();
```

std.typecons.Flag

Typed flags instead of **bool**:

```
void foo(Flag!"compress" compress, Flag!"skip" skip) {
  // ...
}

  foo(Yes.compress, No.skip);
```

std.typecons.Flag

Typed flags instead of **bool**:

```
void foo(Flag!"compress" compress, Flag!"skip" skip) {
  // ...
}

  foo(Yes.compress, No.skip);
```

Checked at compile time:

```
  foo(No.skip, Yes.compress);    // ← compilation ERROR; good
```

std.typecons.Flag

Typed flags instead of **bool**:

```
void foo(Flag!"compress" compress, Flag!"skip" skip) {
  // ...
}

  foo(Yes.compress, No.skip);
```

Checked at compile time:

```
  foo(No.skip, Yes.compress);    // ← compilation ERROR; good
```

However, passing existing **bool** is not pleasant:

```
  bool compress;
  bool skip;

  foo(compress ? Yes.compress : No.compress,
      skip     ? Yes.skip     : No.skip);
```

Note: Other options are not pleasant either.

std.typecons.Flag (continued)

One solution is **alias** template parameters:

```
auto toFlag(alias variable)() {
  enum name = variable.stringof;

  mixin ("return variable ? Yes." ~ name ~ " : No." ~ name ~ ";");
}
```

std.typecons.Flag (continued)

One solution is **alias** template parameters:

```
auto toFlag(alias variable)() {
  enum name = variable.stringof;

  mixin ("return variable ? Yes." ~ name ~ " : No." ~ name ~ ";");
}
```

```
  bool compress;
  bool skip;

  // Types are Flag!"compress" and Flag!"skip":
  foo(toFlag!compress, toFlag!skip);
```

Module as a "singleton" object

Assume a top-level module function:

```
void topLevel(int[] a, string[] s) {
  // ... calls a graph of dozens of other functions ...
}
```

Module as a "singleton" object

Assume a top-level module function:

```
void topLevel(int[] a, string[] s) {
  // ... calls a graph of dozens of other functions ...
}
```

Assume a new variable is introduced:

```
void topLevel(int[] a, string[] s, Flag!"verbose" verbose) {
  // ...
}
```

Module as a "singleton" object

Assume a top-level module function:

```
void topLevel(int[] a, string[] s) {
  // ... calls a graph of dozens of other functions ...
}
```

Assume a new variable is introduced:

```
void topLevel(int[] a, string[] s, Flag!"verbose" verbose) {
  // ...
}
```

Now dozens of function signatures may need to be modified:

```
void foo(int i, Flag!"verbose" verbose) {
  // ...
}

void bar(double d, Flag!"verbose" verbose) {
  // ...
}

// ... many more ...
```

Module as a "singleton" object (continued)

A solution is to use a module variable:

```
Flag!"verbose" verbose;

void topLevel(int[] a, string[] s, Flag!"verbose" verbose) {
  .verbose = verbose;
  // ...
}

void foo(int i) {     // No need to change
  // ...
}

void bar(double d) {  // No need to change
  // ...
}
```

Module as a "singleton" object (continued)

A solution is to use a module variable:

```
Flag!"verbose" verbose;

void topLevel(int[] a, string[] s, Flag!"verbose" verbose) {
  .verbose = verbose;
  // ...
}

void foo(int i) {     // No need to change
  // ...
}

void bar(double d) {  // No need to change
  // ...
}
```

BUG: Only this thread is affected! To make it "per program":

Module as a "singleton" object (continued)

A solution is to use a module variable:

```
Flag!"verbose" verbose;

void topLevel(int[] a, string[] s, Flag!"verbose" verbose) {
  .verbose = verbose;
  // ...
}

void foo(int i) {     // No need to change
  // ...
}

void bar(double d) {  // No need to change
  // ...
}
```

BUG: Only this thread is affected! To make it "per program":

```
shared Flag!"verbose" verbose;
```

Module as a "singleton" object (continued)

A solution is to use a module variable:

```
Flag!"verbose" verbose;

void topLevel(int[] a, string[] s, Flag!"verbose" verbose) {
  .verbose = verbose;
  // ...
}

void foo(int i) {     // No need to change
  // ...
}

void bar(double d) {  // No need to change
  // ...
}
```

BUG: Only this thread is affected! To make it "per program":

```
shared Flag!"verbose" verbose;
```

***Warning**: : May not work as desired if* **`topLevel`** *is called multiple times.*

Parsing files at compile time

Requirements:

- A program that parses a file at run time

```
$ my_program file.txt
```

Parsing files at compile time

Requirements:

- A program that parses a file at run time

```
$ my_program file.txt
```

- The file should be optional

```
$ my_program           ← Use default content
```

Parsing files at compile time

Requirements:

- A program that parses a file at run time

```
$ my_program file.txt
```

- The file should be optional

```
$ my_program           ← Use default content
```

Boring in D: The same function for compile time and run time.

```
// Returns significant lines of the content
string[] parse(string content) {
  auto isComment = (string line) => line.startsWith('#');
  auto isSignificant = (string line) => !line.empty && !isComment(line);

  return content
         .splitter('\n')          // Split by lines
         .map!strip               // Strip whitespace
         .filter!isSignificant
         .array;
}
```

Parsing files at compile time (continued)

`import` expression to read a file at compile time:

```
immutable defaultList = parse(import("default_file.txt"));
```

Parsing files at compile time (continued)

import expression to read a file at compile time:

```
immutable defaultList = parse(import("default_file.txt"));
```

Alternatives:

```
static const defaultList = /* ... */;
enum         defaultList = /* ... */; // Generally more costly at run time
```

Parsing files at compile time (continued)

`import` expression to read a file at compile time:

```
immutable defaultList = parse(import("default_file.txt"));
```

Alternatives:

```
static const defaultList = /* ... */;
enum         defaultList = /* ... */; // Generally more costly at run time
```

User code:

```
void main(string[] args) {
  auto theList = (args.length == 1
                  ? defaultList
                  : args[1].readText.parse);
  // ...
}
```

Conclusion

- D is a powerful engineering tool.
- D is very productive.
- D is very much fun.