# Linked data structures

EECS 211

Winter 2019

# Initial code setup

```
$ cd eecs211
$ curl $URL211/lec/07linked.tgz | tar zx
…
$ cd 07linked
```

# Preliminaries

# Two views on `malloc` and `free`

The client/C view:

- `malloc(n)` gives you an *abstract reference* to a shiny, new, never-before-seen object of n bytes.
- `free(p)` destroys the object `*p`, never to be seen again.

# Two views on `malloc` and `free`

The client/C view:

- `malloc(n)` gives you an *abstract reference* to a shiny, new, never-before-seen object of n bytes.
- `free(p)` destroys the object ∗p, never to be seen again.

The implementation/machine view:

- `malloc(n)` searches a huuuge array of bytes for an unused section of size n, makes a note that the section is now used, and returns its address.
- `free(p)` marks the section that p refers to unused again.

# `valgrind(1)` is a memory debugger

```
$
```

# `valgrind(1)` is a memory debugger

```
$ cat oops.c
```

# valgrind(1) is a memory debugger

```
$ cat oops.c
#include <stdlib.h>
int main() { malloc(211); }
$
```

# valgrind(1) is a memory debugger

```
$ cat oops.c
#include <stdlib.h>
int main() { malloc(211); }
$ cc -o oops oops.c
```

# `valgrind`(1) is a memory debugger

```
$ cat oops.c
#include <stdlib.h>
int main() { malloc(211); }
$ cc -o oops oops.c
$
```

# valgrind(1) is a memory debugger

```
$ cat oops.c
#include <stdlib.h>
int main() { malloc(211); }
$ cc -o oops oops.c
$ ./oops
```

# valgrind(1) is a memory debugger

```
$ cat oops.c
#include <stdlib.h>
int main() { malloc(211); }
$ cc -o oops oops.c
$ ./oops
$
```

# `valgrind`(1) is a memory debugger

```
$ cat oops.c
#include <stdlib.h>
int main() { malloc(211); }
$ cc -o oops oops.c
$ ./oops
$ valgrind ./oops
```

# valgrind(1) is a memory debugger

```
$ cat oops.c
#include <stdlib.h>
int main() { malloc(211); }
$ cc -o oops oops.c
$ ./oops
$ valgrind ./oops
…
==25879== HEAP SUMMARY:
==25879==     in use at exit: 211 bytes in 1 blocks
==25879==   total heap usage: 1 allocs, 0 frees, 211 bytes
==25879==
==25879== LEAK SUMMARY:
==25879==    definitely lost: 211 bytes in 1 blocks
==25879==    indirectly lost: 0 bytes in 0 blocks
==25879==      possibly lost: 0 bytes in 0 blocks
==25879==    still reachable: 0 bytes in 0 blocks
==25879==         suppressed: 0 bytes in 0 blocks
==26289== Rerun with --leak-check=full to see details of le
…
```

## The main event

# How can we deal with growing data?

- `malloc` returns a fixed-sized array

# How can we deal with growing data?

- `malloc` returns a fixed-sized array
- So how does, say, `read_line` work?

# How can we deal with growing data?

- `malloc` returns a fixed-sized array
- So how does, say, `read_line` work?
- It reallocates and copies as needed

# Simplification of read_line

```c
char* read_line(void)
{
    size_t cap   = 0;
    size_t size  = 0;
    char* buffer = NULL;

    for (;;) {
        if (size + 1 > cap) {
            cap    = cap? (2 * cap) : CAPACITY0;
            buffer = realloc_or_die(buffer, cap);
        }

        int c = getchar();

        if (c == EOF || c == '\n') {
            buffer[size] = '\0';
            return buffer;
        } else buffer[size++] = (char) c;
    }
}
```

# The real, slightly more efficient read_line

```
char* read_line(void)
{
    int c = getchar();
    if (c == EOF) return NULL;

    size_t cap   = CAPACITY0;
    size_t size  = 0;
    char* buffer = realloc_or_die(NULL, cap);

    for (;;) {
        if (c == EOF || c == '\n') {
            buffer[size] = '\0';
            return buffer;
        } else buffer[size++] = (char) c;

        c = getchar();

        if (size + 1 > cap) {
            cap   *= 2;
            buffer = realloc_or_die(buffer, cap);
        }
    }
}
```

# The alternative

Doubling a big buffer is highly performant.

# The alternative

Doubling a big buffer is highly performant. (Only not growing at all would be better.)

# The alternative

Doubling a big buffer is highly performant. (Only not growing at all would be better.)

But it’s not smooth, and it’s not very flexible, so there’s an alternative: Instead of one big allocation, lots of small allocations, pointing to each other.

# Remember this?

```
; length : [List-of X] -> Nat
; Finds the length of a list.
(define (length lst)
  (if (empty? lst)
    0
    (+ 1 (length (rest lst)))))
```

# Remember this?

```
; length : [List-of X] -> Nat
; Finds the length of a list.
(define (length lst)
  (if (empty? lst)
    0
    (+ 1 (length (rest lst)))))

(length (cons 2 (cons 3 (cons 4 '()))))
```

# Here’s how it works*

```
struct cons_pair
{
    int car;
    struct cons_pair* cdr;
};
```

# Here’s how it works*

```
typedef struct cons_pair* list_t;


struct cons_pair
{
    int car;
    struct cons_pair* cdr;
};
```

# Here’s how it works*

```
typedef struct cons_pair* list_t;


struct cons_pair
{
    int car;
    list_t cdr;
};
```

# Here's how it works*

In `cons.h`:

```
typedef struct cons_pair* list_t;
```

In `cons.c`:

```
struct cons_pair
{
    int car;
    list_t cdr;
};
```

# cons == malloc + initialization

```
#include <stdlib.h>

list_t cons(int first, list_t rest)
{
    list_t result = malloc(sizeof *result);
    if (result == NULL) … bail out …;

    result->car = first;
    result->cdr = rest;
    return result;
}
```

empty = NULL*

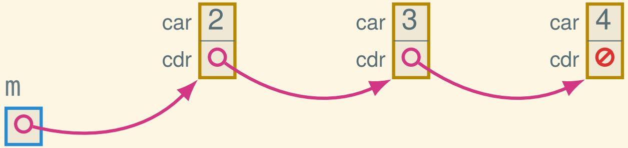
```
const list_t empty = NULL;
```

# Using cons and empty

```
#include "cons.h"

int main()
{
    list_t m = cons(2, cons(3, cons(4, empty)));
```

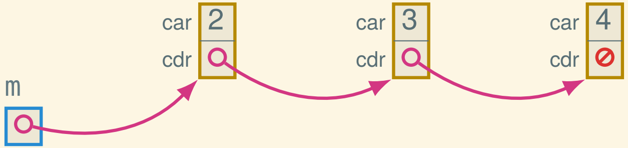
# Using cons and empty

```
#include "cons.h"

int main()
{
    list_t m = cons(2, cons(3, cons(4, empty)));
```



# Using cons and empty

```
#include "cons.h"

int main()
{
    list_t m = cons(2, cons(3, cons(4, empty)));
    // Now what?
```



# We need predicates and selectors

```
bool is_empty(list_t lst) { return lst == NULL; }

bool is_cons(list_t lst) { return lst != NULL; }

int first(list_t lst)
{
    assert( lst );
    return lst->car;
}

list_t rest(list_t lst)
{
    assert( lst );
    return lst->cdr;
}
```

# A whole list program

```c
#include "cons.h"
#include <stdio.h>

int main()
{
    list_t m = cons(2, cons(3, cons(4, empty)));

    while (is_cons(m)) {
        printf("%d\n", first(m));
        m = rest(m);
    }
}
```

# A whole list program, or is it?

```
#include "cons.h"
#include <stdio.h>

int main()
{
    list_t m = cons(2, cons(3, cons(4, empty)));

    while (is_cons(m)) {
        printf("%d\n", first(m));
        m = rest(m);
    }
}
```

# List fun, 111 style

```c
#include "cons.h"

size_t list_len(list_t lst)
{
    return is_empty(lst)
        ? 0
        : 1 + list_len(rest(lst));
}
```

# List fun, 111 style

```
#include "cons.h"

size_t list_len(list_t lst)
{
    return is_empty(lst)
        ? 0
        : 1 + list_len(rest(lst));
}
```

```
(define (length lst)
  (if (empty? lst)
    0
    (+ 1 (length (rest lst)))))
```

# List fun, 211 style

# List fun, 211 style

```
(define (length-acc acc lst)
  (if (empty? lst) acc
    (length-acc (+ 1 acc) (rest lst))))
(define (length lst) (length-acc 0 lst))
```

# List fun, 211 style

```
(define (length-acc acc lst)
  (if (empty? lst) acc
    (length-acc (+ 1 acc) (rest lst))))
(define (length lst) (length-acc 0 lst))
```

```
size_t list_len(list_t lst)
{
    size_t result = 0;
    while (is_cons(lst)) {
        lst = rest(lst);
        ++result;
    }
    return result;
}
```

# Freeing a list, recursively

Back to `cons.c`…

# Freeing a list, recursively

Back to cons.c…

```
void uncons_all(list_t lst)
{
    if (lst) {
        free(lst);
        uncons_all(lst->cdr);
    }
}

void uncons_all(list_t lst)
{
    if (lst) {
        uncons_all(lst->cdr);
        free(lst);
    }
}
```

# Freeing a list, recursively

Back to `cons.c`…

```
void uncons_all(list_t lst)  //Fully broken
{
    if (lst) {
        free(lst);
        uncons_all(lst->cdr);
    }
}

void uncons_all(list_t lst)  //Semi-broken, but
{                            //go with it for now
    if (lst) {
        uncons_all(lst->cdr);
        free(lst);
    }
}
```

# What’s wrong with this program?

```
#include "cons.h"

int main()
{
    list_t m = cons(3, cons(4, empty));
    list_t n = rest(m);
    uncons_all(m);
    printf("%d\n", first(n));
    uncons_all(n);
}
```

# What about this program?

```c
#include "cons.h"

int main()
{
    list_t m = cons(3, cons(4, empty));
    list_t n = cons(2, m);
    printf("%d\n", first(n));
    uncons_all(n);
    printf("%d\n", first(m));
    uncons_all(m);
}
```

# What about this program?

```c
#include "cons.h"

int main()
{
    list_t m = cons(3, cons(4, empty));
    list_t n = cons(2, m);
    printf("%d\n", first(n));
    uncons_all(n);
    printf("%d\n", first(m));
    uncons_all(m);
}
```

Idea: Owners and borrowers.

# Ownership protocol

- The owner of a heap-allocated object is responsible for deallocating it. (No one else may.)

# Ownership protocol

- The owner of a heap-allocated object is responsible for deallocating it. (No one else may.)
- When passing a pointer, it may or may not transfer ownership

# Ownership protocol

- The owner of a heap-allocated object is responsible for deallocating it. (No one else may.)
- When passing a pointer, it may or may not transfer ownership:
  - If it does, then the caller must pass a heap-allocated object that it owns, giving up ownership.

# Ownership protocol

- The owner of a heap-allocated object is responsible for deallocating it. (No one else may.)
- When passing a pointer, it may or may not transfer ownership:
    - If it does, then the caller must pass a heap-allocated object that it owns, giving up ownership.
    - If it does not, then the caller need not own the object, as the callee merely borrows it, and no ownershp changes.

# Ownership protocol

- The owner of a heap-allocated object is responsible for deallocating it. (No one else may.)
- When passing a pointer, it may or may not transfer ownership:
    - If it does, then the caller must pass a heap-allocated object that it owns, giving up ownership.
    - If it does not, then the caller need not own the object, as the callee merely borrows it, and no ownershp changes.

  The only way to tell which is which is to read the contract.

# Ownership protocol

- The owner of a heap-allocated object is responsible for deallocating it. (No one else may.)
- When passing a pointer, it may or may not transfer ownership:
  - If it does, then the caller must pass a heap-allocated object that it owns, giving up ownership.
  - If it does not, then the caller need not own the object, as the callee merely borrows it, and no ownershp changes.

  The only way to tell which is which is to read the contract.
- Functions can also return either owned or borrowed pointers.

# Ownership in `cons.h`

```
// Takes ownership of `rest`, returns owned list:
list_t cons(int first, list_t rest);
```

# Ownership in cons.h

```
// Takes ownership of `rest`, returns owned list:
list_t cons(int first, list_t rest);

// Borrows `lst`, just for call:
bool is_empty(list_t lst), is_cons(list_t lst);
int first(list_t lst);
```

# Ownership in `cons.h`

```
// Takes ownership of `rest`, returns owned list:
list_t cons(int first, list_t rest);

// Borrows `lst`, just for call:
bool is_empty(list_t lst), is_cons(list_t lst);
int first(list_t lst);

// Borrows `lst` and returns borrowed sub-part:
list_t rest(list_t lst);
```

# Ownership in cons.h

```
// Takes ownership of `rest`, returns owned list:
list_t cons(int first, list_t rest);

// Borrows `lst`, just for call:
bool is_empty(list_t lst), is_cons(list_t lst);
int first(list_t lst);

// Borrows `lst` and returns borrowed sub-part:
list_t rest(list_t lst);

// Takes ownership of `lst` (and all it points to):
void uncons_all(list_t lst);
```

# Ownership in `cons.h`

```c
// Takes ownership of `rest`, returns owned list:
list_t cons(int first, list_t rest);

// Borrows `lst`, just for call:
bool is_empty(list_t lst), is_cons(list_t lst);
int first(list_t lst);

// Borrows `lst` and returns borrowed sub-part:
list_t rest(list_t lst);

// Takes ownership of `lst` (and all it points to):
void uncons_all(list_t lst);

// Takes ownership of `lst`, and returns owned
// version of `rest(lst)`:
list_t uncons_one(list_t lst);
```

# Implementations of unconsing

```
list_t uncons_one(list_t lst)
{
    free(lst);
    return lst->cdr;
}
```

# Implementations of unconsing

```
list_t uncons_one(list_t lst)
{
    free(lst);
    return lst->cdr;    //UB!
}
```

# Implementations of unconsing

```
list_t uncons_one(list_t lst)
{
    free(lst);
    return lst->cdr;    //UB!
}

list_t uncons_one(list_t lst)
{
    list_t next = lst->cdr;
    free(lst);
    return next;
}
```

# Implementations of unconsing

```
list_t uncons_one(list_t lst)
{
    free(lst);
    return lst->cdr;    //UB!
}

list_t uncons_one(list_t lst)
{
    list_t next = lst->cdr;
    free(lst);
    return next;
}

void uncons_all(list_t lst)
{
    while (lst) lst = uncons_one(lst);
}
```

# The fixed program

```
#include "cons.h"

int main()
{
    list_t m = cons(3, cons(4, empty));
    list_t n = uncons_one(m);
    printf("%d\n", first(n));
    uncons_all(n);
}
```

# The fixed program

```c
#include "cons.h"

int main()
{
    list_t m = cons(3, cons(4, empty));
    list_t n = uncons_one(m);
    printf("%d\n", first(n));
    uncons_all(n);
}
```

– Next time: RAII –

# Notes

* Lies