The Future of D

Walter Bright Andrei Alexandrescu

Overview

Introduction

Design Objectives

- Make writing generic code easier
- Improve robustness and auditability of code
- Add support for more programming paradigms
- Facilitate large-scale development
- Lay groundwork for supporting parallel programming

- *All of this is work in progress and subject to change*

Principled Approach

- "No issue left behind"
- Leave as few dark corners as possible
- Have good engineering rationale for those that do exist (e.g. casts, constructor restrictions, non-hygiene in macros)
- Avoid the "onion in the varnish" syndrome
  - Entails breaking with some past mistakes
- Avoid the "what the f. . . heck were we thinking???" question

The Rule of Least Astonishment

- Function parameter definitions same as data definitions
- Enum lookup rules same as function pointer lookup rules
- Reading polysemous values same as function pointer lookup rules
- Macro argument pattern matching rules same as template argument pattern matching rules
- User-defined conversion rules same as built-in conversion rules

Features for Modular Development

Functions & Templates Overloading

The functions:

```
void foo(int i);
void foo(T)(T* t);
```

should overload against each other.

- Rationale: Seamless co-operation between generic and specialized code
- Unlike C++, the function is not preferred over the template.
- Functions will not be overloadable against macros.

Uniform Function Call Syntax

```
a.foo(args...)
foo(a, args...)
```

will now be interchangeable.

- Rationale: difficulty of adding member functions to a library class.
- Some people consider member calls superior to, or aesthetically nicer than, free calls
- $\Rightarrow$ 'kitchen sink' syndrome of the class designer bloating up a class with every member function she can think of.
- For a class like a matrix class, this can be a lot

Uniform Function Call Syntax (2)

- Instead, a class designer can just implement the minimally useful set of member functions.
- Then, other people can define modules that, using free functions, add specific sets of functionality.

```
import matrix;    // core functionality
import eigens;    // add eigenvector package
...
```

- To the user the eigenvector 'member' functions will act as if they were part of the original matrix class
- The add-on free functions can't be polymorphic and don't enjoy special access privileges.

Uniform Function Call Syntax (3)

- Another nice benefit: things like `int`s, etc., which normally have no member functions, can be used as if they did have member functions, which helps in writing generic code.
- Scott Meyers has eloquently elaborated on the idea of minimizing the number of member functions in his article "How Non-Member Functions Improve Encapsulation," DDJ Feb 2000 (`http://www.ddj.com/cpp/184401197`)

Function Overload Sets

- An overload set is a group of functions that are overloaded against each other.

```
// module A
void foo(int);
void foo(long);
```

- Suppose there is a module B, that has `foo()` too, but for entirely unrelated parameters:

```
// module B
class X { ... };
void foo(X);
```

Function Overload Sets (2)

And we import A and B in a third module:

```
// module C
import A;
import B;

class X x { ... };
...
foo(x);
foo(3);
```

Function Overload Sets (3)

- Two distinct overload sets of `foo()`, one set from module `A`, and one from module `B`
- Currently, it is an error to call them without explicit disambiguation (`A.foo` or `B.foo`)
- Change: allow overloading across overload sets, as long as (and this is critical) there is no overlap between the sets
- This applies to other cases where more than one overload set is possible, such as with template mixins

Function Overload Sets (4)

- If there's a match in `A` and any, even a worse, match in `B`, it will be an error
- A call to a function can only match in one of the overload sets
- In other words, the overload sets must be disjoint

- Rationale: Never allow a module hijack calls to functions of another module when `imported`

nothrow

- Exception specifications: bad idea
- Good case for allowing `nothrow` to mean that a function never throws an exception
- This means it neither generates an exception, nor propagates one from a function it calls (statically checked)
- A `nothrow` function is useful for those making their code exception safe by establishing an enforceable contract
- Additionally, enables exception frame optimizations
  - Opportunity forwent by C++ because it does not statically check `throw()` specifications

Pure Functions

- Do not read any non-const global data
- Do not write any global data
- Do not modify anything reachable through their arguments
- May modify their arguments and own stack variables

- Function result depends only on its arguments' values

Advantages of Pure Functions

- This means that pure functions:
  - Can undergo common subexpression elimination
  - Can be memoized
  - Can be tabulated
  - Can be reordered and 'scheduled' by the compiler
  - Can be automatically parallelized

Object Model Improvements

Struct ctor/dtor/opCopy/opAssign

- Rationale: enable the creation of reference counting template wrappers for objects.
- Copying value objects:
  1. member-wise copy
  2. Call the opCopy method (if supplied)
- The opCopy can e.g. increment a reference count.
- The twist to opCopy is it won't be called if the copy being made is a transfer (last read of the source)

opImplicitCastTo

```
struct S {
    int opImplicitCastTo() { ... }
    float opImplicitCastTo() { ... }
}
```

allows instances of `S` to be implicitly cast to `int`s or `float`s.

- Such casts obey the same exact rules as the built-in conversions
- Avoid dichotomy between built-in magic and user-defined abracadabra
- Helps expression templates

opImplicitCastFrom

```
struct F { ... }
struct S {
   S opImplicitCastFrom(F f) { ... }
}
```

- The two casts allow implementing types indistinguishable from built-in types
- Useful for adjusting calling convention:

```
void Fun(int[] s); // by reference
void Gun(Value!(int[]) s); // by value
```

Polysemous Values

- Quiz: What's the type of `x`?

```
int i;
uint u;
auto x = i + u;
```

- Currently, it is `uint` (following C's precedent)
- Choice of type is fundamentally arbitrary

Polysemous Values (2)

- Idea: have such expressions be typed as `(u)int`, meaning either `int` or `uint`
- It's like overloaded function pointers
- If a `(u)int` is used in a context where sign doesn't matter, it compiles without error.
- If it is used in a context where sign does matter, such as comparisons, then an error is issued.

Polysemous Values (3)

- Generalization to all types defining > 1 `opImplicitCastTo` functions
- Hash lookup result
  - Is it a value?
  - Is it a reference?
- Other sparse arrays
- Function results (polysemous: result type or error type)
- String literals (`"abc"` is really polysemous)

- Sound solution to a nasty problem

Arrays and Slices

- Problem: not known whether a slice is a “genuine” array or a window into another array
- Separate array type from slice type
  - `T[]` is a slice
  - `T[new]` is an array
- Slices are supertypes of arrays, so most existing code still works
- Arrays are extensible, slices are not
- Clarifies what functions/interfaces expect and offer

`struct` “inheritance”

- `structs` don't inherit from `interfaces`
- Or can they?
- An interface is a list of functions and function signatures
- If a `struct` can 'inherit' from an `interface`, it must provide an implementation of those functions
- This can be handy in writing template specializations—sort of like C++ concepts.

`struct` “inheritance”

```
interface Addable { void opAdd(int i); }
struct S : Addable {
    void opAdd(int i) { }}
```

- However, it’s not true subtyping—you can’t cast `S` to `Addable`

Immutability

- Needed for modular development and parallelism
- Three basic flavors:
  - Shallow constant a la C++: `final`
  - "Tail" is constant: `const`
  - "Tail" is world-immutable: `invariant`
- `final` can be combined with `const` or `invariant`
- Hard to typecheck constructors for invariant objects
  - Might need to recourse to a notion of `pure` values
- Area of very active design

Simplifying Code

Static Function Parameters

- A static function parameter:

```
void foo(static int i, int j) { ... }
```

is equivalent to creating a function template with a value parameter:

```
void foo(int i)(int j) { ... }
```

- Rationale: This is syntactic sugar to make templates even easier for mortals to write and use

Order of function argument evaluation

- Function arguments should be evaluated left to right, regardless of the order in which they are pushed on the stack.
- Payoff: improved source portability
- Cost: occasional creation of temporaries; minimal

- Put an end to a long debate over an anachronic feature

Overload On Compile-Time Values

- Static parameters are preferred when overloading:

```
int foo(int x, int y) { ... } // (1)
int foo(static int x, int y) { ... } // (2)
foo(getchar(), y); // calls (1)
foo(3, y); // calls (2)
```

- This enables:
  - optimizations based on constant folding
  - avoiding passing the 3, as it would be like a template specialized on the 3:

```
int foo(int x)(int y) { ... }
```

Enum Member Lookup Rules

- Now: Compulsively require the enum's name before the value: `File.READ`
- Planned: Treat standalone `enum` names like overloaded function names constrained by their type

```
enum OpenMode { READ, WRITE, READ_WRITE }
class File {
    this(string name, OpenMode mode = READ) {
        ...
    }
}
final f = new File("/bin/laden", WRITE);
```

String Literals

- The current methods for creating string literals turns out to be too restrictive
- 3 new syntaxes of string literals are proposed
- Delimited Strings:

```
q"(foo(xxx))"   // "foo(xxx)"
q"[foo(]"   // "foo("
q"/foo]/"   // "foo]"
```

String Literals

- D Code Strings:

```
q{foo}   // "foo"
q{/*}*/ }   // "/*?*/ "
q{ foo(q{hello}); }   // " foo(q{hello}); "
q{ 67QQ }   // error, not a valid D token
```

String Literals

- Heredoc Strings:

```
writefln(q"EOS
This
is a multi-line
heredoc string
EOS
);
```

`return` Storage Class

- In C++, a function that returns its argument is often written twice, once for `const` and once for non-`const`
- The bodies of the function are identical, just the return types change.
- The `return` storage class enables just one function to be written (and one in the object code):

```
const(char[])
capitalize(return const(char[]) s) {
    ...
}
```

`return` Storage Class

- The return type's `const`ness depends on the `const`ness of the argument to `s` (not the parameter)
- The `const char[]` return type is for static type checking inside the function
- The astute observer would say, why not do this with a template function?
- You can, but template functions cannot be virtual
  - Nyuk-nyuk-nyuk...

Analyze `this`...no, better `alias` it

`alias this` allows 'importing' the fields of a member into the name space of a `struct`:

```
struct M { int a; }
struct S {
    M m;
    alias m this;
    int b;
}
S s;
s.a;   // refers to s.m.a
s.b;   // refers to s.b
```

`switch`: The `final` Frontier

- `final switch` ensures at compile time that there is a `case` statement for every possible value of the switch expression

```
enum E { A, B, C }
E e;
...
final switch (e) {
    case A: ...
    case B: ...
} // error, no case C
```

- This helps a lot when adding members to an `enum`

`switch`: The `final` Frontier (2)

- It will also help with things like:

```
final switch (x & 3) {
    case 0: ...
    case 1: ...
    case 3: ...
} // error, no case 2
```

- A `default` label automatically makes a `final switch` vacuously correct

Generic Programming

Compile-Time Reflection

- Will be expanded as needed.
- Run-time reflection can then be done in a library based on compile-time reflection.
- Applicable to a variety of situations:
  - Marshaling/unmarshaling objects
  - Automated interface implementation
  - Parallel hierarchies

AST Macros

- AST macros manipulate abstract syntax trees in an analogous way that text macros manipulate text.
- They look like this:

```
macro foo(e) { e = 3; }
```

AST Macros (2)

- Invoking with:

```
foo(*q);
```

will replace `foo(*q)` with:

```
*q = 3;
```

- Macro parameters can have default values:

```
macro foo(e = 3) {
    ...
}
```

Pattern Matching in Macros

Macro arguments can be pattern matched analogously to template type parameters:

```
macro foo(e) { } // (1)
macro foo(e : e+1) { } // (2)
foo(3);   // matches (1)
foo(3+1);   // matches (2), e aliased to 3
foo(x+y+1);   // matches (2)
   // e gets aliased to (x+y)
foo(1+x);   // matches (1)
   // (matching predates semantic analysis)
foo(x+(3-2)); // matches (1)
   // (matching predates constant folding)
```

Macro Overloading

- Templates are pattern-matched on the shape of their argument types
- Macros are pattern-matched on the shape of their argument syntax

Hygiene

- Hygienic behavior:
  - Names after expansion are looked up in the context of the macro definition
  - Names defined by the macro invisible outside

- Too much hygiene not fun
  - Expressions preceded by a $ are looked up in the context of the macro invocation.

Dirt

```
int x = 3;
macro foo(e) {
    writeln(e, x, $x, $(x + x));
}
void test() {
    int x = 4;
    foo(7);   // prints "7348"
}
```

Dirt

- “Dirt” very useful for things like interpolated strings:

```
macro say(s) { ... }
auto a = 3, b = "Pi", c = 3.14;
say("$a musketeers computed $b to be $c\n");
```

- `say` must be able to see its caller’s symbol table
- (Do not get confused by the use of `$` in the format string as well)

static foreach

- Make unrolled loops easy to write

```
static final branches = 4;
double temp[branches], result = 0;
for (size_t i = 0; i != n; i += branches) {
   static foreach (j ; 0 .. branches) {
      temp[j] = a[i + j] * b[i + j];
   }
   static foreach (j ; 0 .. branches) {
      result += temp[j];
   }
}
```

Standard Library

DTL

- Plans to build a basic containers and algorithms library
- Serve higher-level libraries
- Modeled after STL, probably with reference semantics
  - On-demand value semantics with `std.Value!(T)`
- Containers & algos compatible with built-in arrays
- Implementation contingent on implementation of proper overloading rules

Conclusion

Acknowledgments

The following people have contributed ideas, text, code examples, perspective, etc., to this:
Andrei Alexandrescu
David Held
Bartosz Milewski
Eric Niebler
Brad Roberts
. . . and most of all, the **D** community!

The Future of **D** Is Bright