Allocating Memory with the D Programming Language

by Walter Bright

Many Keys to Performance

- better algorithms
- low level optimizations
- memory caching / layout
- code caching / layout
- multithreading
- and ...

Allocating Memory is Critical for Non-Trivial Programs

- D supports multiple methods
- different methods can be used for different purposes in the same program
- there is no one-size-fits-all
- dramatic differences in performance, memory consumption, ease of programming, etc.

Automatic Memory Management

Advantages

- easy
- memory safe
- faster to write code
- handles cycles

Disadvantages

- uses 3x the memory
- pause while collection runs
- indeterminate when destructor is run
- long running programs can exhaust memory due to pinning

Use For

- smaller programs (like scripts)
- parts of code that are rarely run
  - initialization
  - error handling
- batch utilities
- where dev costs are higher than compute costs

malloc / free

Advantages

- familiar, well understood
- fast
- compact
- well implemented
- interoperate with C code

Disadvantages

- no memory safety checks
- dangling pointers
- memory leaks
- double frees
- very hard to audit

Use For

- compute time is far costlier than dev time
- experienced devs are available
  - of course, can't get experience without using malloc / free
- not a disaster when the program fails
  - because experience shows it will fail

Making Your Own malloc / free

- to instrument it
- add sentinels
- application specific allocation strategy
- try your hand at making it faster

The Memory Safe malloc()

Impocerous!

… or is it?

There's a Trick to It!

- just allocate, never free!
- very fast
- use for data that lasts until program exit
- use for batch processing
  - like (cough) compilers (cough)

```
import core.stdc.stdio, core.stdc.stdlib;

static size_t heapLeft = 0;
static void* heapPtr;

void* heapAlloc(size_t nbytes) {
    static void error() {
        printf("Error: out of memory\n");
        exit(EXIT_FAILURE);
        assert(0);
    }

    // 16 byte alignment is better
    // (sometimes needed) for doubles
    const sz = (nbytes + 15) & ~15;

    // code layout is so most
    // common case is straight through
    if (sz <= heapLeft) {
     L1:
        heapLeft -= sz;
        void *p = heapPtr;
        heapPtr += sz;
        return p;
    }

    enum ChunkSize = (4096 * 16);
    if (sz > ChunkSize) {
        if (auto p = malloc(sz))
            return p;
        error();
    }

    heapLeft = ChunkSize;
    heapPtr = malloc(ChunkSize);
    if (!heapPtr)
        error();
    goto L1;
}

void heapFree(void *p) { }
```

Scope Guard

- method for hooking how a function exits, and attaches code to it
  - both normal and exception (i.e. error) exits
- related to try-catch-finally
  - in fact, the compiler converts scope guard statements to try-catch-finally

For Example

```
import core.stdc.stdlib;

auto p = cast(T*)malloc(length * T.sizeof);
assert(!length || p);
auto array = p[0 .. length];
scope (exit) free(array.ptr);
...
```

Pros and Cons

- natural, readable, convenient syntax
- resolves the dreaded “forgot to free it when exiting early”
- resolves the dreaded “forgot to free it when exceptions were thrown” (i.e. it is exception safe)
- still vulnerable to the other problems with malloc/free

RAII (aka Destructors)

The Simpsons

```
struct S(T) {
    import core.stdc.stdlib;
    this(size_t length) {
        auto p = cast(T*)malloc(length * T.sizeof);
        assert(!length || p);
        T[ ] array = p[0 .. length];
    }
    ~this() { free(array.ptr); }
    T[ ] array;
}

{
    auto s = S!int(10);
    ...
}
```

Pros / Cons

- well understood
- exception safe
- nobody ever got fired for using RAII
- still vulnerable to dangling pointers
- only workable when there's one owner
- doesn't work so well with cyclic graphs

Reference Counting

Walter Crane

```
struct S(T) {
    import core.stdc.stdlib;
    this(size_t length) {
        auto p = cast(T*)malloc(length * T.sizeof);
        assert(!length || p);
        array = p[0 .. length];
        auto pcount = cast(size_t*)malloc(size_t.sizeof);
        assert(pcount);
        *pcount = 1;
    }
    this(ref S s) {
        array = s.array;
        pcount = s.pcount;
        ++*pcount;
    }
    ~this() { if (--*pcount == 0) { free(array.ptr); free(pcount); }
    void opAssign(ref S s) {
        auto tmparray = array;
        auto tmppcount = pcount;
        array = s.array;
        pcount = s.pcount;
        ++*pcount;
        if (--*tmppcount == 0) { free(tmparray.ptr); free(tmppcount); }
    }
    T[ ] array;
    size_t* pcount;
}
```

Advantages

- no pauses
- memory reclaimed as soon as possible

Disadvantages

- cycles
- expensive due to exception handling
- can be slower than automatic memory management!

Stack

```
T[100] tmp = void;
T[ ] buffer = tmp[0 .. length];
...
```

Advantages

- pretty much free
- automatic cleanup at zero cost
- no worries about exception handling
- don't forget `= void;` initialization!

Disadvantages

- what if the buffer is too small?
  - then you've got to add error handling code
- running out of stack space
  - especially for small embedded systems or ones where you're running a zillion concurrent threads

Hybrid stack / malloc

```
debug
    enum tmplen = 2;
else
    enum tmplen = 100;

T[tmplen] tmp = void;
T[ ] buffer;
if (length <= tmp.length)
    buffer = tmp[0 .. length];
else {
    auto p = cast(T*)malloc(length * T.sizeof);
    assert(!length || p);
    buffer = p[0 .. length];
}
...
if (buffer.ptr != tmp.ptr)
    free(buffer.ptr);
```

Do some testing to pick good value for tmplen so malloc is rarely hit in practice.

Of course, for debugging use a small value so the malloc path gets tested properly.

I use this technique a lot. It's fast and effective.

Hybrid Stack / malloc with Voldemort Types

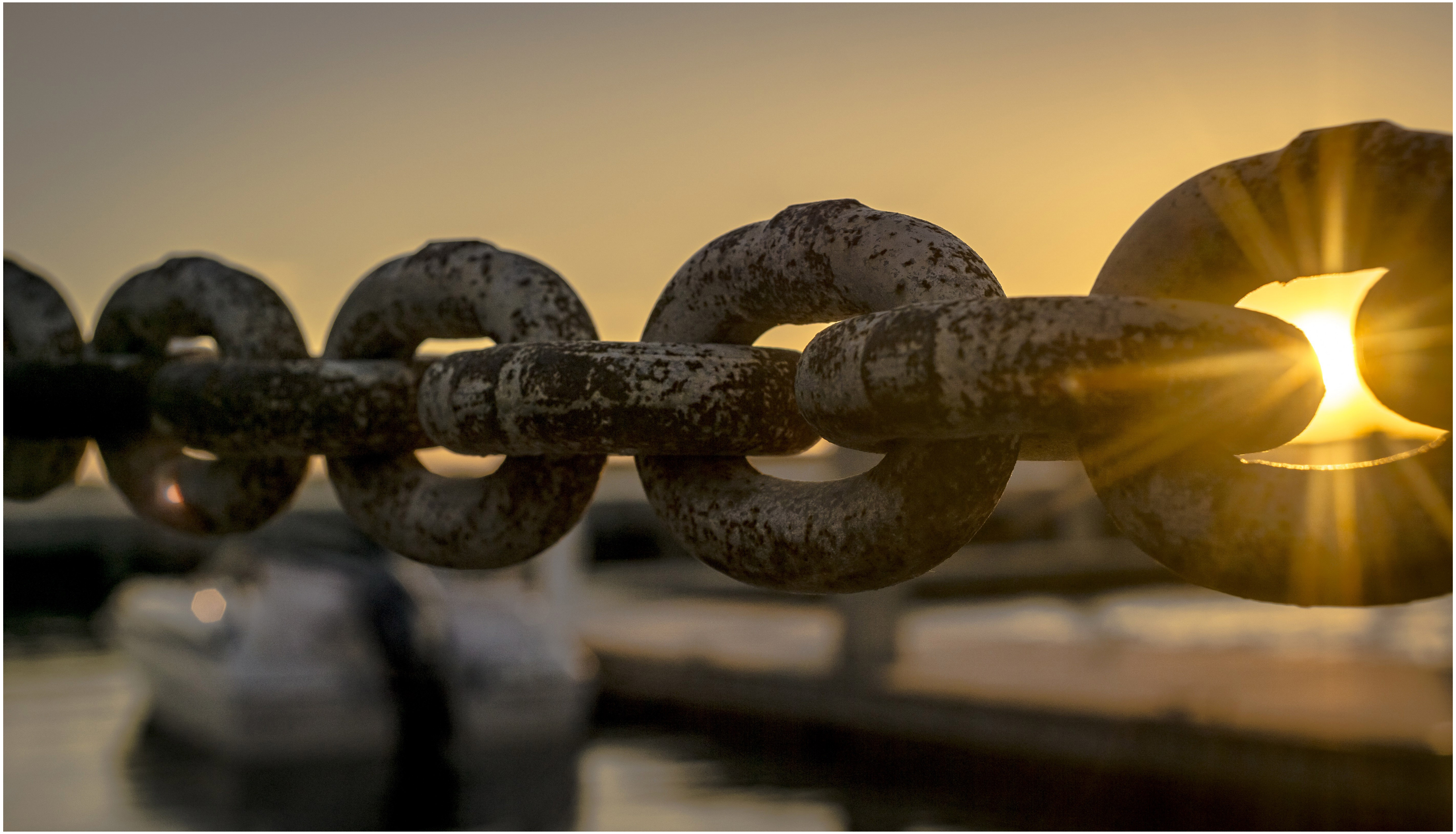
<Insert Picture Here>

```
auto tmpBuf(T)(size_t length) {
    static struct Result {
        void initialize(size_t length) {
            if (length <= tmp.length)
                buffer = tmp[0 .. length];
            else {
                auto p = cast(T*)malloc(length * T.sizeof);
                assert(!length || p);
                buffer = p[0 .. length];
            }
        }
        ~this() { if (buffer.ptr != tmp.ptr) free(buffer.ptr); }
        T[ ] buffer = void;
        T[100] tmp = void;
    }
    Result result = void;
    result.initialize(length);
    return result;
}

…
auto buffer = tmpBuf!T(length);
```

- Nicely encapsulates the allocation and cleanup
- Still uses the stack of the caller!
  - through the magic of the Named Return Value optimization

No-Allocation Allocations using Chain



- no allocation at all
- done with ranges and slices
- can be very efficient
- predictable

```d
auto chain(string s1, string s2) {
  struct Chain {
    string s1, s2;
    bool empty() {
      return s1.length == 0 &&
                s2.length == 0;
    }
    char front() {
      return s1.length
         ? s1[0]
          : s2[0];
    }
    void popFront() {
      if (s1.length)
        s1 = s1[1 .. $];
      else
        s2 = s2[1 .. $];
    }
  }
  return Chain(s1, s2);
}
```

```d
import core.stdc.stdio;

int main() {
  auto r = chain("hello", " betty");
  foreach (c; r)
    printf("%c", c);
  printf("\n");
  return 0;
}
```

- same caveats as slices
- watch lifetimes of s1 and s2
- https://dlang.org/phobos/std_range.html#chain
  - for more general implementation

Summary

- automatic memory management
- malloc / free
- memory safe malloc
- scope guard
- RAII
- reference counting
- stack, hybrid stack and Voldemort hybrid stack
- chaining