Linked Data Structures

CS 211

Initial code setup

The code in this course is available in your Unix shell account. You can get your own copy like this:

```
% cd cs211
% tar -xvkf ~cs211/lec/08_linked.tgz
⋮
% cd 08_linked
```

Road map

Preliminaries

- How to think about `malloc()` & `free()`
- Understanding *Address Sanitizer*

Road map

Preliminaries

- How to think about `malloc()` & `free()`
- Understanding *Address Sanitizer*

Dealing with growing data

- Dynamic arrays
- Singly-linked lists
- Double trouble

Road map

- Preliminaries
  - How to think about `malloc()` & `free()`
  - Understanding *Address Sanitizer*
- Dealing with growing data
  - Dynamic arrays
  - Singly-linked lists
  - Double trouble
- Ownership & borrowing
  - The concept
  - Ownership example: Linked lists

Road map

- Preliminaries
  - How to think about `malloc()` & `free()`
  - Understanding *Address Sanitizer*
- Dealing with growing data
  - Dynamic arrays
  - Singly-linked lists
  - Double trouble
- Ownership & borrowing
  - The concept
  - Ownership example: Linked lists
- Extended coding example

Up next

Preliminaries

- How to think about `malloc()` & `free()`
- Understanding *Address Sanitizer*

Dealing with growing data

- Dynamic arrays
- Singly-linked lists
- Double trouble

Ownership & borrowing

- The concept
- Ownership example: Linked lists

Extended coding example

Up next

- Preliminaries
  - How to think about `malloc()` & `free()`
  - Understanding *Address Sanitizer*
- Dealing with growing data
  - Dynamic arrays
  - Singly-linked lists
  - Double trouble
- Ownership & borrowing
  - The concept
  - Ownership example: Linked lists
- Extended coding example

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 (or fails).
- `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 (or fails).
- `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 (or fails).
- `free(p)` marks the section that `p` refers to unused again.

Two views on `malloc` and `free`

The abstract client/C view:

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

The concrete 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 (or fails).
- `free(p)` marks the section that `p` refers to unused again.

Two views on `malloc` and `free`

The abstract client/C view (CS 211):

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

The concrete implementation/machine view (CS 213):

- `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 (or fails).
- `free(p)` marks the section that `p` refers to unused again.

Up next

Preliminaries

- How to think about `malloc()` & `free()`
- Understanding *Address Sanitizer*

Dealing with growing data

- Dynamic arrays
- Singly-linked lists
- Double trouble

Ownership & borrowing

- The concept
- Ownership example: Linked lists

Extended coding example

`ASan` is a memory debugger

Let’s try it on a mystery program:

```
%
```

`ASan` is a memory debugger

Let’s try it on a mystery program:

```
% cc -o oops oops.c -g
```

`ASan` is a memory debugger

Let's try it on a mystery program:

```
% cc -o oops oops.c -g
%
```

`ASan` is a memory debugger

Let's try it on a mystery program:

```
% cc -o oops oops.c -g
% ./oops
```

`ASan` is a memory debugger

Let’s try it on a mystery program:

```
% cc -o oops oops.c -g
% ./oops
%
```

`ASan` is a memory debugger

Let's try it on a mystery program:

```
% cc -o oops oops.c -g
% ./oops
% cc -o oops oops.c -g -fsanitize=address
```

`ASan` is a memory debugger

Let's try it on a mystery program:

```
% cc -o oops oops.c -g
% ./oops
% cc -o oops oops.c -g -fsanitize=address
%
```

`ASan` is a memory debugger

Let's try it on a mystery program:

```
% cc -o oops oops.c -g
% ./oops
% cc -o oops oops.c -g -fsanitize=address
% ./oops
```

`ASan` is a memory debugger

Let's try it on a mystery program:

```
% cc -o oops oops.c -g
% ./oops
% cc -o oops oops.c -g -fsanitize=address
% ./oops
=================================================================
==9256==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6030000
8 at pc 0x00000040077a bp 0x7ffd6fa83340 sp 0x7ffd6fa83338
READ of size 4 at 0x603000000028 thread T0
SCARINESS: 17 (4-byte-read-heap-buffer-overflow)
    #0 0x400779 in use_it /home/cs211/oops.c:10
    #1 0x40079e in main /home/cs211/oops.c:15
    #2 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
    #3 0x400638  (/home/cs211/oops+0x400638)
```

Let's read some `ASan` output

We went out of bounds on an array ("buffer") allocated using `malloc` ("heap").

```
==9256==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60300000002
8 at pc 0x00000040077a bp 0x7ffd6fa83340 sp 0x7ffd6fa83338
READ of size 4 at 0x603000000028 thread T0
SCARINESS: 17 (4-byte-read-heap-buffer-overflow)
    #0 0x400779 in use_it /home/cs211/oops.c:10
    #1 0x40079e in main /home/cs211/oops.c:15
    #2 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
    #3 0x400638  (/home/cs211/oops+0x400638)

0x603000000028 is located 4 bytes to the right of 20-byte region [0x60300000001
0,0x603000000024)
allocated by thread T0 here:
    #0 0x7fca2d796610 in malloc (/usr/lib64/libasan.so.5+0xe9610)
    #1 0x40071d in allocate_it /home/cs211/oops.c:5
    #2 0x400791 in main /home/cs211/oops.c:15
    #3 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
```

Let's read some ASan output

We were trying to use a 4-byte value (like an `int`).

```
==9256==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6030000002
8 at pc 0x00000040077a bp 0x7ffd6fa83340 sp 0x7ffd6fa83338
READ of size 4 at 0x603000000028 thread T0
SCARINESS: 17 (4-byte-read-heap-buffer-overflow)
    #0 0x400779 in use_it /home/cs211/oops.c:10
    #1 0x40079e in main /home/cs211/oops.c:15
    #2 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
    #3 0x400638  (/home/cs211/oops+0x400638)

0x603000000028 is located 4 bytes to the right of 20-byte region [0x60300000001
0,0x603000000024)
allocated by thread T0 here:
    #0 0x7fca2d796610 in malloc (/usr/lib64/libasan.so.5+0xe9610)
    #1 0x40071d in allocate_it /home/cs211/oops.c:5
    #2 0x400791 in main /home/cs211/oops.c:15
    #3 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
```

Let's read some *ASan* output

The bad access was on L10 of `oops.c` in function `use_it()`...

```
==9256==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60300000002
8 at pc 0x00000040077a bp 0x7ffd6fa83340 sp 0x7ffd6fa83338
READ of size 4 at 0x603000000028 thread T0
SCARINESS: 17 (4-byte-read-heap-buffer-overflow)
    #0 0x400779 in use_it /home/cs211/oops.c:10
    #1 0x40079e in main /home/cs211/oops.c:15
    #2 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
    #3 0x400638  (/home/cs211/oops+0x400638)

0x603000000028 is located 4 bytes to the right of 20-byte region [0x60300000001
0,0x603000000024)
allocated by thread T0 here:
    #0 0x7fca2d796610 in malloc (/usr/lib64/libasan.so.5+0xe9610)
    #1 0x40071d in allocate_it /home/cs211/oops.c:5
    #2 0x400791 in main /home/cs211/oops.c:15
    #3 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
```

Let's read some *ASan* output

…which was called from L15 of `oops.c` in function `main()`.

```
==9256==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60300000002
8 at pc 0x00000040077a bp 0x7ffd6fa83340 sp 0x7ffd6fa83338
READ of size 4 at 0x603000000028 thread T0
SCARINESS: 17 (4-byte-read-heap-buffer-overflow)
    #0 0x400779 in use_it /home/cs211/oops.c:10
    #1 0x40079e in main /home/cs211/oops.c:15
    #2 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
    #3 0x400638  (/home/cs211/oops+0x400638)

0x603000000028 is located 4 bytes to the right of 20-byte region [0x60300000001
0,0x603000000024)
allocated by thread T0 here:
    #0 0x7fca2d796610 in malloc (/usr/lib64/libasan.so.5+0xe9610)
    #1 0x40071d in allocate_it /home/cs211/oops.c:5
    #2 0x400791 in main /home/cs211/oops.c:15
    #3 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
```

Let's read some `ASan` output

The address we tried to access was not far from the end of an array,...

```
==9256==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60300000002
8 at pc 0x00000040077a bp 0x7ffd6fa83340 sp 0x7ffd6fa83338
READ of size 4 at 0x603000000028 thread T0
SCARINESS: 17 (4-byte-read-heap-buffer-overflow)
    #0 0x400779 in use_it /home/cs211/oops.c:10
    #1 0x40079e in main /home/cs211/oops.c:15
    #2 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
    #3 0x400638  (/home/cs211/oops+0x400638)

0x603000000028 is located 4 bytes to the right of 20-byte region [0x60300000001
0,0x603000000024)
allocated by thread T0 here:
    #0 0x7fca2d796610 in malloc (/usr/lib64/libasan.so.5+0xe9610)
    #1 0x40071d in allocate_it /home/cs211/oops.c:5
    #2 0x400791 in main /home/cs211/oops.c:15
    #3 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
```

Let's read some *ASan* output

...which was allocated using `malloc()`,...

```
==9256==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60300000002
8 at pc 0x00000040077a bp 0x7ffd6fa83340 sp 0x7ffd6fa83338
READ of size 4 at 0x603000000028 thread T0
SCARINESS: 17 (4-byte-read-heap-buffer-overflow)
    #0 0x400779 in use_it /home/cs211/oops.c:10
    #1 0x40079e in main /home/cs211/oops.c:15
    #2 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
    #3 0x400638  (/home/cs211/oops+0x400638)

0x603000000028 is located 4 bytes to the right of 20-byte region [0x60300000001
0,0x603000000024)
allocated by thread T0 here:
    #0 0x7fca2d796610 in malloc (/usr/lib64/libasan.so.5+0xe9610)
    #1 0x40071d in allocate_it /home/cs211/oops.c:5
    #2 0x400791 in main /home/cs211/oops.c:15
    #3 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
```

Let's read some *ASan* output

...which was called from L5 of `oops.c` in function `allocate_it()`,...

```
==9256==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60300000002
8 at pc 0x00000040077a bp 0x7ffd6fa83340 sp 0x7ffd6fa83338
READ of size 4 at 0x603000000028 thread T0
SCARINESS: 17 (4-byte-read-heap-buffer-overflow)
    #0 0x400779 in use_it /home/cs211/oops.c:10
    #1 0x40079e in main /home/cs211/oops.c:15
    #2 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
    #3 0x400638  (/home/cs211/oops+0x400638)

0x603000000028 is located 4 bytes to the right of 20-byte region [0x60300000001
0,0x603000000024)
allocated by thread T0 here:
    #0 0x7fca2d796610 in malloc (/usr/lib64/libasan.so.5+0xe9610)
    #1 0x40071d in allocate_it /home/cs211/oops.c:5
    #2 0x400791 in main /home/cs211/oops.c:15
    #3 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
```

Let's read some `ASan` output

...which was called from L15 of `oops.c` in function `main()`.

```
==9256==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60300000002
8 at pc 0x00000040077a bp 0x7ffd6fa83340 sp 0x7ffd6fa83338
READ of size 4 at 0x603000000028 thread T0
SCARINESS: 17 (4-byte-read-heap-buffer-overflow)
    #0 0x400779 in use_it /home/cs211/oops.c:10
    #1 0x40079e in main /home/cs211/oops.c:15
    #2 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
    #3 0x400638  (/home/cs211/oops+0x400638)

0x603000000028 is located 4 bytes to the right of 20-byte region [0x60300000001
0,0x603000000024)
allocated by thread T0 here:
    #0 0x7fca2d796610 in malloc (/usr/lib64/libasan.so.5+0xe9610)
    #1 0x40071d in allocate_it /home/cs211/oops.c:5
    #2 0x400791 in main /home/cs211/oops.c:15
    #3 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
```

Let's read some `ASan` output

Let's go see `oops.c`*...*

```
==9256==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60300000002
8 at pc 0x00000040077a bp 0x7ffd6fa83340 sp 0x7ffd6fa83338
READ of size 4 at 0x603000000028 thread T0
SCARINESS: 17 (4-byte-read-heap-buffer-overflow)
    #0 0x400779 in use_it /home/cs211/oops.c:10
    #1 0x40079e in main /home/cs211/oops.c:15
    #2 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
    #3 0x400638  (/home/cs211/oops+0x400638)

0x603000000028 is located 4 bytes to the right of 20-byte region [0x60300000001
0,0x603000000024)
allocated by thread T0 here:
    #0 0x7fca2d796610 in malloc (/usr/lib64/libasan.so.5+0xe9610)
    #1 0x40071d in allocate_it /home/cs211/oops.c:5
    #2 0x400791 in main /home/cs211/oops.c:15
    #3 0x3f63e1ed1f in __libc_start_main (/lib64/libc.so.6+0x3f63e1ed1f)
```

The source

src/oops.c

```
#include <stdlib.h>

int* allocate_it(size_t n)
{
    return malloc(n * sizeof(int));
}

void use_it(size_t i, int* p)
{
    ++p[i];
}

int main(void)
{
    use_it(6, allocate_it(5));
}
```

Up next

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

Up next

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

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:

```
// Allocates a size-byte buffer, copies the contents of
// ptr to it, frees ptr, and returns a pointer to the
// new buffer:
void* realloc(void* ptr, size_t size);
```

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:

```
// Allocates a size-byte buffer, copies the contents of
// ptr to it, frees ptr, and returns a pointer to the
// new buffer:
void* realloc(void* ptr, size_t size);

// But:
//   - if ptr is NULL it mallocs
//   - probably optimized
```

A pattern for quick-and-dirty error handling

```
static
void* surely_realloc(void* pold, size_t size)
{
    void* pnew = realloc(pold, size);

    if (pnew == NULL) {
        free(pold);
        perror(NULL);
        exit(1);
    }

    return pnew;
}
```

(slightly incorrect) Simplification of `read_line()`

```
char* read_line(void)
{
    size_t cap = 0, fill = 0;
    char* buffer = NULL;

    for (;;) {
        if (fill + 1 > cap) {
            cap    = (cap > 0) ? (2 * cap) : CAPACITY0;
            buffer = surely_realloc(buffer, cap);
        }

        int c = getchar();
        if (c == EOF || c == '\n') { // wrong when c == EOF && !fill
            buffer[fill] = 0;
            return buffer;
        } else buffer[fill++] = (char) c;
    }
}
```

The real, more correct `read_line`

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

    size_t cap = CAPACITY0, fill = 0;
    char* buffer = surely_malloc(cap);

    for (;;) {
        if (c == EOF || c == '\n') {
            buffer[fill] = 0;
            return buffer;
        }

        buffer[fill++] = (char) c;
        c = getchar();

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

The alternative

If you don't know how much data you're going to have, then doubling a big buffer is highly performant.

The alternative

If you don’t know how much data you’re going to have, then doubling a big buffer is highly performant. (Only not growing at all would be better.)

The alternative

If you don’t know how much data you’re going to have, then doubling a big buffer is highly performant. (Only not growing at all would be better.)

But:

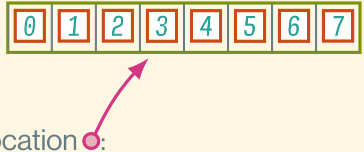
- it’s not smooth, and
- it’s not very flexible

The alternative

If you don't know how much data you're going to have, then 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,



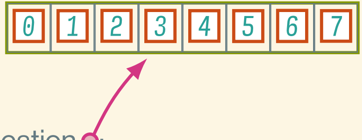
so there's an alternative to one big array allocation :

The alternative

If you don't know how much data you're going to have, then doubling a big buffer is highly performant. (Only not growing at all would be better.)

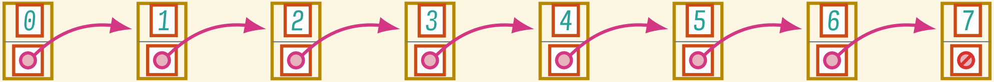
But:

- it's not smooth, and
- it's not very flexible,



so there's an alternative to one big array allocation:

lots of small "node" allocations that point to each other



Up next

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*

```c
typedef struct cons_pair* list_t;


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

Here's how it works*

```c
typedef struct cons_pair* list_t;


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

Here's how it works*

src/cons.h

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

src/cons.c

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

`cons` == `malloc` + initialization

```
list_t cons(int first, list_t rest);
```

`src/cons.h`

`cons` == `malloc` + initialization

src/cons.h

```c
list_t cons(int first, list_t rest);
```

src/cons.c

```c
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**

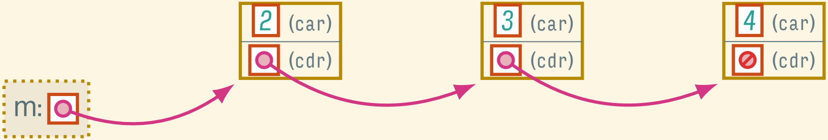
```
list_t const empty = NULL;
```

Using `cons` and `empty`

```c
#include "cons.h"

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

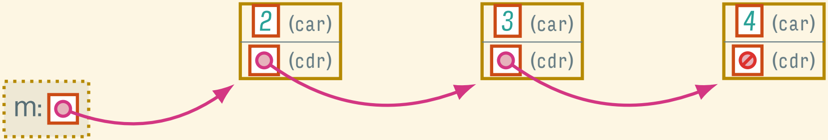
Using `cons` and `empty`

```
#include "cons.h"

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



Using `cons` and `empty`

```c
#include "cons.h"

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



How about some predicates and selectors?

```c
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 );      // crash here to avoid UB
    return lst->car;
}

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

A whole list program

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

int main(void)
{
    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?

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

int main(void)
{
    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

```c
#include "cons.h"

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

```scheme
(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 count = 0;
    while (is_cons(lst)) {
        lst = rest(lst);
        ++count;
    }
    return count;
}
```

Freeing a list

Back to `cons.c`...

Freeing a list

Back to `cons.c`... Which of these is better?

```c
void uncons_all_1(list_t lst)
{
    while (lst) {
        free(lst);
        lst = lst->cdr;
    }
}

void uncons_all_2(list_t lst)
{
    if (lst) {
        uncons_all_2(lst->cdr);
        free(lst);
    }
}
```

Freeing a list

Back to `cons.c`... Which of these is better?

```
void uncons_all_1(list_t lst)    // Fully broken
{
    while (lst) {
        free(lst);
        lst = lst->cdr;
    }
}

void uncons_all_2(list_t lst)    // Could overflow stack,
{                                // but go with it for now
    if (lst) {
        uncons_all_2(lst->cdr);
        free(lst);
    }
}
```

Up next

What's wrong with this program?

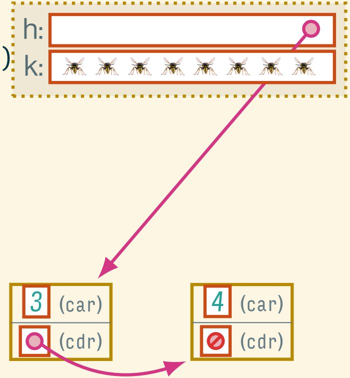
```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = rest(h);

    printf("%d\n", first(h));
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```

What’s wrong with this program?

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty)
    list_t k = rest(h);

    printf("%d\n", first(h));
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```

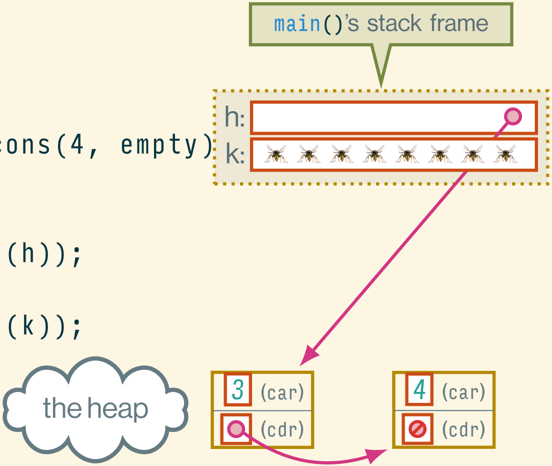
What's wrong with this program?

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = rest(h);

    printf("%d\n", first(h));
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```



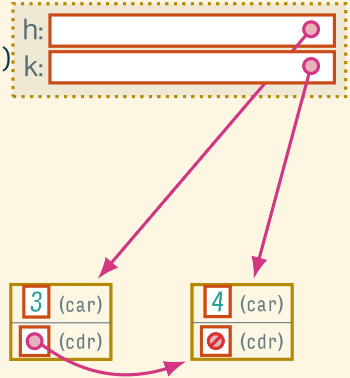
What’s wrong with this program?

```
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = rest(h);

    printf("%d\n", first(h));
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```



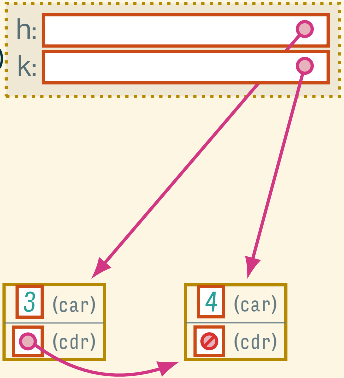
What's wrong with this program?

```
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = rest(h);

    printf("%d\n", first(h));
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```



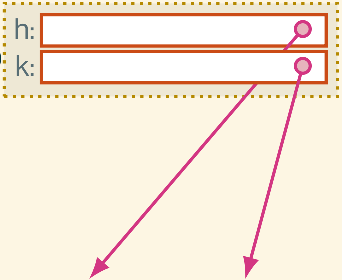
What's wrong with this program?

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = rest(h);

    printf("%d\n", first(h));  // 3⏎
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```



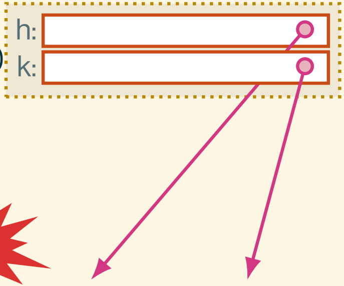
What’s wrong with this program?

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty)
    list_t k = rest(h);

    printf("%d\n", first(h));  // 3⏎
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```



What's wrong with this program?

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty)
    list_t k = rest(h);

    printf("%d\n", first(h));  // 3⏎
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```



What about this program?

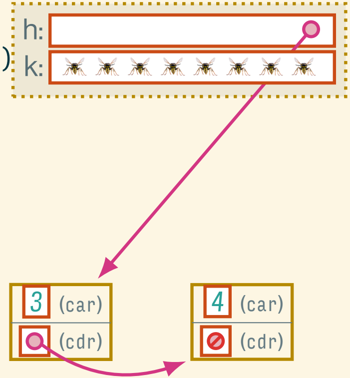
```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = cons(2, h);

    printf("%d\n", first(k));
    uncons_all(k);
    printf("%d\n", first(h));
    uncons_all(h);
}
```

What about this program?

```
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty)
    list_t k = cons(2, h);

    printf("%d\n", first(k));
    uncons_all(k);
    printf("%d\n", first(h));
    uncons_all(h);
}
```

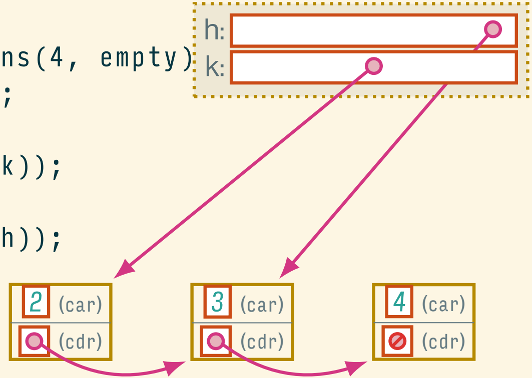
What about this program?

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = cons(2, h);

    printf("%d\n", first(k));
    uncons_all(k);
    printf("%d\n", first(h));
    uncons_all(h);
}
```



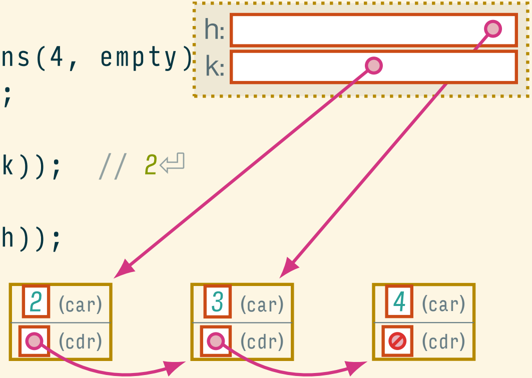
What about this program?

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = cons(2, h);

    printf("%d\n", first(k));
    uncons_all(k);
    printf("%d\n", first(h));
    uncons_all(h);
}
```



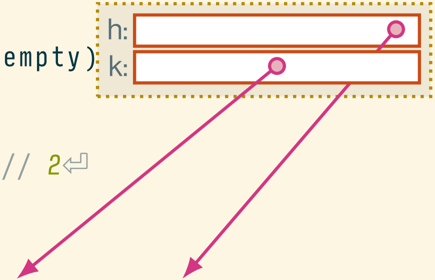
What about this program?

```
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = cons(2, h);

    printf("%d\n", first(k));  // 2⏎
    uncons_all(k);
    printf("%d\n", first(h));
    uncons_all(h);
}
```



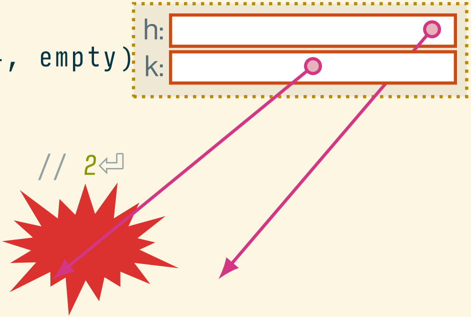
What about this program?

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = cons(2, h);

    printf("%d\n", first(k));  // 2⏎
    uncons_all(k);
    printf("%d\n", first(h));
    uncons_all(h);
}
```



What about this program?

```
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty)
    list_t k = cons(2, h);

    printf("%d\n", first(k));  // 2⏎
    uncons_all(k);
    printf("%d\n", first(h));
    uncons_all(h);
}
```



What about this one?

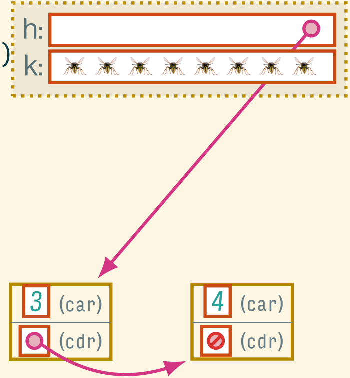
```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = cons(2, h);

    printf("%d\n", first(h));
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```

What about this one?

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty)
    list_t k = cons(2, h);

    printf("%d\n", first(h));
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```

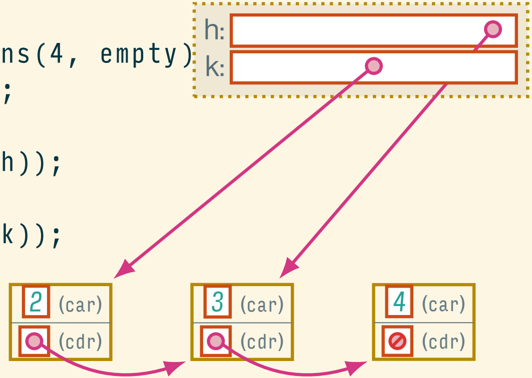
What about this one?

```
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = cons(2, h);

    printf("%d\n", first(h));
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```



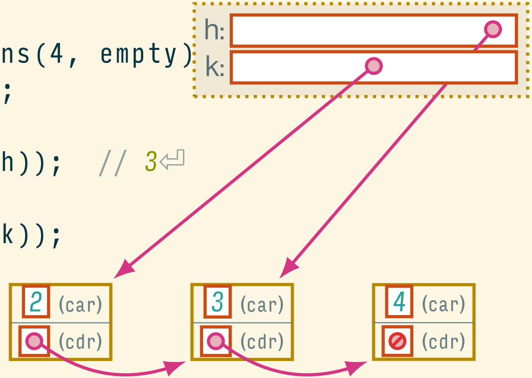
What about this one?

```
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = cons(2, h);

    printf("%d\n", first(h));
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```



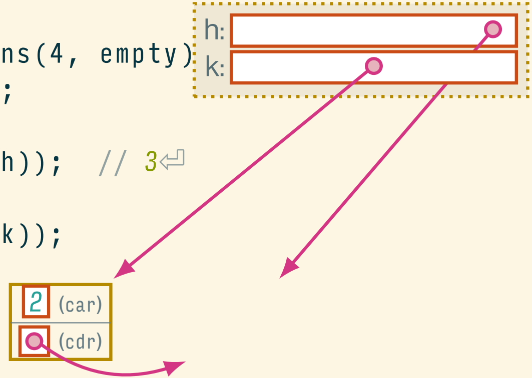
What about this one?

```
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = cons(2, h);

    printf("%d\n", first(h));  // 3⏎
    uncons_all(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```



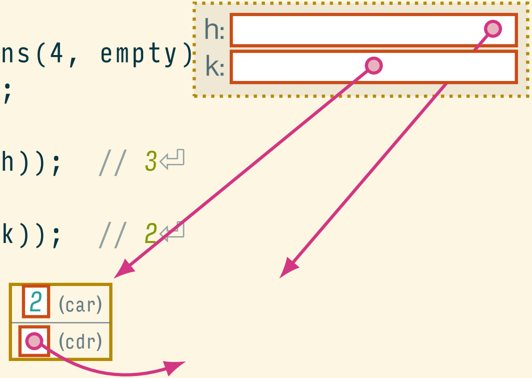
What about this one?

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty)
    list_t k = cons(2, h);

    printf("%d\n", first(h));  // 3↵
    uncons_all(h);
▶   printf("%d\n", first(k));
    uncons_all(k);
}
```



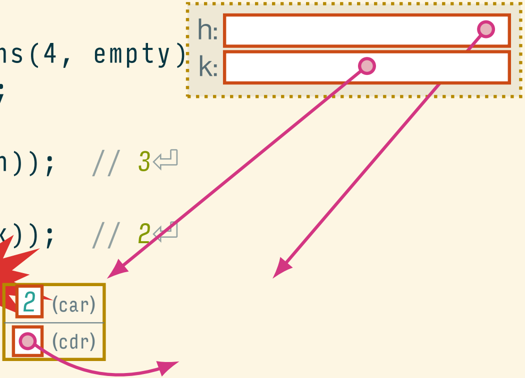
What about this one?

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty)
    list_t k = cons(2, h);

    printf("%d\n", first(h));  // 3⏎
    unconst_all(h);
    printf("%d\n", first(k));  // 2⏎
▶   uncons_all(k);
}
```



What about this one?

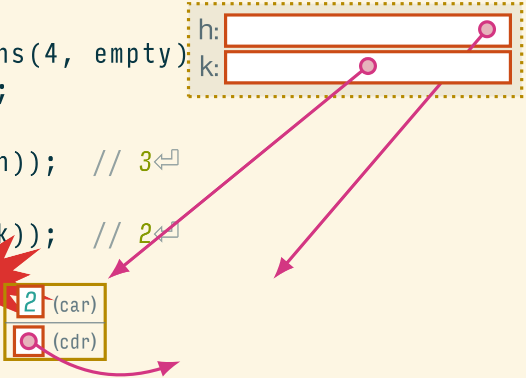
```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty)
    list_t k = cons(2, h);

    printf("%d\n", first(h));  // 3⏎
    uncons_all(h);
    printf("%d\n", first(k));  // 2⏎
    uncons_all(k);
}
```



What about this one?

Idea: Avoid aliasing

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));
    list_t k = cons(2, h);

    printf("%d\n", first(h));  // 3⏎
    uncons_all(h);
    printf("%d\n", first(k));  // 2⏎
    uncons_all(k);
}
```



Up next

A helper function for the next slide

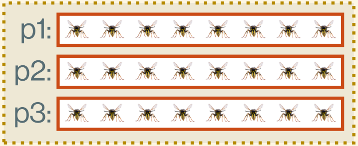
```
// Allocates, initializes, and returns a new `short` object
// on the heap. (Exits, rudely, on failure.)
short* new_short(short value)
{
    short* result = malloc(sizeof(short));
    if (result) {
        *result = value;
        return result;
    } else {
        perror(NULL);
        exit(1);
    }
}
```

Aliasing

A unique pointer is the only pointer to its object

Aliasing is when you have multiple pointers to the same object

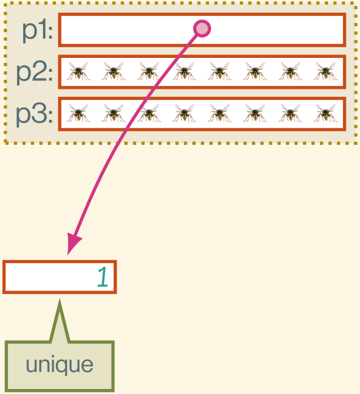
```
void f(void)
{
    short* p1 = new_short(1);
    short* p2 = new_short(2);
    short* p3 = p1;

    *p3 = 10;
    printf("%d\n", *p1);

    free(p3);
    printf("%d\n", *p1);
}
```



Aliasing

A unique pointer is the only pointer to its object
Aliasing is when you have multiple pointers to the same object

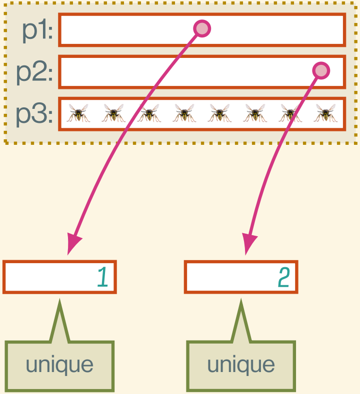
```
void f(void)
{
    short* p1 = new_short(1);
    short* p2 = new_short(2);
    short* p3 = p1;

    *p3 = 10;
    printf("%d\n", *p1);

    free(p3);
    printf("%d\n", *p1);
}
```



Aliasing

A unique pointer is the only pointer to its object
Aliasing is when you have multiple pointers to the same object

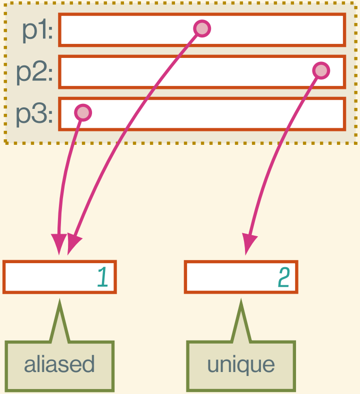
```
void f(void)
{
    short* p1 = new_short(1);
    short* p2 = new_short(2);
  ▶ short* p3 = p1;

    *p3 = 10;
    printf("%d\n", *p1);

    free(p3);
    printf("%d\n", *p1);
}
```



Aliasing

A unique pointer is the only pointer to its object
Aliasing is when you have multiple pointers to the same object

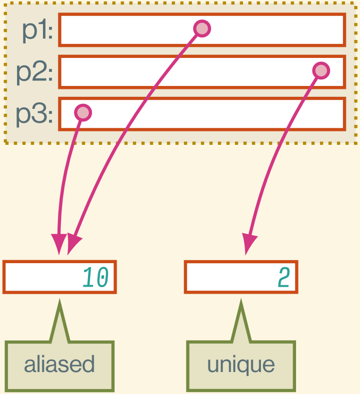
```
void f(void)
{
    short* p1 = new_short(1);
    short* p2 = new_short(2);
    short* p3 = p1;

  ▶ *p3 = 10;
    printf("%d\n", *p1);

    free(p3);
    printf("%d\n", *p1);
}
```



Aliasing

A unique pointer is the only pointer to its object
Aliasing is when you have multiple pointers to the same object

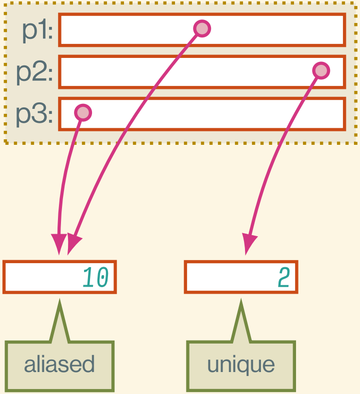
```
void f(void)
{
    short* p1 = new_short(1);
    short* p2 = new_short(2);
    short* p3 = p1;

    *p3 = 10;
  ▶ printf("%d\n", *p1);

    free(p3);
    printf("%d\n", *p1);
}
```



Aliasing

A unique pointer is the only pointer to its object
Aliasing is when you have multiple pointers to the same object

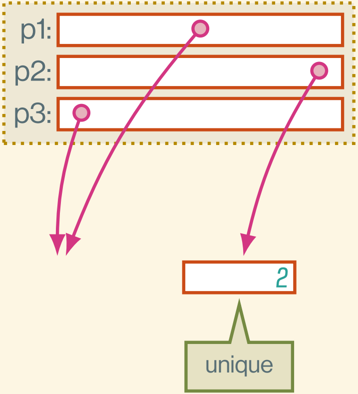
```
void f(void)
{
    short* p1 = new_short(1);
    short* p2 = new_short(2);
    short* p3 = p1;

    *p3 = 10;
    printf("%d\n", *p1);    // 10⏎

  ▶ free(p3);
    printf("%d\n", *p1);
}
```



Aliasing

A unique pointer is the only pointer to its object
Aliasing is when you have multiple pointers to the same object

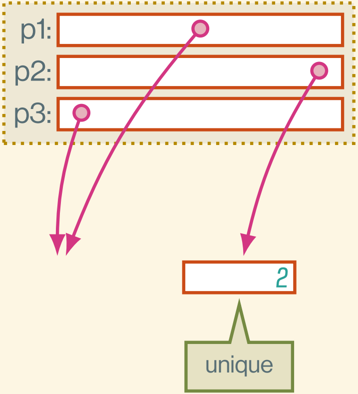
```
void f(void)
{
    short* p1 = new_short(1);
    short* p2 = new_short(2);
    short* p3 = p1;

    *p3 = 10;
    printf("%d\n", *p1);    // 10⏎

    free(p3);
  ▶ printf("%d\n", *p1);
}
```



Aliasing

A unique pointer is the only pointer to its object
Aliasing is when you have multiple pointers to the same object

```
void f(void)
{
    short* p1 = new_short(1);
    short* p2 = new_short(2);
    short* p3 = p1;

    *p3 = 10;
    printf("%d\n", *p1);    // 10⏎

    free(p3);
    printf("%d\n", *p1);
}
```



What if we avoided aliasing?

If no objects are aliased, then we know:

What if we avoided aliasing?

If no objects are aliased, then we know:

- If we have a pointer to an object, no one else has it

What if we avoided aliasing?

If no objects are aliased, then we know:

- If we have a pointer to an object, no one else has it
- Thus we can safely free it!

What if we avoided aliasing?

If no objects are aliased, then we know:

- If we have a pointer to an object, no one else has it
- Thus we can safely free it!
- Furthermore: We *must* free it, since no one else will (or can)

What if we avoided aliasing?

If no objects are aliased, then we know:

- If we have a pointer to an object, no one else has it
- Thus we can safely free it!
- Furthermore: We *must* free it, since no one else will (or can)

Problem: Uniqueness is too restrictive

What if we avoided aliasing?

If no objects are aliased, then we know:

- If we have a pointer to an object, no one else has it
- Thus we can safely free it!
- Furthermore: We *must* free it, since no one else will (or can)

Problem: Uniqueness is too restrictive

For example, it doesn't allow `rest()`:

```
// Returns an alias of `cdr` of `lst`
list_t rest(list_t lst)
{
    assert( lst );
    return lst->cdr;
}
```

Up next

Next idea: Owning pointers & borrowing pointers

We will consider some pointers to own their pointed-to object,

Next idea: Owning pointers & borrowing pointers

We will consider some pointers to own their pointed-to object,
and other pointers merely to borrow their pointed-to object.

Next idea: Owning pointers & borrowing pointers

We will consider some pointers to own their pointed-to object,
and other pointers merely to borrow their pointed-to object.

Then:

- Owning pointers must be unique as owners — one object can't have multiple owners
- Borrowing pointers are allowed to alias anything, borrowed or owned

The Ownership Protocol

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

The Ownership Protocol

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

The Ownership Protocol

- The owner of a heap-allocated object is responsible for deallocating it. (No one else may.)
- Borrowers of an object may use it.
- Passing a pointer may or may not transfer ownership:

The Ownership Protocol

- The owner of a heap-allocated object is responsible for deallocating it. (No one else may.)
- Borrowers of an object may use it.
- Passing a pointer may or may not transfer ownership:
  - If so, then the caller must own the object before the call; the caller gives up ownership via the call.

The Ownership Protocol

- The owner of a heap-allocated object is responsible for deallocating it. (No one else may.)
- Borrowers of an object may use it.
- Passing a pointer may or may not transfer ownership:
  - If so, then the caller must own the object before the call; the caller gives up ownership via the call.
  - If not, then the caller need not own the object, as the callee merely borrows it; no ownership changes.

The Ownership Protocol

- The owner of a heap-allocated object is responsible for deallocating it. (No one else may.)
- Borrowers of an object may use it.
- Passing a pointer may or may not transfer ownership:
  - If so, then the caller must own the object before the call; the caller gives up ownership via the call.
  - If not, then the caller need not own the object, as the callee merely borrows it; no ownership changes.

  The only way to tell which is which is to read the contract.
- Same deal when returning a pointer: it may or may not transfer ownership, so you have to read the contract.

Up next

Ownership in `cons.h`

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

Ownership in `cons.h`

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

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

// Borrows `lst` and returns borrowed sub-object
list_t rest(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' transiently (just for the call)
bool is_empty(list_t lst), is_cons(list_t lst);
int first(list_t lst);

// Borrows 'lst' and returns borrowed sub-object
list_t rest(list_t lst);

// Takes ownership of 'lst' (and all that 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` transiently (just for the call)
bool is_empty(list_t lst), is_cons(list_t lst);
int first(list_t lst);

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

// Takes ownership of `lst` (and all that 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);
```

Implementing `uncons_one()`

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

Implementing `uncons_one()`

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

Implementing `uncons_one()`

```c
list_t faulty_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;
}
```

Correct and efficient implementation of `uncons_all()`

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

The first example, fixed

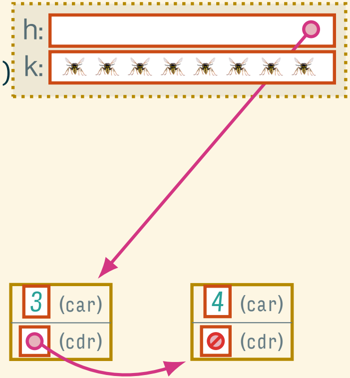
```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));

    printf("%d\n", first(h));
    list_t k = uncons_one(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```

The first example, fixed

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));

    printf("%d\n", first(h));
    list_t k = uncons_one(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```

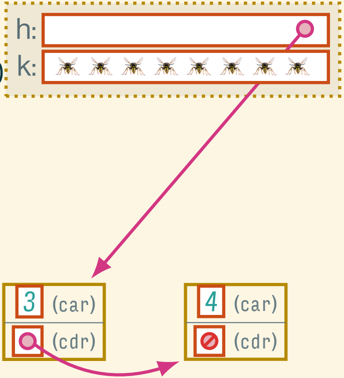
The first example, fixed

```c
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));

    printf("%d\n", first(h));
    list_t k = uncons_one(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```



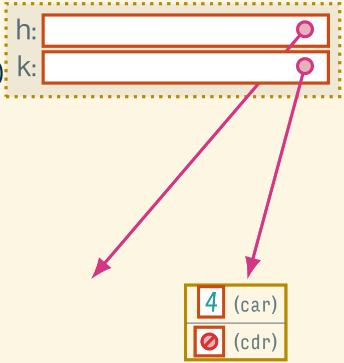
The first example, fixed

```
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));

    printf("%d\n", first(h));  // 3⏎
    list_t k = uncons_one(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```



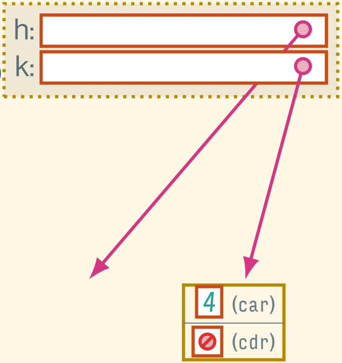
The first example, fixed

```
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty));

    printf("%d\n", first(h));  // 3⏎
    list_t k = uncons_one(h);
    printf("%d\n", first(k));
    uncons_all(k);
}
```



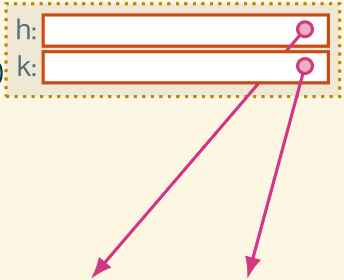
The first example, fixed

```
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty)

    printf("%d\n", first(h));  // 3⏎
    list_t k = uncons_one(h);
    printf("%d\n", first(k));  // 4⏎
    uncons_all(k);
}
```



The first example, fixed

```
#include "cons.h"

int main(void)
{
    list_t h = cons(3, cons(4, empty)

    printf("%d\n", first(h));  // 3⏎
    list_t k = uncons_one(h);
    printf("%d\n", first(k));  // 4⏎
    uncons_all(k);
}
```



Ownership: Major points

- Every heap object has an owner.

Ownership: Major points

- Every heap object has an owner.
- Owners *can and must* either free the objects they own, or transfer ownership.

Ownership: Major points

- Every heap object has an owner.
- Owners *can and must* either free the objects they own, or transfer ownership.
- Non-owners *must not* free the objects they don't own.

Ownership: Major points

- Every heap object has an owner.
- Owners *can and must* either free the objects they own, or transfer ownership.
- Non-owners *must not* free the objects they don't own.
- Ownership is imaginary.

Up next

To Emacs!

– Next time: C Wrap-Up –

Notes

* Lies