Design by Introspection

Andrei Alexandrescu, Ph.D.

2017-11-03
What’s the Deal??
Initial Motivation

- Systems-level programming a necessity
- Faster is better—no “good enough” limit
- Ever-growing modeling needs
- No room below, no escape: all in same language
  - Runtime support
  - Machine, device interface
  - Base library
Design Principles

- Multi-paradigm; balanced
- Practical
- Principled
- Avoid arcana; turtles all the way down
Hello, World!

```rdmd
#!/usr/bin/rdmd
import std.stdio;
void main() {
    writeln("Hello, world!");
}
```

- “Meh”worthy
- However:
  - Simple
  - Correct
  - Scriptable
  - Features turtles
#!/usr/bin/rdmd

```rmd
void main() {
    import std.stdio;
    writeln("Hello, world!");
}
```

- Most everything can be scoped everywhere
- Better scoping, reasoning
- Functions
- Types (Voldemort types)
- Even generics
Segue into generics

```plaintext
void log(T)(T stuff) {
    import std.datetime, std.stdio;
    writeln(Clock.currTime(), ' ', stuff);
}

void main() {
    log("hello");
}
```

- If not instantiated, no import
- import cached once realized
- Generics faster to build, import
- Less pressure on linker
Make Language Features Simple & Combinable
Example: **static if** statement

- Started as “#if done right”
- Allowed new interesting things
- Put pressure on combinability
  - What expressions can I evaluate statically?
  - What program elements can I examine?
struct RobinHashTable(K, V, size_t maxLength) {
    static if (maxLength < ushort.max-1) {
        alias CellIdx = ushort;
    } else {
        alias CellIdx = uint;
    }

    static if (K.sizeof % 8 < 7) {
        align(8) struct KV {
            align(1):
                K k;
                ubyte cellData;
            align(8):
                V v;
        }
    }
}
Example: Hash table layout

... else {
    align(8) struct KV {
        K k;
        align(8):
        V v;
        align(1):
        ubyte cellData;
    }
}
Design Patterns and Friends
Semi-Automatic use of Design Patterns
Coined by “Modern C++ Design” in 2001
Enjoys use in C++, D
Inducted in Wikipedia’s “hall of fame” at
http://en.wikipedia.org/wiki/Programming_paradigm
(along with 75 others)
“[...] the Design Patterns solution is to turn the programmer into a fancy macro processor.”

– Mark Dominus
“What would happen if we could arrange the atoms one by one the way we want them?”

– Richard P. Feynman
Core Idea

- Patterns: programmer “expands” mental macros
  - Total plasticity, no code reuse
- PBD: programmer assembles rigid macros
  - No plasticity, good code reuse
- DbI: programmer *molds* macros that communicate with, and adapt to, one another
  - Good plasticity, good code reuse
DbI Prerequisites

- DbI Input
DbI Prerequisites

- DbI Input
  - Introspect types: “What are your methods?”
  - Variant: “Do you support method xyz?”
DbI Prerequisites

- DbI Input
  - Introspect types: “What are your methods?”
  - Variant: “Do you support method xyz?”
- DbI Processing
DbI Prerequisites

- DbI Input
  - Introspect types: “What are your methods?”
  - Variant: “Do you support method xyz?”
- DbI Processing
  - Arbitrary compile-time evaluation
DbI Prerequisites

- DbI Input
  - Introspect types: “What are your methods?”
  - Variant: “Do you support method xyz?”
- DbI Processing
  - Arbitrary compile-time evaluation
- DbI Output
DbI Prerequisites

- DbI Input
  - Introspect types: “What are your methods?”
  - Variant: “Do you support method xyz?”
- DbI Processing
  - Arbitrary compile-time evaluation
- DbI Output
  - Generate arbitrary code
How does D stack up?

- DbI Input
How does D stack up?

- DbI Input
  - `tupleof, __traits, ...`
How does D stack up?

- DbI Input
  - tupleof, `_traits`, ...
- DbI Processing
How does D stack up?

- DbI Input
  - `tupleof`, `__traits`, ...
- DbI Processing
  - CTFE, `static if`, ...
How does D stack up?

- DbI Input
  - `tupleof, __traits, ...`
- DbI Processing
  - `CTFE, static if, ...`
- DbI Output
How does D stack up?

- **DbI Input**
  - tupleof, \_\_traits, ...

- **DbI Processing**
  - CTFE, static if, ...

- **DbI Output**
  - template expansion, mixin, ...
Optional Interfaces
Optional Interfaces

- A DbI component typically prescribes:
  - $n_r$ required primitives (may be 0)
  - $n_o$ optional primitives
- Introspection queries for optionals
- What’s missing as important as what’s present

- Up to $2^{n_o}$ possible interfaces, in compact form!
Optional Interfaces: Aftermath

- Linear code for exponential behaviors
  - Includes state variations, too
  - `static if` the “magic design fork”
- No penalty for fat interfaces
- Graceful degradation
  - Old: Less capable components $\Rightarrow$ errors
  - New: Less capable components $\Rightarrow$ reduced features
Each use of \texttt{static if} doubles the design space covered.
Realized Designs

- `std.experimental.allocation`: unbounded allocator designs in 12 KLOC
  - `jemalloc`: 1 allocator in 45 KLOC
- Collections: see talk by Eduard Stănîloiu
- `std.experimental.checkedint`: now
Checked Integrals

- $+$, $+=$, $-$, $-=$, $++$, $--$, $*$, $*=$ may lose information
- Division by zero in $/$, $/$=
- $-x$.min negative for all signed types
- $-1 == \texttt{uint}.\text{max}$, $-1 > 2u$

- That’s pretty much it!
Possible Designs (1/2)

- Options that come at a runtime cost
  - Integrate in the programming language
  - Do away with fixed-size arithmetic altogether
- Have the programmer insert tests appropriately
  - For an appropriate definition of “appropriately”
  - Bulky, difficult to follow, fragile
Possible Designs (2/2)

- Designate “checked integral” types
- Hook all operations and insert checks
- User replaces primitive types with these
  - Selectively depending on safety/speed tradeoff
- Requires user-defined operator overloading
Design Challenges

- What gets checked: overflows? div0? negation? mixed-sign comparisons? conversions? some of the above—which?
- Type system integration: statically disallow some operators/conversions?
- Make it efficient (not easy!)

- Make it small
  - Proportional response
  - Not rocket surgery after all
Meta Design Challenges

- No trouble to implement any given behavior
- Much more difficult to allow behaviors that are as of yet unspecified
- Scaffolding scales poorly with behaviors
- “Sticker shock” of generic libraries
  - “You mean I need to use this 5 KLOC library coming with 20 pages of documentation to check a few overflows?”
Baselines

- Mozilla’s CheckedInt for C++
- Microsoft’s SafeInt for C++
- safe_numerics for C++ by Robert Ramey
- checkedint for D by T. S. Bockman
• 3 KLOC (code + unit tests + documentation)
• Code: 1200 LOC
• Tests: 900 LOC
• Documentation: 900 LOC

• Speed: comparable to hand-inserted checks
• Flexibility: unbounded
Overall Design

- “Shell with hooks” approach
- Shell: high-level language integration
- Hook: optional intercepts of ops/events
- Default hook: just abort on anything fishy

```cpp
struct Checked(T, Hook = Abort) if (isIntegral!T) {
    private T payload;
    Hook hook;
    ...
}
```
Stateless hook? No problem!

```c
struct Checked(T, Hook = Abort) if (isIntegral!T) {
    private T payload;
    static if (stateSize!Hook > 0) Hook hook;
    else alias hook = Hook;
    ...
}
```
The Shell

• Factors all commonalities
• Handles qualifiers
• Drives hooks
• Type system integration (bool, float etc)
• Composition mediation

• Uses introspection to “look” at hooks
  ○ What can you do?
  ○ What operation(s) are you interested in?
Defined Hook Primitives

- **Statics:** `defaultValue`, `min`, `max`
- **Event handling:** `onBadCast`, `onOverflow`, `onLowerBound`, `onUpperBound`
void opUnary(string op)()
if (op == "++" || op == "--") {
  static if (hasMember!(Hook, "hookOpUnary")) {
    hook.hookOpUnary!op(payload);
  } else static if (hasMember!(Hook, "onOverflow")) {
    ...
  } else {
    mixin(op ~ "payload;");
  }
}
Defined Hooks

- Abort
- Throw
- Warn: output issues to stderr
- ProperCompare: fix comparisons on the fly
- WithNaN: Reserve “not a number” value
- Saturate: sticky saturation instead of overflowing

- Your own
  - Average length: 50 lines
Hook Example

- No Pesky Comparisons

```c
struct NoPeskyCmps {
    static int hookOpCmp(Lhs, Rhs)(Lhs lhs, Rhs rhs) {
        const result = (lhs > rhs) - (lhs < rhs);
        if (result > 0 && lhs < 0 && rhs >= 0 ||
            result < 0 && lhs >= 0 && rhs < 0) {
            assert(0, "Mixed-signed comparison failed.");
        }
        return result;
    }

    alias MyInt = Checked!(int, NoPeskyCmps);
}
```
Flexibility

- No Pesky Comparisons—EVAR!

```c
struct NoPeskyCmpsEver {
    static int hookOpCmp(Lhs, Rhs)(Lhs lhs, Rhs rhs) {
        static if (lhs.min < 0 && rhs.min >= 0 &&
                   lhs.max < rhs.max || rhs.min < 0 &&
                   lhs.min >= 0 && rhs.max < lhs.max) {
            static assert(0, "Mixed-sign comparison of " ~
                           Lhs.stringof ~ " and " ~ Rhs.stringof ~
                           " disallowed. Cast one of the operands.");
        }
        return (lhs > rhs) - (lhs < rhs);
    }

    alias MyInt = Checked!(int, NoPeskyCmpsEver);
}
```
Design by Introspection

- Assembly with plastic, adaptable components
- Combine:
  - *static if*
  - Compile-time introspection
  - Compile-time evaluation
  - Code generation
Destructionize!