Productive D

An Experience Report

Ali Çehreli • acehreli@yahoo.com

D Programming Language Symposium at Yale • April 11-12, 2026

Ali

- Self-taught software engineer with C and C++ background
- Using D since 2009
  - Semi-active on the D forums
  - Occasional presenter at DConf, D Programming Language Symposium, and other venues
  - Self-published the book "Programming in D" in 2015 (a happy accident that is freely available online)
- Communities
  - A founding member of the D Language Foundation
  - Co-organizer of the Silicon Valley DLang meetups (defunct)
  - Former co-organizer of the Silicon Valley C++ meetups (defunct)

Contents

- Experience
  - Mostly D
  - Some C++
- Thoughts on
  - Coding
    - Guidelines
    - Semantics
  - Software Engineering

Software Engineering

- A craft
- Fun
- Rewarding
- Seeking simplicity
- Emergent
    - Concepts
    - Semantics
    - Code structure (functions, structs, etc.)
    - Names (type, function, variable, etc.)

Does not involve the following:

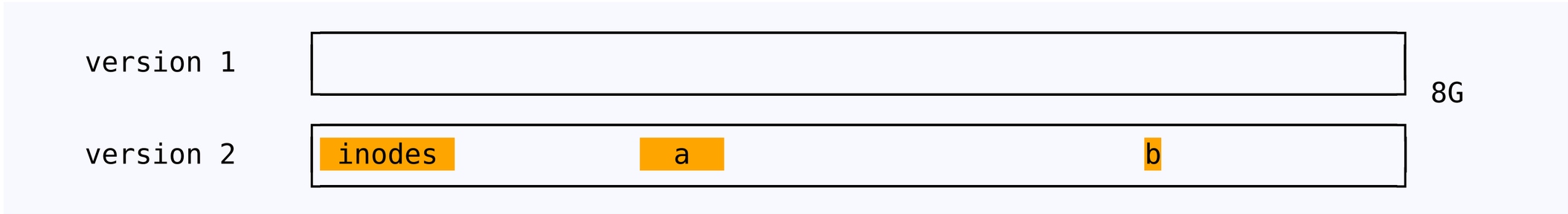
- Perfectionism
- Applying guidelines blindly

Productive D

- Designed properties
  - Simpler
  - Safer
  - More correct
  - Faster
  - Time saving
  - Sane
  - Elegant
  - ...
- Emergent properties:
  - Pragmatic
  - Refactorable (Moldable)
  - Less code
  - Fun
  - Great community
  - ...

Current project

- Automotive industry
- Differential software updates of file system partition images



The goal is to transfer only the differences (the delta), not the entire 8G

See also: DConf 2019 presentation Using D for ROS Bag File Manipulation for Autonomous Driving.

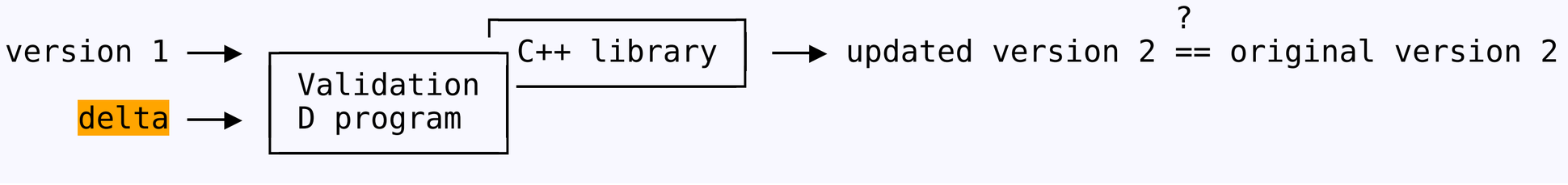
D Program, C++ Library

- D program generates the delta at release time

- C++ library applies the delta on the vehicle

Validation

- Another D program uses the same C API of the library
- Applies the generated delta
- Proves that the library updates the exact "version 2" partition with no bit difference



Lines of Code

Excerpt from a `cloc` output:

```
  Language      Files       Lines of code
    D             ~20         ~6.2K  (4K actual code)
    C++           ~15         ~3.5K  (2K actual code)
```

Library API in C

C is considered to be the *lingua franca* of APIs.

```
typedef struct {

    void* arg;      // Caller's "this" pointer
    Slice delta;    // Bytes of the delta

    // ...

} Context;
```

```
Status applyDelta(Context* context);
```

Library API in C

C is considered to be the *lingua franca* of APIs.

```
typedef struct {

    void* arg;      // Caller's "this" pointer
    Slice delta;    // Bytes of the delta

    // ...

} Context;
```

```
Status applyDelta(Context* context);
```

The implementation is in C++; let's call the header `my_lib.h`:

```
#if defined __cplusplus
extern "C" {
#endif

// ... declarations of the API ...

#if defined __cplusplus
} // extern "C"
#endif
```

See also: DConf 2020 presentation Exposing a D Library to Python Through a C API.

Library API in C - Callback functions

The concrete operations of the system are handled by callback functions:

```
typedef struct {

    // ...

    BlockWriter* writer;            // Writes blocks to target partition
    ProgressReporter* reporter;     // Reports progress (e.g. "50% completed")
    Logger* logger;                 // Logs messages

} Context;
```

Function pointer definitions:

```
typedef int BlockWriter(void* arg, Slice block, uint64_t offset);

typedef int ProgressReporter(void* arg, uint64_t progressed, uint64_t total);

typedef int Logger(void* arg, StringView message);
```

ImportC

Given the header **`my_lib.h`**:

```
// ...

Status applyDelta(Context* context);

// ...
```

1) Include that header in a **`my_lib.c`** file so that the *C preprocessor is run automatically*:

```
#include "my_lib.h";
// EOF
```

2) Import the C file as a regular module:

```
import my_lib;    // <-- Importing a C file

// ...

    Context context;
    // ...
    auto status = applyDelta(&context);
```

See also: ImportC Specification

ImportC - Some Examples

```
// C
#define COLOR 0x123456
#define HELLO "hello"
```

```
// ImportC's interpretation
enum COLOR = 0x123456;
enum HELLO = "hello";
```

```
// C
#define ABC a + b
#define DEF(a) (a + x)
```

```
// ImportC's interpretation
auto ABC() { return a + b; }
auto DEF(T)(T a) { return a + x; }
```

Disclaimer: Will not work for all header files out of the box.

ImportC - Some Examples

```
// C
#define COLOR 0x123456
#define HELLO "hello"
```

```
// ImportC's interpretation
enum COLOR = 0x123456;
enum HELLO = "hello";
```

```
// C
#define ABC a + b
#define DEF(a) (a + x)
```

```
// ImportC's interpretation
auto ABC() { return a + b; }
auto DEF(T)(T a) { return a + x; }
```

Disclaimer: Will not work for all header files out of the box.

Fun fact: ImportC's extension `__import` allows importing C files as modules *from C*:

```
// C
__import my.other.c.file;    // From directory my/other/c/file.c

// ...
```

Component Programming

Chained range operations make the flow of code clear:

```
    File("foo.txt")         // start with a file
    .byLine                 // traverse line-by-line
    .map!strip              // remove leading and trailing spaces
    .filter!(not!empty)     // keep the non-empty ones
    .map!capitalize         // capitalize each
    .take(42)               // but only the first 42
```

Component Programming

Chained range operations make the flow of code clear:

```
    File("foo.txt")         // start with a file
    .byLine                 // traverse line-by-line
    .map!strip              // remove leading and trailing spaces
    .filter!(not!empty)     // keep the non-empty ones
    .map!capitalize         // capitalize each
    .take(42)               // but only the first 42
```

That syntax thanks to

- UFCS
- Optional function call parentheses

```
    take(map!capitalize(filter!(not!empty)(map!strip(File("foo.txt").byLine))), 42);
```

Component Programming

Chained range operations make the flow of code clear:

```
    File("foo.txt")         // start with a file
    .byLine                 // traverse line-by-line
    .map!strip              // remove leading and trailing spaces
    .filter!(not!empty)     // keep the non-empty ones
    .map!capitalize         // capitalize each
    .take(42)               // but only the first 42
```

That syntax thanks to

- UFCS
- Optional function call parentheses

```
    take(map!capitalize(filter!(not!empty)(map!strip(File("foo.txt").byLine))), 42);
```

However, I use loops if it's easier to understand:

```
    foreach (foo; foos) {
        // ...
        foreach (bar; foo.bars) {
            // ...
        }
    }
```

unittest

For code correctness:

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

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

For documentation:

```
string dumpHex(R)(R bytes) {
    // ...
}

unittest {
    auto bytes = iota(ubyte('a'), ubyte('q'));

    assert(dumpHex(bytes) ==
           "61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70    abcdefghijklmnop");
}
```

unittest

For code correctness:

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

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

For documentation:

```
string dumpHex(R)(R bytes) {
    // ...
}

unittest {
    auto bytes = iota(ubyte('a'), ubyte('q'));

    assert(dumpHex(bytes) ==
           "61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70    abcdefghijklmnop");
}
```

However,

- Not all of my functions have unit tests
- Test driven development (TDD) is too much work

parallel

Every non-trivial program of mine involved *embarrassingly parallel* operations:

- Processing multiple files independently
- Processing multiple objects independently
- etc.

```
import std.parallelism;

// ...

    foreach (file; files.parallel) {
        // ... this code block is executed in parallel for each file
    }
```

Library Implementation in C++ - Slice types

- Implemented D-style slice types in C++
- C++ code became easier to work with
- **`std::span`** (C++20) and **`std::string_view`** (C++17) were not available to this C++14 project

```
typedef struct {

    void* ptr;
    uint64_t length;

} Slice;
```

Library Implementation in C++ - Slice types

- Implemented D-style slice types in C++
- C++ code became easier to work with
- **`std::span`** (C++20) and **`std::string_view`** (C++17) were not available to this C++14 project

```
typedef struct {

    void* ptr;
    uint64_t length;

} Slice;
```

Special purpose functions:

```
// Transfer bytes from 'from' to 'to'. ('to' must be before 'from' in memory.)
//
// Example: 2-byte transfer
//
//   before:      |  to  |    from    |
//    after:      |   to   |   from   |

void transferLeft(Slice& from, uint64_t length, Slice& to) {
    // .. precondition checks here ...

    to.length += length;
    from.length -= length;
    from.ptr += length;
}
```

Library API - Error reporting

```
typedef struct {

    int code;                       // 0 means success
    StringView* errorStrings;       // A stack of error strings
    uint64_t errorStringCount;      // Number of strings

} Status;

Status applyDelta(Context* context);
```

Library API - Error reporting

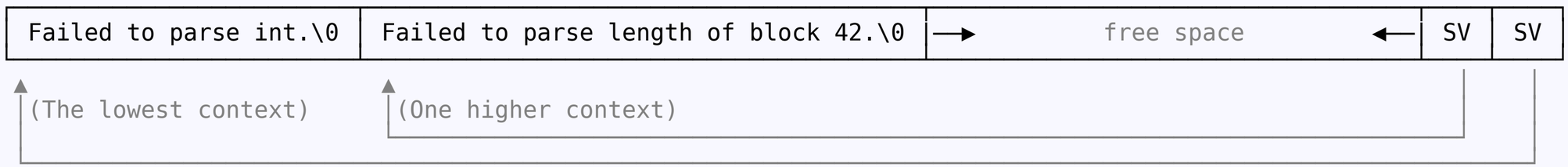
```
typedef struct {

    int code;                       // 0 means success
    StringView* errorStrings;       // A stack of error strings
    uint64_t errorStringCount;      // Number of strings

} Status;

Status applyDelta(Context* context);
```

A single 4K static buffer is used for both the errors and their views:



- The first error is the error reported by the lowest call context
- Error strings are placed from the beginning
- The views are placed from the end
- **Important:** Running out of space is not an error

Obscure Encodings in Hexadecimal Output

```
alias EncodingType = ushort;

enum Encoding : EncodingType {
    monday,     // == 0
    tuesday,    // == 1
}
```

```
00 00 01 00 00 00 03 00 04 00 01 00 06 00 07 00 08 00 09 00    ......................
```

Obscure Encodings in Hexadecimal Output

```
alias EncodingType = ushort;

enum Encoding : EncodingType {
    monday,     // == 0
    tuesday,    // == 1
}
```

```
00 00 01 00 00 00 03 00 04 00 01 00 06 00 07 00 08 00 09 00    ......................
```

Solution: Readable values:

```
enum Encoding : EncodingType {
    monday = readableEncoding("MO"),
    tuesday = readableEncoding("TU"),
}
```

```
00 00 01 00 4d 4f 03 00 04 00 54 55 06 00 07 00 08 00 09 00    ....MO....TU........
```

`readableEncoding()` Function

Possible implementation of `readableEncoding`:

```
EncodingType readableEncoding(string str)
    in (str.length == EncodingType.sizeof,
        format!"Should have %s characters: '%s'"(EncodingType.sizeof, str)) {

    return (str
            .retro
            .fold!((sum, ch) => (sum << 8) | ch)
            .to!EncodingType);
}
```

- `in` for function precondition check
- `format` for useful error string
- Component programming for the win

`const` buffer

`const` is the promise "I shall not mutate your data."

Customarily, API functions take buffers as references to `const`:

```
Status applyDelta(const uint8_t * buffer, uint64_t length) {
    // ...
}
```

`const` buffer

`const` is the promise "I shall not mutate your data."

Customarily, API functions take buffers as references to `const`:

```
Status applyDelta(const uint8_t * buffer, uint64_t length) {
    // ...
}
```

However, mutating the buffer provided performance optimization opportunities. So, I behaved disrespectfully and removed `const`:

```
Status applyDelta(       uint8_t * buffer, uint64_t length) {
    // ...
}
```

Summary

D is a productive engineering tool.