Memory DisAllocation

Walter Bright
dconf.org

Strategies

- Manual Memory Management
- Garbage Collection
- Reference Counting

Manual Memory Management

- Efficient
- Minimal memory use
- Fastest

But There's Always a But...

- Plentiful source of errors
- Complex and time consuming
- Tends to obscure the algorithm

Garbage Collection

- Easy
- Memory safe
- Fast

But...

- 3x memory consumption
- Pauses
- Not usuable when resources are tight

Reference Counting

- Predictable
- Minimal memory use
- Memory safe

But...

- Slower
- Cycles are problematic
- Memory safety involves compromises

Problem #1

- Do I even have a problem?
- Where and how much memory is being allocated?

GC Memory Profiler

`-profile=gc`

Using it on Warp...

```
bytes allocated, type, function, file:line
 1764  id.Id id.Id.pool id.d:107
  640  context.Source[] context.Context!(LockingTextWriter).Context.push context.d:400
  336  immutable(ubyte)[][] directive.lexMacroParameters!(Lexer!(Context!(LockingTextWriter)*)).lexMacroParameters directive.d:61
  128  std.array.Appender!(string[]).Appender.Data std.array.Appender!(string[]).Appender.this array.d:2617
   80  char[] std.path.buildNormalizedPath!char.buildNormalizedPath path.d:1244
   64  closure macros.stringize!(Textbuf!(ubyte, "lex")).stringize macros.d:396
   40  closure macros.stringize!(Textbuf!(ubyte, "exp")).stringize macros.d:396
   40  immutable(ubyte)[][] directive.lexMacroParameters!(Lexer!(Context!(LockingTextWriter)*)).lexMacroParameters directive.d:86
   32  std.array.Appender!(const(wchar)[]).Appender.Data std.array.Appender!(const(wchar)[]).Appender.this array.d:2617
   28  std.getopt.Option[] std.getopt.getoptImpl!(string, string[]*).getoptImpl getopt.d:558
   28  std.getopt.Option[] std.getopt.getoptImpl!(string, bool*).getoptImpl getopt.d:558
   28  std.getopt.Option[] std.getopt.getoptImpl!(string, string[]*).getoptImpl getopt.d:558
   28  std.getopt.Option[] std.getopt.getoptImpl!(string, bool*, string, bool*).getoptImpl getopt.d:558
   28  std.getopt.Option[] std.getopt.getoptImpl!(string, string[]*).getoptImpl getopt.d:558
   28  std.getopt.Option[] std.getopt.getoptImpl!(string, string*).getoptImpl getopt.d:558
   28  std.getopt.Option[] std.getopt.getoptImpl!().getoptImpl getopt.d:617
   28  std.getopt.Option[] std.getopt.getoptImpl!(string, string[]*).getoptImpl getopt.d:558
   16  immutable(char)[][] cmdline.combineSearchPaths cmdline.d:185
   12  closure cmdline.combineSearchPaths cmdline.d:173
    8  const(char)[][] std.path.buildNormalizedPath!char.buildNormalizedPath path.d:1200
    8  immutable(char)[][] cmdline.parseCommandLine cmdline.d:116
    6  immutable(char)[] cmdline.parseCommandLine cmdline.d:116
    0  immutable(char)[][] cmdline.combineSearchPaths cmdline.d:187
```

Problem #2

I'm designing reusable code.
Which strategy should I use?

Watcha Gonna Do?

Don't Allocate Memory!

(absurd, right? Walter's really stepped in it this time!

Typical Example

```
import std.conv;
import std.stdio;

writeln(to!string(28));
```

allocates memory

Usual Implementation

```
string toString(uint u) {
  char[uint.sizeof * 3] buf;
  size_t idx = buf.length;
  do {
    buf[--idx] = (u % 10) + '0';
    u /= 10;
  } while (u);
  return buf[idx .. $].idup;
}

import std.stdio;

void main() {
    writeln(toString(28));
}
```

DisAllocation

```
auto toString(uint u) {
  static struct Result {
    this(uint u) {
      idx = buf.length;
      do {
        buf[--idx] = (u % 10) + '0';
        u /= 10;
      } while (u);
    }
    @property bool empty() { return idx == buf.length; }
    @property char front() { return buf[idx]; }
    void popFront() { ++idx; }
    char[uint.sizeof * 3] buf;
    size_t idx;
  }
  return Result(u);
}

import std.stdio;

void main() { writeln(toString(28)); }
```

Committing To Memory

```
import std.array;

string s = toString(18).array;
```

No Allocation!

- Lazy
- State is on the stack
  - (hot in the cache)

Note that the allocation decision
was at the higher level.

Concatenating Strings

```
auto s = [1,2,3] ~ [8,7,6];
```

```
import std.range;
auto chain(R1, R2)(R1 r1, R2 r2)
    if (isInputRange!R1 && isInputRange!R2 &&
        is(ElementEncodingType!R1 == ElementEncodingType!R2))
{
    static struct Result {
        this(R1 r1, R2 r2) {
            this.r1 = r1;
            this.r2 = r2;
        }
        @property bool empty() {
            return r1.empty && r2.empty;
        }
        @property auto front() {
            return r1.empty ? r2.front : r1.front;
        }
        void popFront() {
            r1.empty ? r2.popFront() : r1.popFront();
        }
      private:
        R1 r1;
        R2 r2;
    }
    return Result(r1, r2);
}
```

```d
import std.stdio;

void main() {
    writeln(chain([1,2,3], [8,7,6]));
}
```

writes:

```
[1,2,3,8,7,6]
```

Note that writeln also accepts ranges as input

Memory allocation ceases to be a decision made by low level algorithms, and instead is pushed up to the higher semantic level.

Range Checklist

- lazy
- trivial construction
- no memory allocation
- present widest possible interface
- pure nothrow @safe @nogc

pure nothrow @safe @nogc

```
pure nothrow @safe @nogc unittest
{
    immutable int[3] a = [1,2,3];
    immutable int[3] b = [4,5,6];
    auto c = chain(a[], b[]);
    int i;
    foreach (e; c)
        assert(e == ++i);
}
```

Realistically

Ranges are harder to write than
loop oriented code. But they
are much easier to use and reuse.

=> good investement

Vision

- This is the future of D
- This is where programming is going
- D can lead or follow
- We have an opportunity to lead

Call to Action

Scrutinize all APIs that accept or return arrays. Can they be generalized to be ranges instead?

'Range-ified' Phobos Functions

- std.path.baseName()
- std.path.stripDrive()
- std.string.indexOf()
- std.string.lineSplitter()
- std.string.soundexer()
- … etc ...

But Wait, There's More!

All the previous slides were about ranges that can be used today.

D ranges are based on the concept of arrays.

The core D language has special syntax for arrays...

… maybe that can be extended to ranges?

Existing Support

```
foreach (element; range) {
    … element …
}
```

very successful

Array Initialization

```
T[10] array = range;
```

Array Concatenation

```
array ~ range
```

Array Operations

```
array1[ ] = array1[ ] + range1[ ] / range2[ ];
```

Back To Memory Allocation

It's usually a low level decision.

I've almost never seen successful mixing of components using different memory management schemes. A library has to choose which camp it is in.

With ranges, which are allocation agnostic, this is far less of a problem. Reusable libraries become practical that will work with whatever scheme the user selects.

Conclusion

Ranges! Ranges! Ranges!