C object storage

Based on “Modern C”, 2nd edition, chapter 13
by Jens Gustedt

Ivan Krylov, 2023

Storage duration

- “Static”
  - Lifetime spans entire execution of program
  - More complicated for thread-local variables
- “Automatic”
  - Lifetime tied to block where declared
  - Slightly different for variable-length arrays
- “Allocated”

Storage-class specifiers

```c
// A.c
extern int common;
int common = 0;

static int private = -1;

static void foo(void) {
    extern int also_common;
    static int private = 1;
}

void bar(int arg) {
    static int private = 2;
    auto int local = 3;
    int also_local = 4;
    if (arg > 0) bar(arg - 1);
}
```

```c
// B.c
extern int common;
extern int also_common;
int also_common = 1;

static int private = -2;

extern void bar(int arg);

static void foo(void) {
    static int private = 2;
    register int local_no_alias = 5;
    bar(42);
}
```

- **static** storage duration
  - Visibility restricted to enclosing block/file
- **extern**: static storage duration
  - global visibility
  - Implied outside functions
- **auto**matic storage duration
  - Implied inside functions
  - **register**: forbidden address-of

Exercise 18.2.2

- Which storage class…
  - … is applicable for variables shared by several files?
    - `extern`
  - …is applicable for variables shared by several functions in one file?
    - `static`
  - Changes the storage duration of a variable?
    - `extern` and `static` for a local variable

Exercise 18.2.4

```
int f(int i) {
    static int j = 0;
    return i * j++;
}
```

Number of previous runs of the function

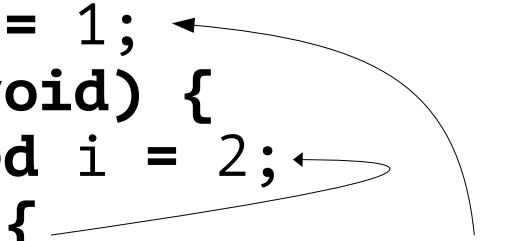
```
f(10);
for (int i = 0; i < 4; ++i) f(0);
f(10);
```

→ 10 * 0 → 0

→ 10 * 5 → 50

Exs 33: shadowing declarations

```
#include <stdio.h>
unsigned i = 1;
int main (void) {
    unsigned i = 2;
    if (i) {
        extern unsigned i;
        printf("%u\n", i);
    } else {
        printf("%u\n", i);
    }
}
```



Branch taken becase 2 is true

Refers to external-linkage, file-scope object

Note: TinyCC prints 2 instead of 1, but that's probably a mistake

VLA lifetime

- Automatic storage duration spans whole scope
- VLA storage duration is less than automatic: definition must be evaluated

```
int size = 0x13;
{
    int * pointer = 0;
  AGAIN:
        'automatic' exists, addressable via 'pointer'
    int automatic = 42, VLA[size];
          'VLA' exists from here to end of block
    if (...) {
        size = 0x37;
        pointer = &automatic;
        goto AGAIN;
    }           'VLA' will reappear with new
                size after definition is evaluated
}
```

Initialization

- Objects with static storage duration are always initialised (to zero if not specified)
- Objects with automatic and allocated storage duration must be initialised manually
  - Best to write initialisation functions for custom types: `foo * foo_init(foo *, ...);`

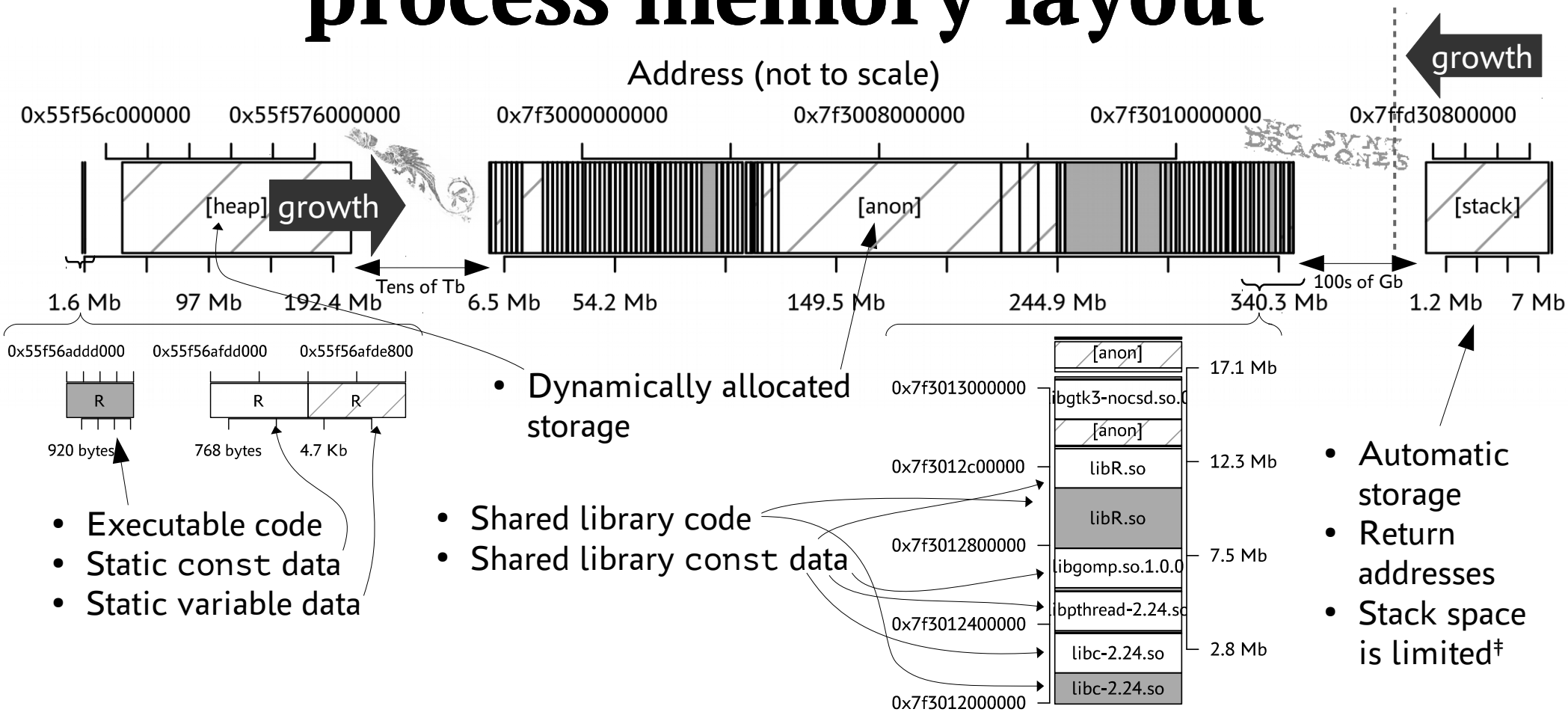
C dynamic allocation API

```c
#include <stdlib.h>
// Allocate `size` bytes
void * malloc(size_t size);
// Allocate `nmemb * size` bytes initialised to 0
void * calloc(size_t nmemb, size_t size);
// Allocate `size` bytes aligned to `alignment` bytes
void * aligned_alloc(size_t alignment, size_t size);
// Change existing allocation to store `size`
void * realloc(void * ptr, size_t size);
// Deallocate the buffer
void free(void * ptr);
```

May invalidate `ptr`

See also: Package eaf vs GCC bug; optimisers vs. math; pointer provenance

Implementation details: process memory layout



- Executable code
- Static `const` data
- Static variable data

- Dynamically allocated storage

- Shared library code
- Shared library `const` data

- Automatic storage
- Return addresses
- Stack space is limited‡

Images taken from https://en.wikipedia.org/wiki/Here_be_dragons

R dynamic allocation API

- `SEXP Rf_alloc<...>(<...>);`
  `SEXP PROTECT(SEXP);`
  `void UNPROTECT(int);`
  `void PROTECT_WITH_INDEX(SEXP, PROTECT_INDEX *);`
  `void REPROTECT(SEXP, PROTECT_INDEX);`

A family of allocating functions

Anything not protected *may* be freed on the next R API call

- `void* vmaxget(void);`
  `char* R_alloc(R_SIZE_T, int);`
  `void vmaxset(const void *);`

Call **vmaxset**() to reset allocation stack to a previous value (done by R at the end of **.C**() and friends). Used by e.g. **translateChar**().

- *type*`* R_Calloc(size_t, `*type*`);`
  type`* R_Realloc(`*type*`*, size_t, `*type*`);`
  `void R_Free(void*);`

error() instead of null pointer

Must deallocate manually (on `.exit`?)

See also: "Unprotecting by value" by Tomas Kalibera, The R Blog

Exs 29: large malloc()

- A `double` for every person in the world: how much memory?
  - `length` ≈ $8 \cdot 10^9$ people in the world
  - `sizeof(double)` = 8 bytes per person
- `size_t` multiplication may silently overflow
  - 64-bit: 2^{33} (>8 billion) of 2^{32} (4 GiB)-sized entries
  - Use `calloc()`
- Allocations may succeed anyway‡

Exercise 17.2.2: duplicate a string

```
char * duplicate(const char * src) {
    size_t len = strlen(src);

    char * ret = malloc(len+1);
    if (!ret) return ret; // or NULL

    memcpy(ret, src, len+1);
    return ret;
}
```

Available as non-standard `strdup()` on some systems.

`strlen()+1` should be unable to overflow because the terminator is a part of the string

Exercise 17.3.3: create an array†

Avoid using `int` for sizes that are not guaranteed to be small and nonnegative.

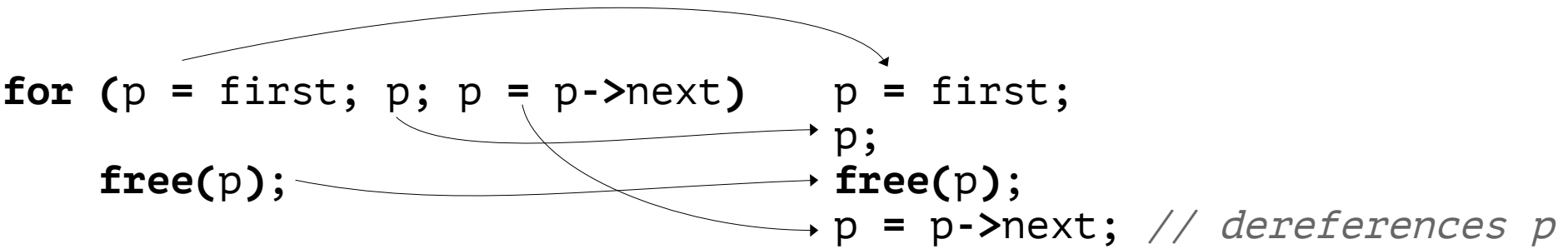
A `size_t` overflow will silently allocate a shorter buffer.

```
int * create_array(size_t n, int initial_value) {
    // n * sizeof(int) may overflow
    if (SIZE_MAX / sizeof(int) < n) return NULL;

    int * ret = malloc(sizeof(int[n])); // or: sizeof(int) * n
    if (!ret) return ret;

    for (size_t i = 0; i < n; ++i) ret[i] = initial_value;

    return ret;
}
```

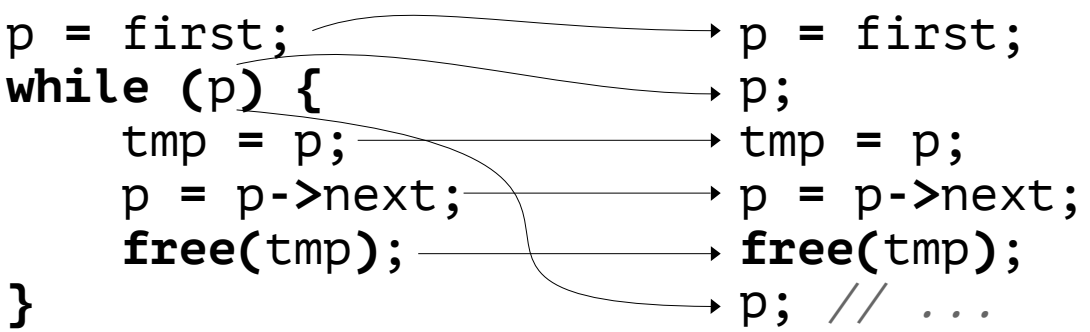

E.g. crash in tools::parseLatex due to array size overflow

Exercise 17.5.7: use-after-free

What's wrong with this code?



This works:



Project 17.1: dynamic inventory†

```
#define MIN_PARTS 10

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} *inventory = NULL;

size_t num_parts = 0, allocated_parts = 0;


if (num_parts == allocated_parts) {
  size_t newsz = allocated_parts ? allocated_parts * 2 : MIN_PARTS;
  struct part * newinv = realloc(inventory, newsz * sizeof(*inventory));
  if (!newinv) {
    printf("Database is full; can't add more parts.\n");
    return;
  }
  allocated_parts = newsz;
  inventory = newinv;
}
```

Project 17.2: sorted inventory

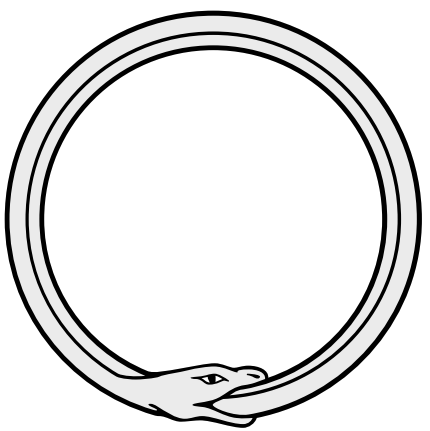
```
static int invcmp(const void *a, const void *b) {
  int anum = ((struct part *)a)->number,
      bnum = ((struct part *)b)->number;
  return anum < bnum ? -1 : anum > bnum ? 1 : 0;
}
```

```
qsort(inventory, num_parts, sizeof *inventory, invcmp);
```

Interlude: circular buffers

- Producer-consumer problem
- FIFO queue
- Used in sound, networking

```
struct circular {
  size_t start;
  size_t len;
  size_t max_len;
  double * tab;
};
```



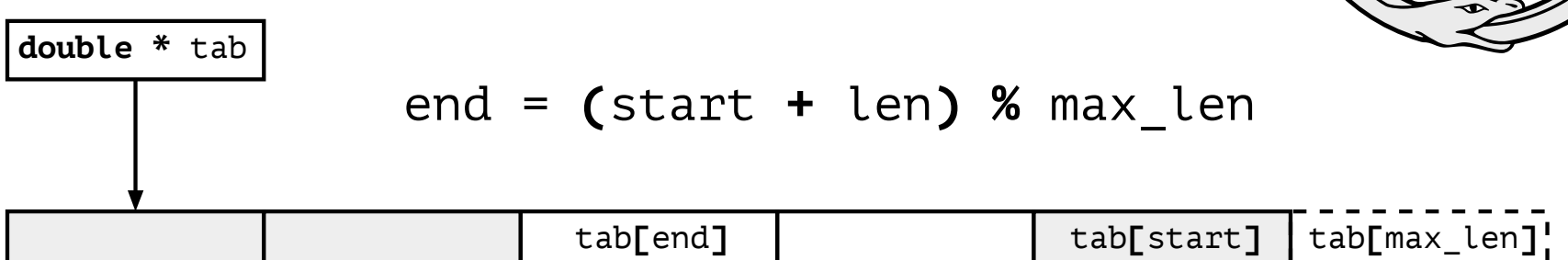


Image taken from https://commons.wikimedia.org/wiki/File:Ouroboros-simple.svg

Exs 30: lifecycle of `circular*`

```
circular* circular_new(size_t len) {
    return circular_init(
        malloc(sizeof(circular)), len
    );
}

size_t circular_getlength(circular* c) {
    return c->len;
}

void circular_delete(circular* c) {
    if (!c) return;
    free(c->tab);
    free(c);
}
```

Caller can't manage memory for incomplete type `circular`

Need accessors for every useful bit of information

Like `free()`, accept null pointers when disposing of the object

Exs 30: operations on `circular*`

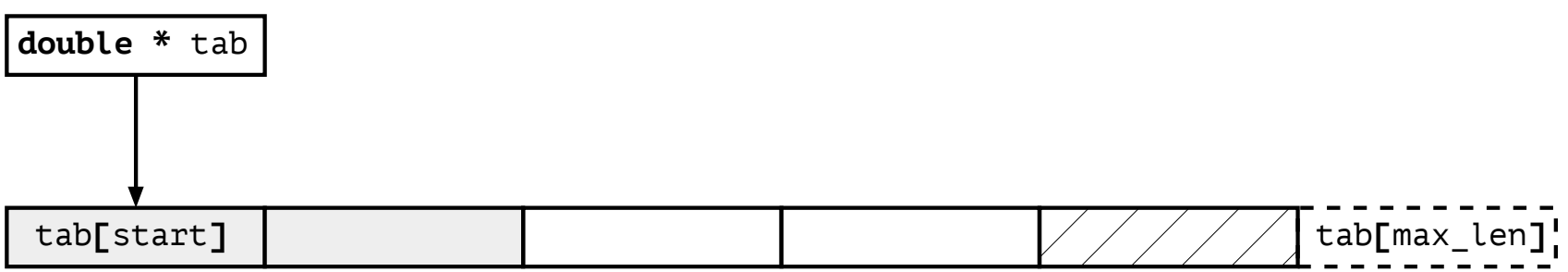
```
circular* circular_append(circular* c, double value) {
    // the element at c->len is just past the end
    double * new = circular_element(c, c->len);
    if (!new) return 0;

    *new = value;
    ++c->len;
    return c;
}
```

Exs 30: operations on `circular*`

```
double circular_pop(circular* c) {
    double ret = NAN;
    if (!c->len) return ret;

    ret = *circular_element(c, 0);
    c->start += 1;
    c->start %= c->max_len;
    --c->len;

    return ret;
}
```

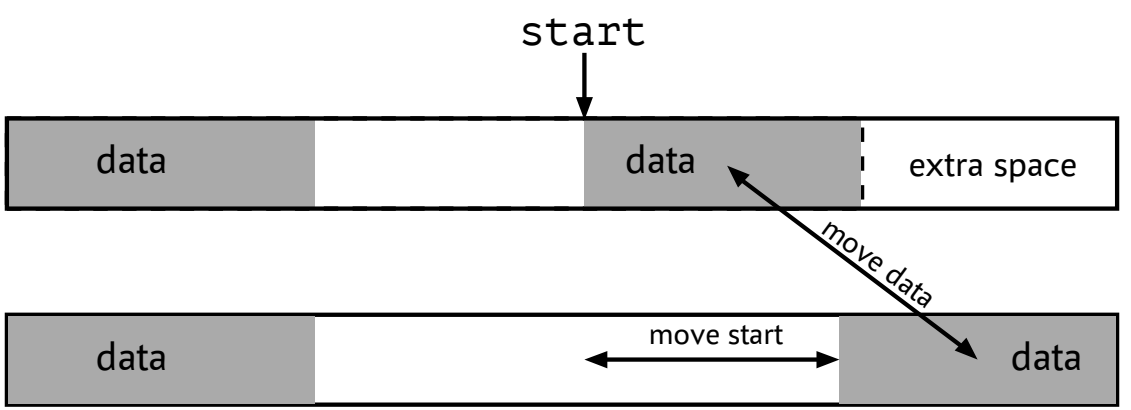
No good way to say “no more elements” while returning a `double`

`start < max_len` is an invariant



Exs 31: enlarging circular_resize

```
if (
  max_len > c->max_len
  && c->len
  && circular_getpos(c, c->len-1) < c->start
) {
  memmove(
    c->tab + max_len - (c->max_len - c->start),
    c->tab + c->start,
    sizeof(double[c->max_len - c->start])
  );
  c->start += max_len - c->max_len;
}
```



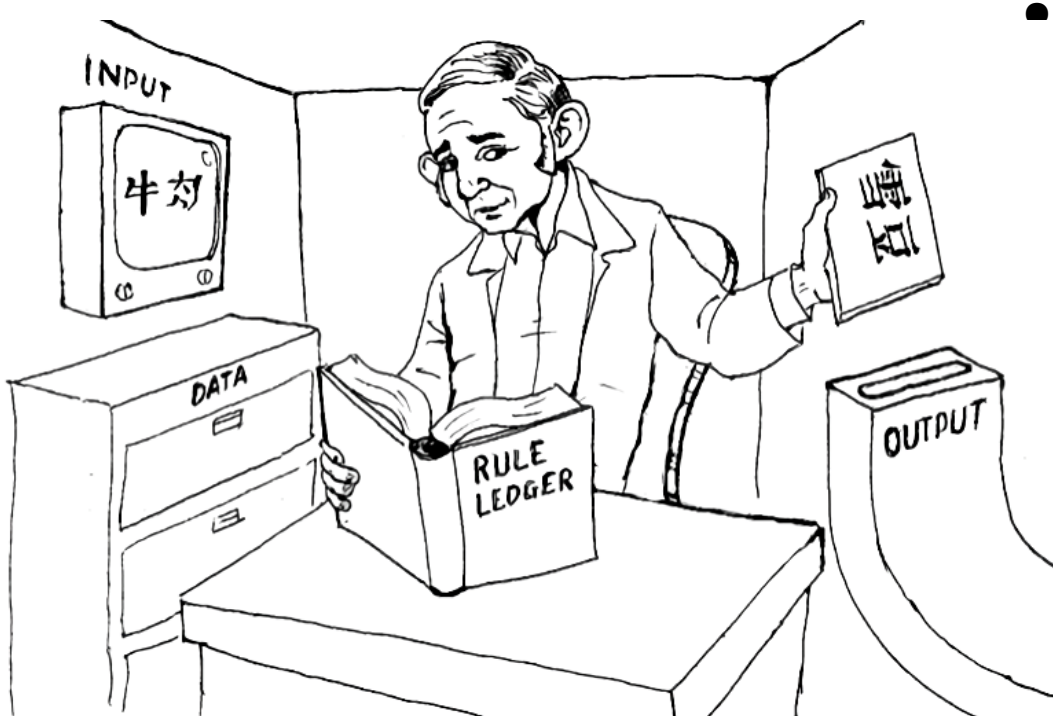
Exs 31: shrinking circular_resize

```
bool pre_wrapping = false;
size_t saved_start = 0;
if (
  max_len < c->max_len
  && c->len
  && circular_getpos(c, c->len-1) < c->start
) {
  pre_wrapping = true;
  saved_start = c->start;
  memmove(
    c->tab + c->start - (c->max_len - max_len),
    c->tab + c->start,
    sizeof(double[c->max_len - c->start])
  );
  c->start = max_len - (c->max_len - c->start);
}

double *newtab = realloc(c->tab, sizeof(double[max_len]));
```

Get ready to undo if needed

Even a shrinking `realloc` may fail

Machine representation



- Von Neumann model:
  - finite number of *registers*
  - *main memory* containing code and data
  - *instruction set* operating on registers and memory transfers

Image attribution: John Searle?

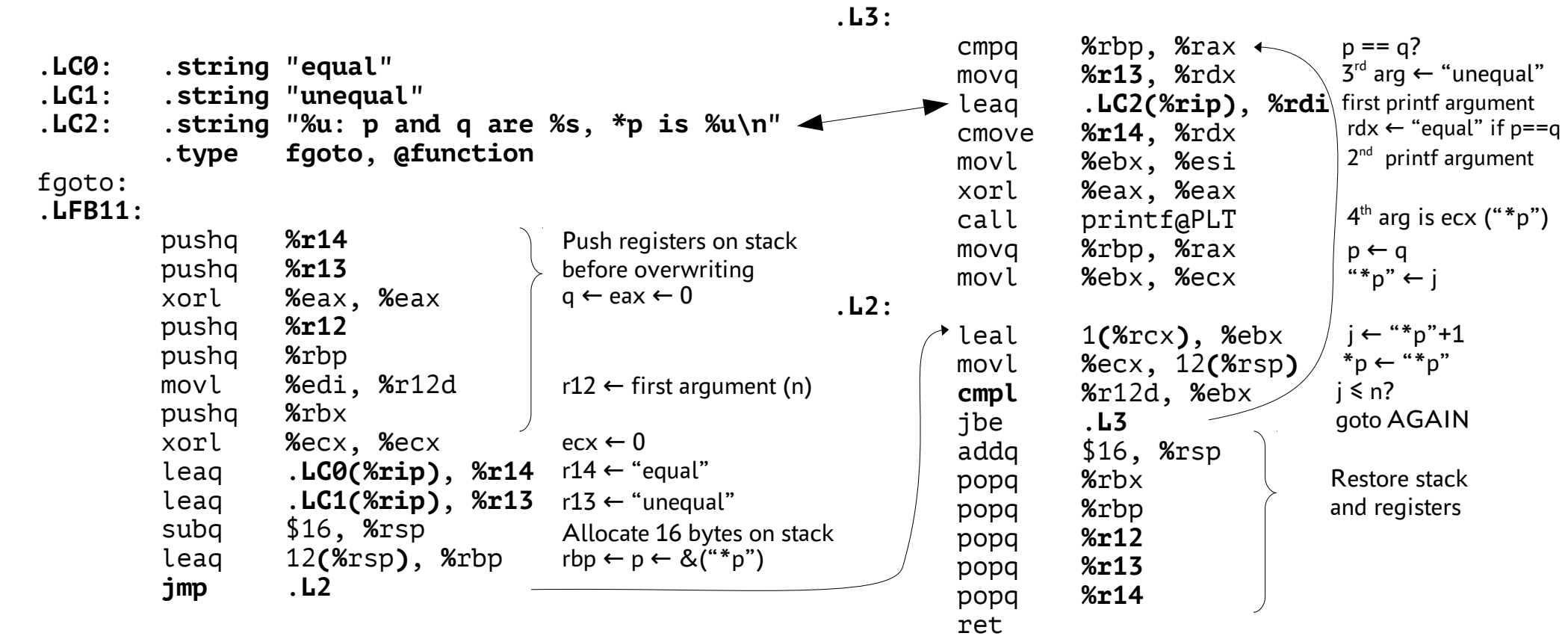
Exs 38: obtaining assembler listings

- `gcc, clang -S`
- `cl.exe /FAcsu`
- Matt Godbolt's Compiler Explorer
- Intel C compiler accepts `-S | /S | /Fa | -Fa`
- Some compilers (e.g. TinyCC) don't output assembly; use `objdump` or a debugger
- The faker's guide to reading x86 assembly

Contrived use of compound literal

```
void fgoto (unsigned n) {
    unsigned j = 0;
    unsigned * p = 0;
    unsigned * q;
  AGAIN:
    if (p) printf (
        "%u: p and q are %s, *p is %u\n",
        j, (q == p) ? "equal" : "unequal", *p
    );
    q = p;
    p = &((unsigned){j,});
    ++j;
    if (j <= n) goto AGAIN;
}
```

Corresponding assembler output



C code approximating assembler output

```
void fgoto (unsigned n) {
    unsigned * q = 0;
    unsigned * p = &((unsigned){0,});
    for (unsigned j = 1; j <= n; ++j) {
        printf(
            "%u: p and q are %s, *p is %u\n",
            j, (q == p) ? "equal" : "unequal", *p
        );
        q = p;
        *p = j;
    }
}
```

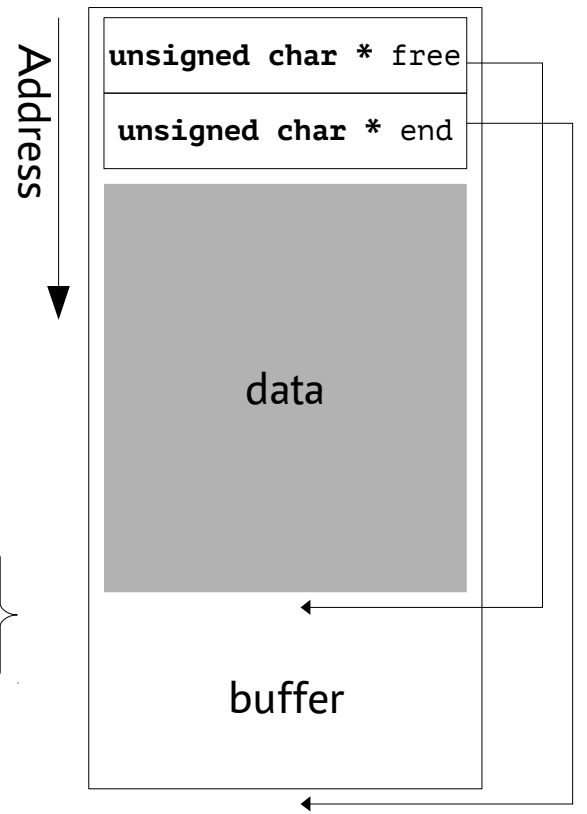
Perform the `j=0` iteration outside of the loop

Assign `p` only once

Update the pointed-to value instead of assigning the pointer to the same updated object

The allocator challenge: simple allocator

```
struct context {
    unsigned char *free, *end;
};

void dyn_init(void * buffer, size_t size) {
    struct context * ctx = buffer;
    ctx->free = (void*)(ctx + 1);
    ctx->end = (unsigned char*)buffer + size;
}

void * dyn_alloc(void * context, size_t size) {
    struct context * ctx = context;
    size_t remainder = size % sizeof(long double);
    if (remainder) size += sizeof(long double) - remainder;
    if (size > ctx->end - ctx->free) return NULL;
    void * ret = ctx->free;
    ctx->free += size;
    return ret;
}
```

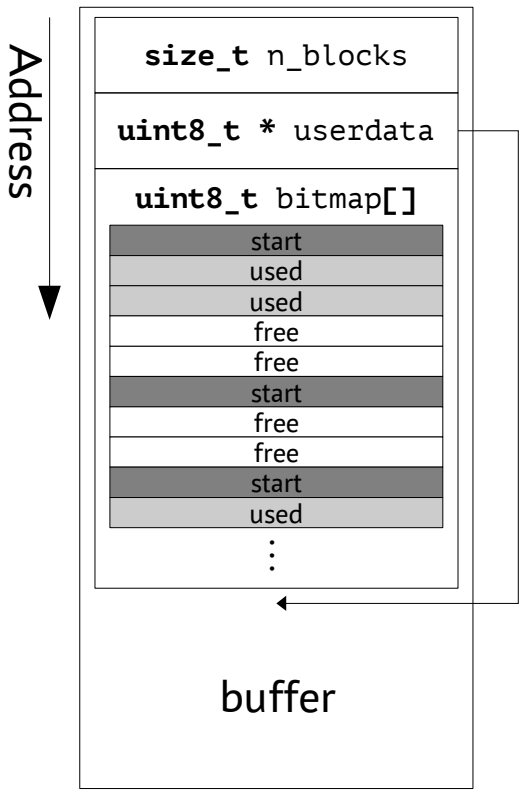


Bump up the allocation size so the next buffer would also be allocated

The allocator challenge: bitmap allocator

```
struct header {
    size_t n_blocks;
    uint8_t * userdata;
    uint8_t bitmap[];
};
```

```
static size_t block_size(
    const struct header * ctx, const uint8_t * start
) {
    uint8_t lookfor =
        *start == STATE_FREE ? STATE_FREE : STATE_USED;
    const uint8_t * block = start+1;
    for (
        const uint8_t * end = ctx->bitmap + ctx->n_blocks;
        block < end && *block == lookfor;
        ++block
    );
    return block - start;
}
```




```
static inline size_t size_to_blocks(size_t size) {
    return size / BLKSZ + (size % BLKSZ > 0);
}
```