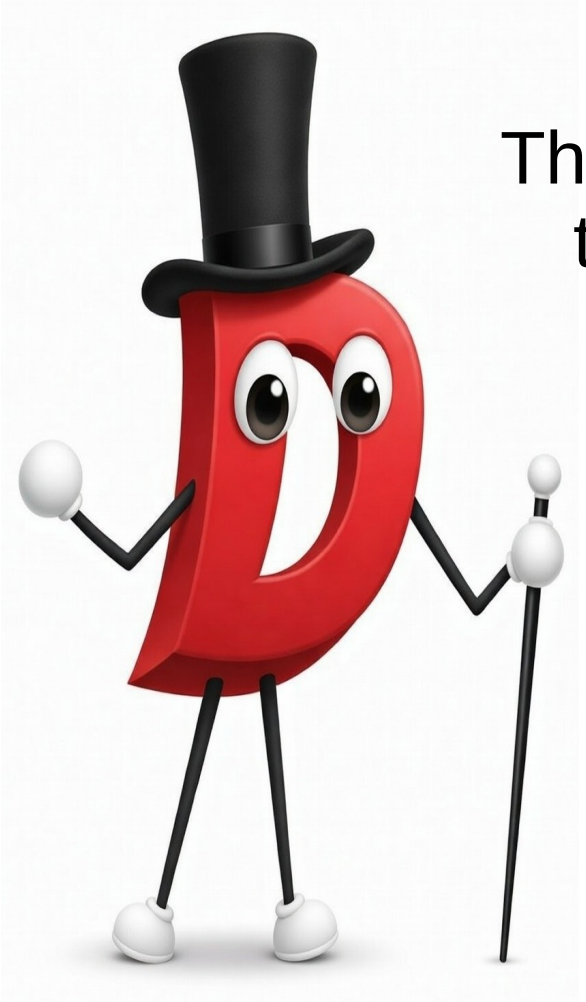
Elegant D



-or-

There's more to programming
than getting code to work

by Walter Bright
Dlang.org
November 2026
https://x.com/WalterBright

What Is Elegance In A Programming Language?


DELTA
N713TW

Characteristics

- Predictable, consistent, orthogonal, powerful
- No need to fight the language
- Minimal “noise” in the code
- Code does what it looks like it does
- Code looks like a specification of the problem rather than messy guts
- Code should just look good – frame it and put it on the mantel!

Yes You Can Get The Job Done

- With C
- C++
- Rust
- Zig
- Etc...

But Then There's D

- A re-imagining of C
- Looks / behaves a lot like C
- Unequaled metaprogramming
- Modules that work
- The preprocessor will never darken your code again
- Just as performant as C,C++,...

D Also Says No

- No cupholder for your Ferrari
  - but you can buy one for $3,533!
- No macros
  - I could spend the whole presentation ranting about macros!
- No general operator overloading
  - allowed for arithmetic purposes
- No user-defined operators
  - Such as (a ^% b)

Simple Features Replace Complex Ones

- String mixins
- Compile Time Function Evaluation (CTFE)
- Templates

Has D Succeeded In All This?

Nothing is perfect, but we keep making it better!

Simpler Declarations

- C: typedef struct S { … } S;
- D: struct S { … }



Incomplete Abstraction

```
typedef int* T;
int x;
#define Y x
```

```
alias T = int*;
int x;
alias Y = x;
```

Code Should Flow Like A Newspaper

- Left to right
- Top to bottom



Left to Right

```
int a();
int b(int);

int insideOut() => b(a);
int elegant() => a.b;
```

If You're Not Convinced Yet

g(f(e(d(c(b(a))),3))))

UFCS To The Rescue

a.b.c.d(3).e.f.g;

Bottom Up C/C++ vs Top Down D

```
static int leaf()
{
    return 25;
}

int math()
{
    return leaf() + 1;
}
```

```
int math()
{
    return leaf() + 1;
}

private int leaf()
{
    return 25;
}
```

Debug Code

```
#ifdef DEBUG
    printf(“xyzzy\n”);
#endif
```

```
debug printf(“xyzzy\n”);
```

Integer Types

wchar_t
wint_t
char16_t
char32_t
char
signed char
unsigned char
short
signed short
unsigned short
int
signed int
unsigned int
long
signed long
unsigned long
long long
signed long long
unsigned long long

intN_t
uintN_t
int_leastN_t
uint_leastN_t
int_fastN_t
uint_fastN_t
intptr_t
uintptr_t
intmax_t
uintmax_t
size_t
ptrdiff_t

byte
ubyte
short
ushort
int
uint
long
ulong
char
wchar
dchar
size_t
ptrdiff_t

Floating Point

__complex long double

creal

Cruft



typedef struct S
{ ... } S;

struct S { ... }

Manifest Constants

- C: #define E 23
- C++: const int E = 23;
- C#: static class Constants { public const int E = 23; }
- D: enum E = 23;

Initialize by Default

```
int x;          // initialize to zero
int y = void; // leave uninitialized
```

C++ Proposal for Non-Initialization

- int x = noinit;
- int x = std::uninitialized
- int x [[indeterminate]];
  - uninitialized
  - noinit
  - for_overwrite
  - no_trivial_initialization
  - not_zeroed
  - possible_security_hole
  - assume_indeterminate
  - indeterminate

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2795r5.html#optout-alts

Should Have Just Checked with D!

Lexing, Parsing and Semantic Analysis are Distinct in D

- int x = (T)(3); // cast or function call?
- A < B < C >> D // templates or operators?
- int x = cast(T)(3); // cast
- A!(B!C) D // templates

Getters, Setters, Oh My

```
struct S
{
    int a;
    int f();
    void f(int i);
};
```

```
void test()
{
    S s;
    int i = s.a;
    int j = s.f();
    s.f(i);

    S *ps;
    int k = ps->a;
    int l = ps->f();
    ps->f(k);
}
```

Easy Refactoring

```
struct S
{
    int a;
    int f();
    void f(int i);
}
```



```
void test()
{
    S s;
    int i = s.a;
    int j = s.f;
    s.f = i;

    S* ps;
    int k = ps.a;
    int l = ps.f;
    ps.f = k;
}
```

Writing To Stdout

C++

```
std::cout << "Hello, " << name << "! You are " << age <<
" years old." << std::endl;
```

D

```
writefln("Hello %s! You are %s years old.", name, age);
```

Concatenating Strings

```
char *concat(char *s1, char *s2)
{
    size_t len1 = strlen(s1);
    size_t len2 = strlen(s2);
    char *s = (char*)malloc(len1 + len2 + 1);
    assert(s);
    strcpy(s, s1);
    strcat(s, s2);
    return s;
}
```

```
string concat(string s1, string s2) => s1 ~ s2;
```

With Manual Memory Management

```
string concat(string s1, string s2)
{
   char* s = cast(char*)malloc(s1.length + s2.length);
   assert(s);
   s[ ] = s1[ ];
   s[s1.length .. s1.length + s2.length] = s2[ ];
   return s;
}
```

Report from the C Trenches

I had to deal with cleaning up string code for compliance with security standards. Every module has its own way of working with strings; some have fancy buffers with custom functions for building strings (and each gratuitously incompatible with the others), others abuse snprintf and strncat all over the place (hidden O(n^2) costs, anyone?), and yet others outright strcat and sprintf (yes, the unsafe variants!) to a buffer whose size is never checked that's passed around as a bare char*. (It took some digging to discover that everyone uses the same underlying 64k buffer size. Still, extremely scary.) – H.S. Teoh

Macros for Basic Necessities

```
#define ArrayCount(a) (sizeof(a) / sizeof((a)[0]))

typedef uint32_t U32;
typedef uint64_t U64;

global U64 max_U64 = 0xffffffffffffffffull;

#define DeferLoop(begin, end)       for(int _i_ = ((begin), 0); !_i_; _i_ += 1, (end))
#define DeferLoopChecked(begin, end) for(int _i_ = 2 * !(begin);
(_i_ == 2 ? ((end), 0) : !_i_); _i_ += 1, (end))

#define EachIndex(it, count) (U64 it = 0; it < (count); it += 1)
#define EachElement(it, array) (U64 it = 0; it < ArrayCount(array); it += 1)
#define EachEnumVal(type, it) (type it = (type)0; it < type##_COUNT; it = (type)(it+1))
#define EachNonZeroEnumVal(type, it) (type it = (type)1; it < type##_COUNT; it = (type)(it+1))
#define EachInRange(it, range) (U64 it = (range).min; it < (range).max; it += 1)
#define EachNode(it, T, first) (T *it = first; it != 0; it = it->next)
#define EachBit(it, flags) (U64 (_i_) = (flags), it = (flags) & -(flags); (_i_) != 0; (_i_) &= ((_i_) -
1), it = (flags) & -(flags))
```

https://github.com/EpicGamesExt/raddebugger/blob/7021d6056a8058b34e7b026e728b5f44346be1a1/src/base/base_core.h#L412

Basic Language Features

- a.length
  - https://dlang.org/spec/arrays.html#comparison
- ulong.max
  - https://dlang.org/spec/property.html#numeric
- foreach
  - https://dlang.org/spec/statement.html#foreach-statement

Nested Functions

```
int test(int x)
{
    int y = 4;

    int add(int z) { return x + y + z; }

    return add(3);
}
```

Nested Functions in C

```
struct S { int *px; int y; };

int add(struct S *ps, int z)
{
    return *ps->px + ps->y + z;
}

int test(int x)
{
    struct S s;
    s.px = &x;
    s.y = 4;

    return add(&s, 3);
}
```

C++ vs D Challenge

- Construct a piece of code that creates a variable by allocating it dynamically
- Creates 100 instances of a thread
- Each increments the variable 10,000 times
- Use appropriate locks to handle race conditions

C++ by Lloyd Moore

```
#include <mutex>
#include <print>
#include <ranges>
#include <thread>
#include <vector>

// Mutex to protect the shared resource
std::mutex mtx;

// Number of increments per thread
const int INCS_PER_THREAD = 1000;

// Number of threads to run
const int THREADS = 100;

// Increments the dynamically allocated shared variable
void increment(int* shared_var, int increments) {
 for (int i = 0; i < increments; ++i) {
   std::lock_guardstd::mutex lock(mtx);
   ++(*shared_var);
 }
}
```

```
int main() {
 // Dynamically allocate memory for the shared variable
 int* shared_var = new int(0);

 std::vector<std::jthread> threads;  // Create a vector to store the thread handles

 std::println(
     "Spawning {} threads to increment heap-allocated 'count' {} times each...",
     THREADS,
     INCS_PER_THREAD );

 // Create threads to increment the shared variable
 for (int i : std::ranges::iota_view(0, THREADS)) {
     threads.emplace_back(increment, shared_var, INCS_PER_THREAD);
 }

 // Wait for threads to finish
 for (auto &t : threads) {
     t.join();
 }

 // Output the final value of the shared variable
 std::println("Expected total count: {}; Actual count: {}",
     THREADS * INCS_PER_THREAD,
     *shared_var );

 delete shared_var;  // Free the dynamically allocated memory

 return 0;
}
```

D by Bruce Carneal

```
import core.atomic : atomicOp, atomicLoad;
import std.parallelism : parallel;
import std.range : iota;
import std.stdio : writeln;

enum threadCount = 100;
enum incrementsPerThread = 10000;
enum expected = threadCount * incrementsPerThread;

void main() {
    shared(ulong)* counter = new ulong;

    // Forces 100 threads by limiting the number of work items per thread to '1'
    foreach (i; parallel(iota(0, threadCount), 1)) {
        foreach (_; 0 .. incrementsPerThread)
            atomicOp!"+="(*counter, 1);
    }

    writeln("Expected total count: ", expected, "; Actual count: ",
            atomicLoad(*counter));
}
```

Describes what is happening rather than how

```
import std.stdio;
import std.algorithm;
import std.array;

void main() {
    // Read lines from stdin, take unique lines, sort them,
    // and print to stdout
    stdin
        .byLine(KeepTerminator.yes) // Read line by line, keeping newlines
        .uniq                       // Take only the unique lines
        .map!(a => a.idup)          // Convert to mutable strings
        .array                      // Collect into an array for sorting
        .sort                       // Sort the array
        .copy(stdout.lockingTextWriter()); // Write to stdout
}
```

Array and Associative Arrays

```
void main() {
    auto array = new int[10];      // Dynamic array of 10 ints
    auto half = array[4..$].dup; // Slice of the second half
    half[ ] += 1;                          // Increment all elements by one
    auto concat = array ~ half; // Concatenate arrays

    int[string] dict = [ "Hello": 10, "World": 23 ]; // Associative array
    if (int* value = "Hello" in dict) { ... } // existence check and get
value
}
```

Generic Programming and Type Deduction

```
auto add(T)(T lhs, T rhs) {
    return lhs + rhs;
}

void main() {
    int a = 5, b = 10;
    auto result1 = add(a, b); // T is deduced to int
    auto result2 = add!float(a, b); // force T to float
}
```

Compile-Time Code Generation

```
import std.conv : to;
import std.stdio : writeln;

void main() {
    int num = to!int("123"); // String to integer
    writeln(num);
    string text = to!string(456); // Integer to string
    writeln(text);
    Int[ ] arr = to!(int[])("[1, 2, 3, 4]");// String to array
    writeln(arr);
}
```


Best Lang
CHANGE MY MIND

Feedback, Questions, Comments