Strawberries and Cream
aka
Delightful Emergent Properties of D

by Walter Bright
Dlang.org
August 2022
https://twitter.com/WalterBright

Nicer Faster Readable Fun

- Octal literals
- 0-terminated strings without allocating
- Voldemort types
- Chains to avoid memory allocation
- Avoiding return errors
- Nested functions replacing gotos

We Have The Technology

- Half floats
- Using CTFE to initializer arrays
- Using enums to generate scoped list of names
- Interfacing to D from C

Octal Literals

- 0755
- PDP-10 with 18 bit words
- Used today only for file permissions
- But they're still nice for that
  - Quick! what is 0755 in decimal?

Template Literals

```
// RX for everyone +W for owner
enum RX = octal!755;

pragma(msg, RX);     // 493 decimal

template octal(int i) {
    enum octal = convert(i);
}

int convert(int i) {
    return i ? convert(i / 10) * 8 + i % 10 : 0;
}
```

Instead of Builtin Literal Syntax

- Template literals
  - Like binary!1100_1111
- No need to extend compiler
- Users can add them as necessary

0-Terminated String Without Allocating

- Slices are length delineated, not 0 terminated
- How to call a C function that wants 0 termination
  - Without allocating memory
- Allocate the stringz on the stack!
  - But how?

```
auto toCStringThen(alias dg)(const(char)[ ] src) nothrow
{
    import dmd.common.string : SmallBuffer;

    const len = src.length + 1;
    char[512] small = void;
    auto sb = SmallBuffer!char(len, small[ ]);
    scope ptr = sb[ ];
    ptr[0 .. src.length] = src[ ];
    ptr[src.length] = '\0';
    return dg(ptr);
}
```

```
char[ ] name = … ;
int fd = name.toCStringThen!(
    (fname) => open(fname.ptr, O_RDONLY)
);
```

Voldemort Types

```
auto range(int i, int j) {
    struct Result {
        int i, j;

        bool empty() { return i == j; }
        int front() { return i; }
        void popFront() { ++i; }
    }
    return Result(i, j);
}

void main() {
    foreach (x; range(3, 6))
        printf("%d\n", x);
}
```

Prints:

3
4
5

Chains To Avoid Memory Allocation

The allocate memory way:

```
char[ ] path = "include/";
char[ ] name = "file";
char[ ] ext = ".ext";
char[ ] filename = path ~ name ~ ext;
```

The No-Allocate Way

```
import std.stdio;
import std.range : chain;
import std.algorithm.iteration : joiner;
import std.array : array;
import std.utf : byChar;

void main() {
  string path = "include/";
  string name = "file";
  string ext = ".ext";

  auto filename = chain(path, name, ext);
  writeln(filename); // "include/file.ext"
  string f = filename.byChar.array();
  writeln(f.length); // 17
  writeln(f);             // "include/file.ext"
}
```

https://dlang.org/phobos/std_range.html#chain

Avoiding Returning Errors

- Exceptions are expensive and complicated
- Error codes are messy and easily overlooked
- Optional return types are still ugly
- Sooo...
  - Define the error out of existence!

Searching For A Substring and it's not found

Return an empty set (e.g. a 0-length string)

The NaN Method

- Floating point representation includes a NaN (Not a Number) pattern
- Part of IEEE 754 specification
- Any operation on a NaN produces a NaN result
- Making the code free of need for error checking

NaN Variation

- Issue error message when NaN is created
- Any operation on a Nan produces a Nan result
  - D compiler uses this method when “recovering” from errors
  - Prevents meaningless cascading error messages

Replacement char Variation

- D Unicode operations tend to throw an exception when invalid Unicode is found
  - Which is often because Unicode data is messy
  - Quitting processing is undesirable
    - Like if rendering text for display
- My solution would be to define all code points, so there are no errors
- Instead, we replace invalid code points with the replacement char U+FFFD

Nested Functions Replace Gotos

```
void plan(int i) {
    switch (i)    {
        case 1:
            a();
            goto L3;
        case 2:
            goto L4;
        case 3:
            e();
        L3:
            b();
        L4:
            c();
            return;
    }
}
```

```
void plan(int I)
{
    void doc()
    {
        c();
    }

    void dobc()
    {
        b();
        doc();
    }
```

```
    switch (i)
    {
        case 1:
            a();
            return dobc();
        case 2:
            return doc();
        case 3:
            e();
            return dobc();
    }
}
```

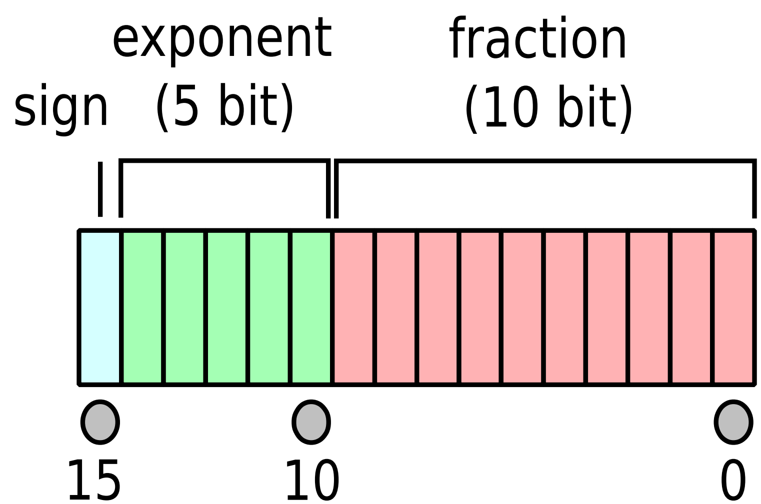
Half Floats

- A 16 bit floating point type
- 16 bits for storage economy
- Was requested as a builtin type
  - Trouble is, there are many 16 bit float formats
  - https://en.wikipedia.org/wiki/Half-precision_floating-point_format
- Is a reasonable library solution possible?

Half Float Usage

```
HalfFloat h = hf!27.2f;
HalfFloat j = cast(HalfFloat)( hf!3.5f + hf!5 );
HalfFloat f = HalfFloat(0.0f);
```

Half Float Format



https://en.wikipedia.org/wiki/Half-precision_floating-point_format

Half Float Implementation

- Store as a short
- Implicitly convert HalfFloat to float
- Explicitly convert float to HalfFloat

Half Float Code 1

```
struct HalfFloat {

    @property float toFloat() { return shortToFloat(s); }
    alias toFloat this; // implicitly convert HalfFloat to float

    /* template prevents implicit conversion
     * of argument to float.
     */
    this(T : float)(T f)  {
        static assert(is(T == float));
        s = floatToShort(f);
    }
    ushort s = EXPMASK | 1;     // .init is HalfFloat.nan
```

Half Float Code 2

```
    static @property {
    HalfFloat min_normal() { HalfFloat hf = void; hf.s = 0x0400; return hf; }
    HalfFloat max()        { HalfFloat hf = void; hf.s = 0x7BFF; return hf; }
    HalfFloat nan()      { HalfFloat hf = void; hf.s = EXPMASK | 1; return hf; }
    HalfFloat infinity()   { HalfFloat hf = void; hf.s = EXPMASK; return hf; }
    HalfFloat epsilon()    { HalfFloat hf = void; hf.s = 0x1400; return hf; }
    }

    enum dig =        3;
    enum mant_dig =   11;
    enum max_10_exp = 5;
    enum max_exp =    16;
    enum min_10_exp = -5;
    enum min_exp =    -14;

    ushort s = EXPMASK | 1;     // .init is HalfFloat.nan
}
```

Half Float Literal

```
template hf(float v)
{
    enum hf = HalfFloat(v);
}

HalfFloat h = hf!27.2f;
```

ShortToFloat() and floatToShort() implementations

- Floating point goodness:
  - Rounding
  - Guard bit
  - Sticky bit
  - Hidden bit

https://github.com/DigitalMars/sargon/blob/master/src/sargon/halffloat.d

Using CTFE to Initialize Arrays

Initialize an array of 20 squares:

```
int[20] squares = [0,1,4,9,16,25,36,49,64,81,100,
                   121,144,169,196,225,256,289,324,361,];
```

The Old Way

```
import core.stdc.stdio;

void main()
{
    enum N = 20;
    printf("module table;\n");
    printf("int[%d] squares = [", N);
    foreach (i; 0 .. N) {
        printf("%d,", i * i);
    }
    printf("];\n");
}
```

```
import table;
```

The New Way

Combines Lambdas and CTFE

```
enum N = 20;
int[N] squares = () {
    int[N] squares;
    foreach (i; 0 .. N)
        squares[i] = i * i;
    return squares;
}();
```

Using Enums to Generate Scoped List of Names

(thanks to Dennis Korpel)

```
struct S
{
    bool square : 1,
         circle : 1,
         triangle : 1;
}
```

Result We Want

```
struct S
{
    enum Flags { Square = 1, Circle = 2, Triangle = 4 }

    bool square()   { return !!(flags & Flags.Square); }
    bool circle()   { return !!(flags & Flags.Circle); }
    bool triangle() { return !!(flags & Flags.Triangle); }

    bool square(bool b) { b ? (flags |= Flags.Square)
                            : (flags &= ~Flags.Square); return b; }
    bool circle(bool b) { b ? (flags |= Flags.Circle)
                            : (flags &= ~Flags.Circle); return b; }
    bool triangle(bool b) { b ? (flags |= Flags.Triangle)
                            : (flags &= ~Flags.Triangle); return b; }

    private ubyte flags;
}
```

Would Rather Write

```
void main()
{
    enum F { square, circle, triangle }

    static struct S
    {
        mixin(generateFlags!(F, ubyte));
    }
    S s;
    s.square = true;
    s.circle = false;
    s.triangle = true;
    assert(s.square == true);
    assert(s.circle == false);
    assert(s.triangle == true);
}
```

```
string generateBitFlags(E, T)() {
    string result = "pure nothrow @nogc @safe final {";
    enum enumName = __traits(identifier, E);

    foreach (size_t i, mem; __traits(allMembers, E))  {
        static assert(i < T.sizeof * 8, "too many fields”);
        enum mask = "(1 << "~i.stringof~")";
        result ~= "

        bool "~mem~"() const scope { return !!(flags & "~mask~"); }

        bool "~mem~"(bool v)  {
            v ? (flags |= "~mask~") : (flags &= ~"~mask~");
            return v;
        }";
    }
    return result ~ "}\n private "~T.stringof~" flags;\n";
}
```

Imports in C

```
__import stdio;

int main()
{
    printf("hello world\n");
    return 0;
}
```

What If Imported Module is D?

```
__import stdio;
__import daction;

int main() {
    printf("D function returns %d\n", action(value)); // 10
    return 0;
}
```

```
module daction;

enum value = 7;
int action(int i) { return 3 + i; }
```

Overloaded Functions??

```
__import stdio;
__import daction;

int main()
{
    printf("D function returns %d\n", action(1.0f)); // 5
    return 0;
}
```

```
module daction;

int action(int i) { return 3; }
int action(float f) { return 5; }
```

Templates ????

```
__import stdio;
__import daction;

int main()
{
    printf("D function returns %d\n", action(1)); // 4
    return 0;
}
```

```
module daction;

int action(T)(T t) { return cast(int)t.sizeof; }
```

Importing D modules completes the circle

I.e. C can interface to D functions!

Conclusion

- The whole is more than the sum of the parts
- Capabilities can combine in unexpected ways
- Sometimes delightful discoveries are made!

AMA!

https://twitter.com/walterbright