Compile-Time Types

2019-05-09

Luís Marques <luis@luismarques.eu>

Caveat Emptor

- This is a talk about “compile-time”
- I’m prototyping some of these ideas
- At the moment this is very speculative
- Maybe too early to present but… “present early, present often”
- Heavy skepticism is a healthy reaction
- Feedback still welcome

Peter Principle & D

- Peter principle
  - You are promoted until you reach your level of incompetence
- Management concept
- Applies to other things… like D

Peter Principle & D

- D as an alternative to C
  - Picture a typical C codebase
  - Some parts of C are pretty reasonable. Functions, structs, etc.
    - Some improvements here and there

Peter Principle & D

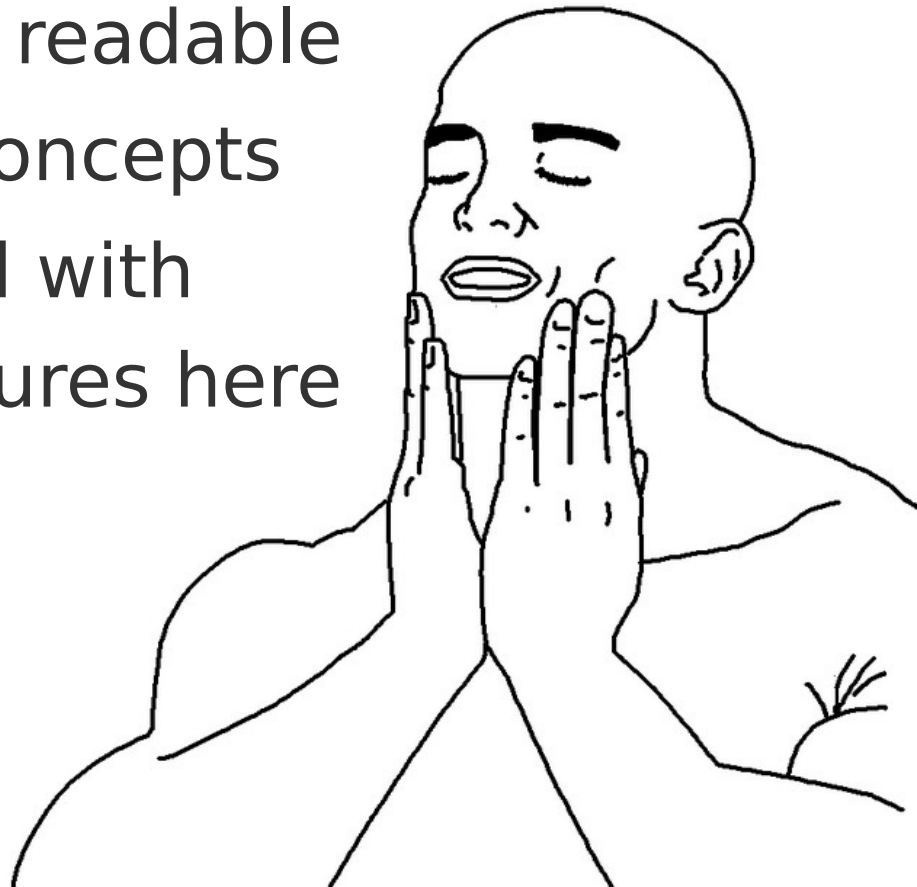
- D as an alternative to C
  - Other parts are a trainwreck
  - Just focusing on the preprocessor...
    - #include headers
    - #define CONSTANTs, MACROs
    - #if, #ifdef

Peter Principle & D

- D as an alternative to C
  - D is created as an improved C
  - #define CONSTANTs become enum consts
  - #define MACROs become functions (inlined, templated, etc.)
  - Sane modules
  - Replaced two separate languages (preprocessor, C) with a single one
    - Safer, more readable, integrated

Peter Principle & D

- D as an alternative to C
  - Things compile super fast
  - The code is simple and readable
  - It mostly uses simple concepts
    - Simple to get started with
    - A few advanced features here and there solve the tougher issues



Peter Principle & D

- D > C
  - Life is good, so why rock the boat?
  - You feel the aluring power of D
  - You no longer want C-style programming
  - You want ranges; generic algorithms; design by instrospection; const and immutable; __traits, mixins, pure, @nogc, @safe...

Peter Principle & D

- Modern D
  - Lots of cool stuff. But sometimes...
  - Slow compilation
  - Inscrutable error messages
  - Compile-time code is inconvenient to debug
  - Features that don’t interact well with each other
  - You often need experts, ugly hacks, etc.

Peter Principle & D

- Modern D
  - D has been promoted to its level of incompetence
  - Why does this happen?

Peter Principle & D

- Modern D
  - D has amassed a lot of features
  - Many of them lack a unifying structure
    - Even if they deal with the same fundamental concepts
    - They clash with each other

Peter Principle & D

- Modern D
  - The programming model is not the most appropriate for modern D style code

The Core of D

- Stroustrup:
  - “Inside C++ is a small elegant language struggling to get out”
- Luís:
  - Is there a smaller, more elegant language struggling to get out of D?
  - What is that core?
    - I will focus on compile-time
  - How do we get there?

D Improvement Process

- Two ways of improving D
- Small, localized problems solved with small, localized fixes
  - The iterative model
  - Provide a compatibility path
  - Examples: ProtoObject, constructors, ...

D Improvement Process

- How do we solve big, fundamental issues?
  - D is large. Lots of moving parts.
  - Keeping up with all the problems that pop up is a losing battle
  - Simplify!
  - Find a few primitives that you can build upon
  - Research project

Compile-Time Model

- What should the core D features be?
- Modern D revolves around compile-time
  - Instrospection
  - Code generation
- You code to an ecosystem of other code
- These should be as natural as regular code

Compile-Time Model

Compile-time is not a single thing

- Different uses
  - Compile-time computation
  - Conditional compilation
  - Metaprogramming
- Disparate implementations
  - Leads to accidental complexity

Compile-Time Model

- “Compile-time vs. compile-time” article
  - https://wiki.dlang.org/User:Quickfur/Compile-time_vs._compile-time
  - Two types of compile-time
    - AST stuff (templates, static if, …)
    - CTFE
  - AST manipulation no access to semantics
  - CTFE has no access to AST manipulation

Compile-Time Model

- “Compile-time vs. compile-time” article
  - “Why can't the compiler read this value at compile-time, since it's clearly known at compile-time?!”

```
int ctfeFunc(bool b)
{
    static if (b) // <--- compile error
        return 1;
    else
        return 0;
}

enum myInt = ctfeFunc(true);
```

Compile-Time Model

- “Compile-time vs. compile-time” article
  - “what does it print?”

```
void func(Args...)(Args args) {
    foreach (a; args) {
        static if (is(typeof(a) == int)) {
            pragma(msg, "is an int");
            continue;
        }
        pragma(msg, "not an int");
    }
}
void main() {
    func(1);
}
```

Compile-Time Model

- What does this print?

```
int foo(int x) {
    return x*2;
}

void main() {
    writeln(typeof(foo).stringof);
}
```

int(int x)

Compile-Time Model

- What about now?

```
int foo()(int x) {
    return x*2;
}

void main() {
    writeln(typeof(foo).stringof);
}
```

void

Template Types

- The type system doesn’t know anything about templates
- Totally reasonable when you think about what a template currently is
- There’s nothing fundamental about this design
- Is it the design that we actually want?

Template Types

- What’s a type?
- A type says:
  - What something represents
  - What its possible states are
  - What operations you can perform with it
- Is this something reasonable to know about a template?

Template Types

- The type of a template:
  - Something you can “instantiate”
  - What you need to instantiate it
  - What you get back
- What should its type be, then?
- What should the exact semantics be?

Template Types

- Semantics of instantiation
  - When? (ordering)
    - How it interacts with declarative features
    - If there are side-effects then ordering matters
    - Yes, there will be side-effects!

Template Types

- Semantics of instantiation
  - How many times? (memoization)
    - Holdover from the cookie cutter model
    - Not always appropriate
      - Repeated side-effects
      - Typedef cookie

```
struct Typedef(T, T init = T.init, string cookie = null);
```

Template Types

- Semantics of “what you get back”
  - You generate something that has a value

```
template Foo() {
    enum value = true;
}

bar(Foo!().value);
```

  - We often just want the value
    - Function-like call and return

Template Types

- Semantics of “what you get back”
  - Eponymous trick

```
template Foo() {
    enum Foo = true;
}

bar(Foo!());
```

  - Reminds me of another language...
  - What about the other fields?

Template Types

- Syntax of instantiation

```
int baz() { return 42; }

template Foo() {
    enum Foo = true;
}

bar(Foo!());
bar(baz);
```

- Accidental complexity: !() required

Compile-Time Computation

- CTFE
  - What does it mean?

19.22 Compile Time Function Execution (CTFE)

1. Functions which are both portable and free of global side-effects can be executed at compile time. In certain contexts, such compile time execution is guaranteed. It is called Compile Time Function Execution (CTFE) then. The contexts that trigger CTFE are:

- initialization of a static variable or a manifest constant
- static initializers of struct/class members
- dimension of a static array
- argument for a template value parameter
- **`static if`**
- **`static foreach`**
- **`static assert`**
- **`mixin`** statement
- **`pragma`** argument

Compile-Time Computation

- CTFE
  - What does it mean?
  - Conceptual simplicity
    - Just call it compile-time execution
  - What should be CTFEable?
    - Is there a simple rule for that?

Compile-Time Integration

- Arbitrary differences tend to compound
  - Example: UFCS
    - `return algorithm3(`
      `algorithm2(algorithm1(data)))`
    - `return data`
      `.algorithm1`
      `.algorithm2`
      `.algorithm3;`

Compile-Time Integration

- Arbitrary differences tend to compound
  - Example: UFCS @ CT
  - https://atilaoncode.blog/2018/12/11/what-d-got-wrong/

```
alias memberNames = AliasSeq!(__traits(allMembers, T));
alias Member(string name) =
   Alias!(__traits(getMember, T, name));
alias members = staticMap!(Member, memberNames);
alias memberFunctions = Filter!(isSomeFunction, members);
```

Compile-Time Integration

- Arbitrary differences tend to compound
  - Example: UFCS @ CT
  - https://atilaoncode.blog/2018/12/11/what-d-got-wrong/

```
alias memberFunctions = __traits(allMembers, T)
    .staticMap!Member
    .Filter!(isSomeFunction);


alias memberFunctions = __traits(allMembers, T)
    .staticMap!(name => Alias!(__traits(getMember, T, name)))
    .Filter!isSomeFunction
```

Compile-Time Integration

- Arbitrary differences tend to compound
  - Example: function arguments
    - foo(&bar)
    - foo!bar
  - Interacts with @property-like behavior, etc.

Compile-Time is Intrusive

- One of the strengths of D is its plasticity

```
struct Foo {
    int x;
}

F f;
foo(f.x)
```

```
struct Foo {
    int _x;
    int x() { return _x*2; }
}

F f;
foo(f.x)
```

- No need to change all the callers

Compile-Time is Intrusive

- One of the strengths of D is its plasticity

```
int x = 7;
int y = 42;
foo(x, y);
```

```
int x = 7;
enum y = 42;
foo!(y)(x);
```

- The caller sites must be changed to forward compile-timeness
  - Well, duh?

Compile-Time is Intrusive

- GCC > D

```
int bar(int x) {
    if (__builtin_constant_p(x) && x < 50)
        return -1;
    return x*2;
}

int foo(int x) {
    return bar(x);
}

int main() { return foo(42); }
```

Compile-Time is Intrusive

- What's the point?

```
#define write_csr(reg, val) ({ \
  if (__builtin_constant_p(val) && (unsigned long)(val) < 32) \
    asm volatile ("csrw " #reg ", %0" :: "i"(val)); \
  else \
    asm volatile ("csrw " #reg ", %0" :: "r"(val)); })


void foo() {
   write_csr(SOME_CSR, 7);
}
```

Compile-Time is Intrusive

- What's the point?

```
string myFormat(string fmt, int value);

auto f1(string fmt, int x) {
    return myFormat(fmt, x);
}

auto f2(int x) {
    return myFormat("x = %d", x);
}

auto f3(int x)() {
    return myFormat("x = %d", x);
}
```

Compile-Time is Intrusive

- What's the point?
  - `writefln!fmt(args)` is now supported
  - We have to change all the call sites
  - How do you take advantage of this overload in generic code?
  - Still doesn't take advantage of compile-time knowledge of `args`

Compile-Time is Intrusive

- What's the point?
  - The `writefln` example is the best case scenario
    - If you take advantage of compile-time knowledge you get a bonus
    - Otherwise, things still work
  - Changes that start requiring an argument to be compile-time are even worse

Experimental Approach

- Solution 0: this is not a problem. That’s just how the language works.
- Solution 1: go over all of the features and try to make them consistent with each other
  - Pros: if you can make it work, the users will be none the wiser
  - Cons: lots of work; I bet it will still leak
- Solution 2:
  - Share common infrastructure

Experimental Approach

- What's the type of this?

```
const x = 42;
```

- What's the DMD output for this?

```
const x = 42;
x = 7;
```

  - Error: cannot modify const expression x

Experimental Approach

- What's the type of this?

```
enum x = 42;
```

- What's the DMD output for this?

```
enum x = 42;
x = 7;
```

  - Error: cannot modify constant x

const expression x

Experimental Approach

- We are missing important information in the type of x
  - Constness
  - Compile-timeness
- How do we add the latter?

Experimental Approach

- How do we declare a compile-time int?
  - We need a “compile-time” type qualifier
  - Let’s go over the options:
    - `compile_time int x = 42;`
    - `comptime int x = 42;`
    - `#int x = 42;`

Experimental Approach

- How do we declare an `enum int`?
  - `#const x = 42;`
  - `#immutable x = 42;`

Experimental Approach

- We can have a module-level mutable `#int x`
  - We just broke the AST/CTFE dichotomy
  - Example:

```
#string[] namesOfStuffUsed;

void foo(#ref f) {
   namesOfStuffUsed ~= f.fullyQualifiedName;
   ...
}
```

Experimental Approach

- Compile-time statements
  - Prefixed with # to enforce compile-time execution
    - Example: `#if(cond)`
  - Unprefixed based on the type of `cond`?
    - No need to rely on the optimizer
- What about the scopes introduced by `{}` ?

Experimental Approach

- What about the scopes introduced by `{}` ?
  - Several possible approaches
  - Prototyping to see how each feels
  - Examples:
    - Contextual (is it a #statement?, etc.)
    - Special braces (`#{}`, etc.)

Experimental Approach

- How do we declare function templates?
  - Old:
    - `foo(int x)(int y);`
  - New:
    - `foo(#int x, int y);`
- Gives a new meaning to UFCS
- No need to reorder: `foo(A a, B b, C c)`

Experimental Approach

- How do we declare function templates?
  - Old:
    - `foo(int x)(int y);`
  - New:
    - `foo(#int x, int y);`
- Gives a new meaning to UFCS
- No need to reorder: `foo(A a, #B b, C c)`

Experimental Approach

- Again, what's the type of a template?
  - It's not "`template`", `void`, etc.
  - It's a `#function`
  - Exact type depends on the type of the parameters and return value
    - `#int #function(#int)`

Experimental Approach

- How do we declare struct templates?
  - Old and new (the hard way)

```
template S(int n) {
   struct S {
      int[n] buffer;
   }
}
```

```
auto S(#int n) {
   struct S {
      int[n] buffer;
   }
   return S;
}
```

  - Requires first class types
  - Annoyingly verbose and opaque. Shorter form desirable

Experimental Approach

- How do we declare struct templates?
  - Old and new (the ~~hard~~ easy way)

```
struct S(int) {
   int[n] buffer;
}
```

```
struct S(#int n) {
   int[n] buffer;
}
```

S!3 becomes S(3)

- Like Peano arithmetic
- Still a #function
- Non-pure #fun impacts type equality

Experimental Approach

- What about other #types?
  - First class types have types themselves
  - Old: `void foo(T)();`
  - New: `void foo(#Type T);`
    - Possibility to integrate typeof & RTTI?
      - #Type vs Type

Experimental Approach

- What about #type deduction?
  - Old: `void foo(T)(T x);`
  - Several possibilities:
    - Wildcards
      - Sparrow: `@AnyType`
      - Jai: `(a: $T, b: T)`
    - Optional / implicit arguments
      - Might require relaxing optional arg rules

Experimental Approach

- What about #too #many #pounds?
  - Putting # everywhere gets old fast
  - Lower #functions
    - `void #foo(int x) { if(x) ... }`
    - `void #foo(#int x) { #if(x) ... }`
  - Helps answer what's CTFEable
    - Make it #fun, lower and do type checking
  - Use #rtFun() to force compile-time execution

Experimental Approach

- Semantics of a mixed template/CTFE model

```
void foo(#int x, int y)
{
    #if(ctFun(x))
        rtFun(y);
}
```

- Like the `struct` long/short form equivalence
- Implicitly returns a runtime `foo` function that implicitly gets called with `y`

Experimental Approach

- Lots of features could be subsumed by #functions
  - pragma(msg, str) => #writeln(str)
  - import(“filename”) => #read
  - __traits => compile-time API
  - Overuse of string mixins
    - mixin(parentName ~ “.” ~ memberName);
    - getMember(parent, memberName);

Prior Art

- I’m not aware of any language that explores quite this region of the design space
  - Sparrow
    - First order types, natural CTFE, etc.
    - Doesn’t address AST/CTFE mismatch
  - Zig
    - Has the comptime syntax
    - `comptime` is not a type qualifier
  - Jai
    - #run, $matching, etc.

Conclusion

- Very speculative approach at this moment
- Just a starting point
- I’m playing around with these ideas
- Lots of hard decisions to make
- Feedback is welcome
- Thank you!