University of Waterloo
David R. Cheriton
School of Computer Science

Unit 5 – Linked Data Structures

The C language does not provide the convenient "growing" data structures that are given by many other languages, such as Racket, Python, etc. Instead you must build them from scratch, and manage the memory yourself. This is tricky, and fraught with many dangers.

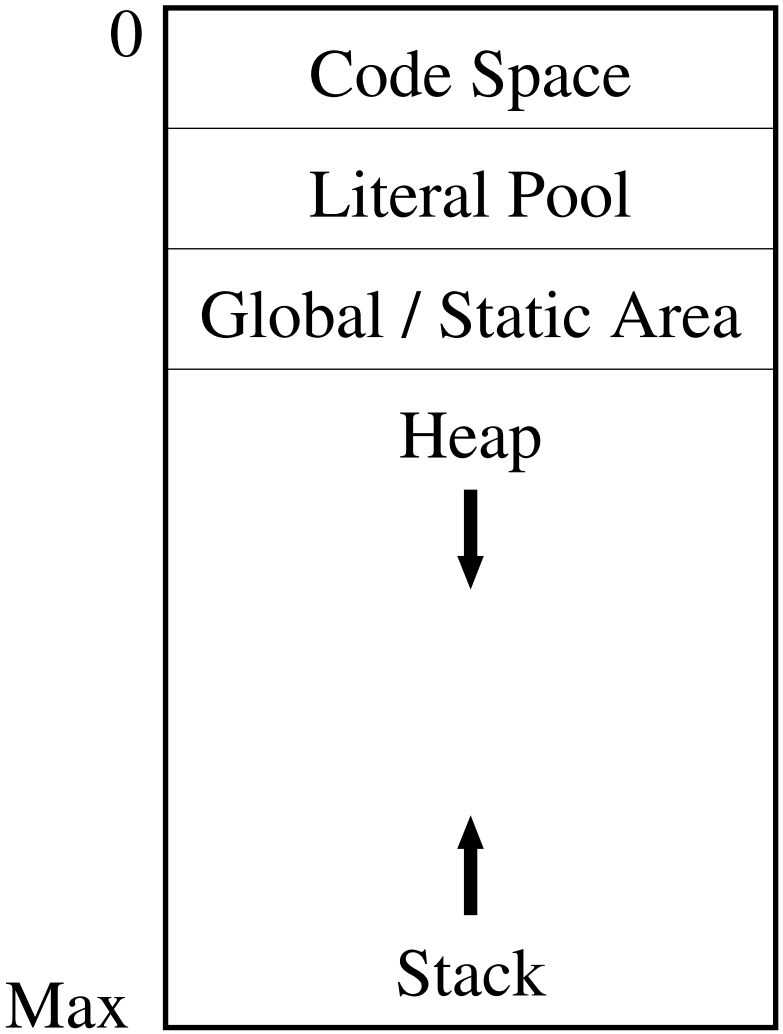
We examine the tools C provides and techniques to implement our own Abstract Data Types that dynamically allocate memory as needed.

Dynamic Memory

So far in our C memory model, all of our data has been stored in either the global/static region or the stack. The global/static region contains global variables and static local variables and can be determined before the program runs. The stack is automatically used to store the stack frames whenever a function is called.

The next region of memory we will explore is known as "the heap" and is designed to provide *dynamic* memory. The memory is considered "dynamic" because it is requested while a program is running and is not known in advance.

An example of dynamic memory in Racket is when we store an arbitrary number of items in a `list`.


0
Code Space
Literal Pool
Global / Static Area
Heap
Stack
Max

The Heap

The name of *"the heap"* is unfortunate. Early CS pioneers (such as Knuth) tried to change the name, but it became too popular and we are now stuck with it.

Some confusion arises because there is also a popular data structure known as a "heap". There is no relation between the two concepts (at least the stack area of memory closely resembles the stack ADT).

For now, simply imagine the heap as a "big pile" of available memory.

Conceptually, you request some memory from the heap when you need it, and then give it back to the heap when you are done with it.

Dynamic Memory in Racket

We did not discuss how Racket obtains memory dynamically for lists, but it is built into the `cons` function, which obtains memory from Racket's *"heap"*.

To introduce dynamic memory in C, we will emulate the behaviour of Racket's `cons` function to build racket-like lists.

First, we will see how we can obtain memory from the heap in C.

The Heap: `malloc`

The function `malloc(s)` obtains `s` bytes of (contiguous) memory from the heap and returns a pointer to the block of memory.

- It's an abbreviation of **m**emory **alloc**ation
- The signature of `malloc` has two things we haven't seen yet:

```
void *malloc(size_t s);
```

- A `void` pointer (`void *`) is a pointer to an "unknown type", and is used here because `malloc` doesn't know (or care) what type of memory you want
  - A `void` pointer (pointer to "unknown type") is different than a `NULL` pointer (pointer to "invalid memory")
- Eventually, the heap will run out of memory, and `malloc` will return a `NULL` pointer
  - You should always check to see if `malloc` returned a `NULL` pointer and gracefully handle the situation instead of crashing

The Heap: `malloc`

```
void *malloc(size_t s);
```

- `size_t` is a special type related to the C keyword `sizeof`
- `sizeof(type)` is the number of bytes required to store `type`
- For example, if you require dynamic memory for an `int`, you would use `malloc(sizeof(int))`
  - Even if you know that an `int` is 4 bytes, you should use `malloc(sizeof(int))` instead of `malloc(4)`, as it will make your code more *portable* and correct on architectures with different sized `int`s
- You *must* use `sizeof` with *structures*
  - `malloc(sizeof(struct posn))` is the proper way to obtain memory for a `struct posn`
  - You shouldn't assume that you know how big a structure is

The Heap: `free`

For every `malloc`, you should eventually `free` (when the memory is no longer needed).

```
void free(void *ptr);
```

- `free(ptr)` releases the block of memory pointed to by `ptr`
- The `free`d block of memory can be *reused* in the future by subsequent calls to `malloc`
- `ptr` **must** point to a block of memory previously allocated by a call to `malloc`
- Do not call `free` more than once for the same `malloc`
- Once a block of memory has been `free`d, you can no longer access that memory again
- **For every `malloc`, you should `free`**, otherwise your program will have *memory leaks*
- Our RunC environment will enforce that you `free` all memory that you `malloc`

```c
#include <stdlib.h>  // malloc and free are in stdlib.h
#include <stdio.h>

int main(void) {
  int *ptr;

  ptr = malloc(sizeof(int));

  *ptr = 7;
  printf("The address of ptr is: %p\n",              &ptr);
  printf("The value of ptr is: %p\n",                 ptr);
  printf("The value of what ptr points at is: %d\n", *ptr);

  free(ptr);

  return 0;
}
```

```
The address of ptr is: 0xbfca12e0
The value of ptr is: 0xaff98f80
The value of what ptr points at is: 7
```

Memory Leaks

A *memory leak* occurs whenever allocated memory is not eventually `free`d. Wasteful programs that continue to leak memory may eventually suffer degraded performance or crash.

```
#include <stdlib.h>
#include <stdio.h>

int main(void) {
  int *ptr;

  ptr = malloc(sizeof(int));

  ptr = malloc(sizeof(int)); // Memory Leak!

  //...
}
```

In the above example, it's now impossible to `free` the original `malloc` because we no longer have the address, and so that memory can never be reclaimed.

Memory is Invalid after Free

Once a `free` occurs, one or more pointer variables may still contain the address of the memory that was `free`d.

Any attempt to read from or write to that memory is invalid, and can cause errors or unpredictable results.

```
int *ptr = malloc(sizeof(int));
//...
free(ptr);
j = *ptr;    // INVALID
*ptr = k;    // INVALID
free(ptr);   // INVALID
```

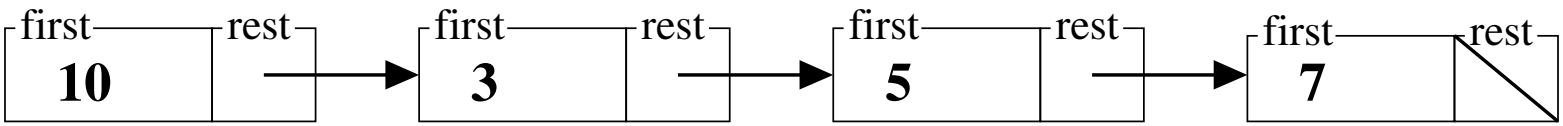
It is often good practice to set a pointer variable to `NULL` after `free` to avoid misuse.

```
free(ptr);
ptr = NULL;
```

Lists in Racket

In CS 135 we learned how to construct lists:

```
(cons 10 (cons 3 (cons 5 (cons 7 empty))))
```



If we didn't have the built-in `cons` function, we could have achieved a similar result with a structure:

```
(define-struct structcons (frst rst))

(define (mycons f r)
  (make-structcons f r))
```

We can mimic this approach to get Racket-like lists in C.

linked list structure

In C, racket-like lists are known as *linked lists*, and each "part" of the list is called a *node*.

For now, all of our linked lists will contain only integers (type `int`).

To mimic the Racket naming conventions, we will *recursively* define each **node** in our linked list as:

```
struct node {
   int first;
   struct node *rest;  // recursive definition
};
```

Each node has an item (`first`) and a **pointer** to the `rest` of the nodes. To represent a linked list, we have a variable that is a pointer to the first node:

```
struct node *mylist;
```

To represent an `empty` linked list, we will use a `NULL` pointer:

```
mylist = NULL;
```

linked list: cons

To continue with our Racket naming conventions, we will create a `cons` function, which will *dynamically* allocate space for a new node:

```
struct node *cons(int item, struct node *lst) {

    struct node *newnode = malloc(sizeof(struct node));

    newnode->first = item;
    newnode->rest = lst;
    return newnode;
}
```

Our C `cons` function mimics the Racket `cons` function: the first parameter is an item (integer), and the second is a linked list (a pointer to a node).

cons: error handling

Earlier, we mentioned that you should always check the value that `malloc` returns to ensure that it was successful. Our updated version of `cons` will perform this check:

```
struct node *cons(int item, struct node *list) {

  struct node *newnode = malloc(sizeof(struct node));

  if (newnode == NULL) {
    printf("ERROR! cons ran out of memory!\n");
    exit(EXIT_FAILURE);
  }

  newnode->first = item;
  newnode->rest = list;
  return newnode;
}
```

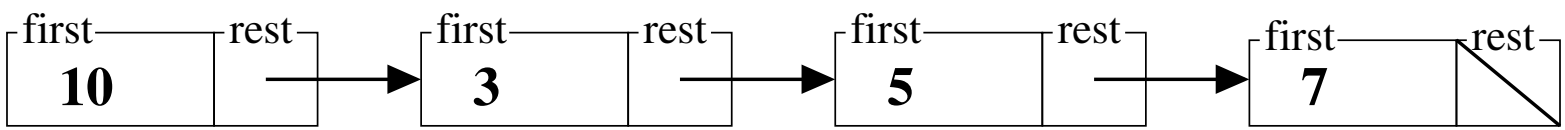
`exit` is a special function that terminates the program. It is similar to the `return` in `main`. `EXIT_FAILURE` is defined in `stdlib.h` and indicates to the OS that an error has occurred.

linked list: using `cons`

We can use our new C `cons` function the same way we used our Racket `cons` function:

```
int main(void) {

  struct node *mylist;

  mylist = cons(10, cons(3, cons(5, cons(7, NULL))));
  //...
}
```

The same box and pointer diagrams we used in CS 135 clearly illustrate why our lists are called *linked lists*, as each node has a *link* to the next node.



Returning Pointers

When we first introduced pointers, we cautioned against having a function return the address of a local variable:

```
int *very_bad(int i) {
  int j = 10;
  return &j;  // NEVER do this!
}
```

At first glance, it may seem that we are violating this with `cons`:

```
struct node *cons(...) {
  struct node *newnode = malloc(sizeof(struct node));
  //...
  return newnode;
}
```

The difference is that `cons` returns the *value* of `newnode` (not its address), which is an address from *the heap* (not the stack).

The ability to have a function (such as `cons`) create memory that exists after the function has `return`ed (beyond the function's *scope*) is one of the benefits of using dynamic memory.

Length of a Linked List

We can now write functions that use linked lists. We can write recursive versions (like we would in Racket), but iterative functions are typically more common in C.

```
int length_recursive(struct node *lst) {
  if (lst == NULL) {
    return 0;
  } else {
    return 1 + length_recursive(lst->rest);
  }
}

int length_iterative(struct node *lst) {
  int length = 0;
  while (lst != NULL) {
    length += 1;
    lst = lst->rest;
  }
  return length;
}
```

A Linked List “template”

The iterative approach in the previous slide is very common. If we wanted to build a “template” as we did in CS 135, the basic template would look like:

```
while (lst != NULL) {
  // body
  lst = lst->rest;
}
```

Since `NULL` is zero, and all non-zero values are considered true, it is also common to see:

```
while (lst) {
  // body
  lst = lst->rest;
}

for (; lst; lst = lst->rest) {
  // body
}
```

Destroying a List: A Common Mistake

We need to `free` or "destroy" a linked list when we no longer need it. The following code highlights a common linked list mistake:

```
void destroy_list_with_a_mistake(struct node *lst) {
  while (lst != NULL) {
    free(lst);
    lst = lst->rest;    // INVALID
  }
}
```

We cannot follow our "template" too closely, because after we `free` the first node, we cannot access that node to determine the address of the rest (or the "next" node).

In our RunC environment, this invalid access is detected and will produce an error. However, in some C environments this may not cause an error and might work 99.9% of the time, but occasionally it will use invalid memory and cause very unpredictable results.

Destroying a List

This implementation of `destroy_list` fixes the problem by making a "backup" pointer variable of the `next` node before each node is `free`d:

```
void destroy_list(struct node *lst) {
  struct node *next;
  while (lst != NULL) {
    next = lst->rest;
    free(lst);
    lst = next;
  }
}
```

We can also implement a straightforward recursive version:

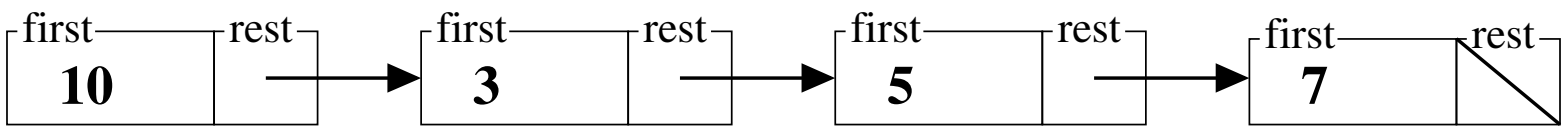
```
void destroy_list_recursive(struct node *lst) {
  if (lst != NULL) {
    destroy_list_recursive(lst->rest);
    free(lst);
  }
}
```

Caution Destroying Lists

We have to be careful when we destroy our lists to ensure that we `free` each memory location only once. Consider the following example:

```
struct node *a = cons(7,NULL);
struct node *b = cons(5,a);
struct node *c = cons(3,b);
struct node *d = cons(10,c);

destroy_list(d);
```

`destroy_list(d)` is the *only* way to free all of the memory properly. If `destroy_list(a)` is called first, it will `free` the node with `7` in it, but then the node with `5` in it will be pointing to an invalid (`free`d) memory location.



Garbage Collection

In Racket, we were not concerned with requesting memory dynamically or `free`ing memory when it was no longer needed.

That is because Racket uses what is known as a *garbage collector*. Most modern languages use a garbage collector.

With a garbage collector, the programmer does not need to `free` any dynamically allocated memory. The garbage collector will *automatically* detect when memory is no longer needed and return the memory to the heap.

One of the few disadvantages of garbage collection is that it does take time, and can affect performance. However, that is typically only an issue in very high performance computing, and in most environments the benefits of garbage collection outweigh any disadvantages.

Garbage Collection in Racket

To illustrate garbage collection in Racket, we will use a very inefficient implementation of the `reverse` function. One reason this function is inefficient is because it will generate a lot of garbage (each `append` in the recursion will generate a new list):

```
(define (my-reverse lst)
  (cond [(empty? lst) empty]
        [else (append (my-reverse (rest lst))
                      (cons (first lst) empty))]))
```

The Racket `time` function will report how much CPU time was used, how much "real time" (also known as wall-clock time) was used and how much time was used by garbage collection (gc). In this example, most of the time was used for garbage collection:

```
> (time (length (my-reverse (build-list 10000 add1))))
cpu time: 2531 real time: 2537 gc time: 2051
```

Introduction to Tail Recursion

Consider the following two implementations of `sumfirst`:

```
(define (sumfirst n)
  (cond [(zero? n) 0]
        [else (+ n (sumfirst (sub1 n)))]))

(define (sumfirst/accumulative n)
  (define (sum/acc n sofar)
    (cond [(zero? n) sofar]
          [else (sum/acc (sub1 n) (+ n sofar))]))
  (sum/acc n 0))
```

We might expect that the two implementations would take the same amount of time:

- Both functions are calculating exactly the same value
- Both algorithms require O(n) steps for the calculation

Introduction to Tail Recursion

```
(define (sumfirst n)
  (cond [(zero? n) 0]
        [else (+ n (sumfirst (sub1 n)))]))

(define (sumfirst/accumulative n)
  (define (sum/acc n sofar)
    (cond [(zero? n) sofar]
          [else (sum/acc (sub1 n) (+ n sofar))]))
  (sum/acc n 0))
```

However, the accumulative implementation is significantly faster:

```
(time (sumfirst 1000000))
  cpu time: 3281 real time: 3285 gc time: 2689

(time (sumfirst/accumulative 1000000))
  cpu time: 63 real time: 70 gc time: 0
```

The most significant difference is the garbage collection (gc) time.

Tail Recursion: Substitution Steps

In the `sumfirst` function, the substitution steps yield:

```
(sumfirst 1000000)
=> (+ 1000000 (sumfirst 999999))
=> (+ 1000000 (+ 999999 (sumfirst 999998))
```

After each recursion, the *size* of the expression continues to grow, which requires more memory. However, for the `sumfirst/accumulative` function, the size of the expression does not grow:

```
(sumfirst/accumulative 1000000)
=> (sum/acc 1000000 0)
=> (sum/acc 999999 1000000)
=> (sum/acc 999998 1999999)
```

The difference is that `sumfirst/accumulative` is *tail recursive*.

Tail Recursion

A Racket function is tail recursive if the recursion is always the **last expression** to be evaluated (the "tail"). Typically, this is achieved by using accumulative recursion and providing a partial result as one of the parameters.

Tail recursion is not just important in Racket. Most modern C environments will detect tail recursion as well. Tail recursion in C occurs when the recursive call is the `return` value, or (for `void` functions) if the recursive call is the last statement executed.

In C, a significant advantage of tail recursion is that it may not require a separate stack frame for each recursive call. When the local variables and parameters are no longer required, C may "re-use" or overwrite the same stack frame. This can be more efficient and avoid overflowing the stack.

Functions that `return` Linked Lists

In Racket, we wrote many functions that consumed a list and produced a new list. Consider the following example:

```
(define (sqr-list lst)
  (cond [(empty? lst) empty]
        [else (cons (sqr (first lst))
                    (sqr-list (rest lst)))]))
```

we can write the same recursive function in C:

```
struct node *sqr_list_recursive(struct node *lst) {
  if (lst == NULL) {
    return NULL;
  } else {
    return cons(lst->first * lst->first,
                sqr_list_recursive(lst->rest));
  }
}
```

Iterative functions that `return` Linked Lists

The *iterative* version of `sqr_list` is more awkward:

```
struct node *sqr_list_iterative(struct node *lst) {
  struct node *head; // head of new list
  struct node *tail; // tail of new list
  if (lst == NULL) {
    return NULL;
  }
  // set up the first node of the new list
  head = cons(lst->first * lst->first, NULL);
  tail = head;
  lst = lst->rest;
  // iterate over the rest of the list
  while (lst != NULL) {
    tail->rest = cons(lst->first * lst->first, NULL);
    tail = tail->rest;
    lst = lst->rest;
  }
  return head;
}
```

Mutating Linked Lists

In Racket, it was natural to construct a **new** list, but in C it's much more common to **mutate** an existing linked list. When using mutation, the iterative solution is *much* more straightforward:

```
void sqr_list_mutation(struct node *lst) {
  while (lst != NULL) {
    lst->first = lst->first * lst->first;
    lst = lst->rest;
  }
}
```

If you only have a list function that mutates, and you don't want to change your original list, you can simply create a copy of a list and then mutate the copy.

```
struct node *newlist = list_copy(lst);
sqr_list_mutation(newlist);
```

As an exercise, try creating a `list_copy` function (both a recursive and iterative version).

Abstract List Function: map

Earlier, we provided a recursive Racket function `sqr-list` to square the elements of a list of numbers.

```
(define (sqr-list lst)
  (cond [(empty? lst) empty]
        [else (cons (sqr (first lst))
                    (sqr-list (rest lst)))]))
```

It's much easier to use the abstract list function `map`:

```
(define (sqr-list lst)
  (map sqr lst))
```

The first parameter of `map` is another function (`sqr` in the above example).

Function Values in C

In Racket, functions were *first-class values*.

In C, a function is more like a “block of code”.

If you try to obtain the “value” of a C function, C will instead give you the address of where that function’s block of code resides in memory:

```
#include <stdio.h>

int main(void) {

  printf("The value of main is: %p\n", main);
  printf("The address of main is: %p\n", &main);

  return 0;
}
```

```
The value of main is: 0x804d2b0
The address of main is: 0x804d2b0
```

Function Addresses in C

Surprisingly, in C code `main` and `&main` are equivalent.

Since C cannot produce the "value" of a C function, C will automatically interpret `main` as `&main`.

The choice of notation is a matter of style and debate.

We will use the `main` notation as it is cleaner, but `&main` is common in practice because it reminds the reader that it is an address.

Function Pointers

In C, a variable can be a *pointer to a function*.

For a function that returns an `int` and has one `int` parameter:

```
int add1(int i) {
  return i + 1;
}
```

the declaration for a pointer to such a function would be:

```
int (*fp)(int);
```

The following code illustrates how it could be used:

```
int main(void) {
  int j;
  int (*fp)(int);
  fp = add1;        // or alternatively, fp = &add1;
  j = fp(5);
  //...
}
```

Function Pointer Declarations

To understand function pointer declarations, it's best to see a few more examples:

```
int functionA(int i) {...}
int (*fpA)(int) = functionA;

void functionB(int j) {...}
void (*fpB)(int) = functionB;

int functionC(int i, int j) {...}
int (*fpC)(int, int) = functionC;

int *functionD(int *ptr, int i) {...}
int *(*fpD)(int *, int) = functionD;

struct posn *functionE(struct posn *p, int i) {...}
struct posn *(*fpE)(struct posn *, int) = functionE;
```

```c
#include <stdio.h>

int sqr(int i) {
  return i*i;
}

int add1(int i) {
  return i+1;
}

int main(void) {

  int (*fp)(int);

  fp = sqr;
  printf("fp(5) = %d\n", fp(5));
  fp = add1;
  printf("fp(5) = %d\n", fp(5));
  return 0;
}
```

```
fp(5) = 25
fp(5) = 6
```

map in C

We can also pass function pointers as parameters to functions, and so now we have all of the necessary components to construct a `map` function in C (using mutation):

```
void map(int (*fp)(int), struct node *lst) {
  while (lst != NULL) {
    lst->first = fp(lst->first);
    lst = lst->rest;
  }
}

int sqr(int i) { return i * i; }
int add1(int i) { return i + 1;}

int main(void) {
  struct node *mylist;
  mylist = cons(10, cons(3, cons(5, cons(7, NULL))));
  // mylist contains: 10 -> 3 -> 5 -> 7
  map(sqr, mylist);
  // mylist contains: 100 -> 9 -> 25 -> 49
  map(add1, mylist);
  // mylist contains: 101 -> 10 -> 26 -> 50
}
```

filter in C

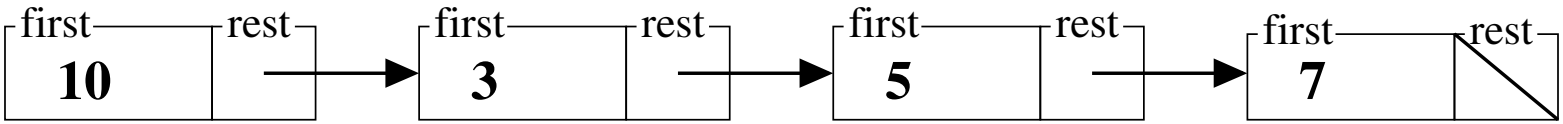
We can also construct a `filter` function:

```
#include <stdbool.h>

void filter(bool (*fkeep)(int), struct node *lst) {
  //... implementation not shown
}

bool even(int i) {
  return (i%2) == 0;
}

int main(void) {
  struct node *mylist;
  mylist = cons(10, cons(3, cons(5, cons(7, NULL))));
  // mylist contains: 10 -> 3 -> 5 -> 7
  filter(even, mylist);
  // mylist contains: 10
}
```

Can you spot the flaw in this design?

What if it was `filter(odd, mylist)`?

Mutating Lists




```
void filter(bool (*fkeep)(int), struct node *lst) {...}

//...

  mylist = cons(10, cons(3, cons(5, cons(7, NULL))));
  filter(odd, mylist);
```

In the above example, after we execute `filter(odd, mylist)` we would expect `mylist` to point to the 3 node.

As another example, if we executed `filter(negative, mylist)` (only keeping negative numbers), we would expect `mylist` to be `NULL` (empty list).

Now can you spot the flaw in this design?

Changing the First Node in a List

```
void filter(bool (*fkeep)(int), struct node *lst) {...}

//...

  mylist = cons(10, cons(3, cons(5, cons(7, NULL))));
  filter(odd, mylist);
```

The problem with the above design is that C uses the "pass by value" convention, and so we cannot change the value of a variable when it is passed as a parameter.

`mylist` is a pointer to the first node in the list: *nothing* in `filter` could possibly change the value of `mylist`, and so `mylist` will continue to point to the address of that same first node.

Potential Solutions to the filter Problem

One solution to our `filter` problem is to avoid mutation altogether and always construct a *new* list:

```
newlist = filter(odd, mylist);
```

But that approach is a little too evasive.

Another solution is to always `return` the value of the new head node, and so the usage would always have to be:

```
mylist = filter(odd, mylist);
```

Some programmers prefer this approach, but it can be difficult to communicate this usage to others and since it can't be enforced it can lead to problems.

Previously, to have a function change the value of a variable we passed a *pointer* to the variable. However, our lists are themselves pointers (to nodes), and so if we want to solve this problem we must use *pointers to pointers*.

Pointers to Pointers

Pointers to pointers are really no different than regular pointers. The only difference is the *type* of what they point to. The following examples help illustrate how they are used:

```
int i;          // i is an int
int *pi;        // p is a pointer to an int
int **ppi;      // pp is a pointer to a pointer to an int

i = 5;          // valid
*i = 5;         // INVALID: You can't dereference an int

pi = &i;        // valid
*pi = 5;        // valid
pi = 5;         // INVALID: the value of p must be a pointer

ppi = &pi;      // valid
*ppi = &i;      // valid
*(*ppi) = 5;    // valid
**ppi = 5;      // valid
ppi = &i;       // INVALID: ppi must point to a pointer
*ppi = 5;       // INVALID: the value of ppi must be a pointer
```

Linked Lists: add vs. cons

Now that we have pointers to pointers, we can write functions that can change the first node in a linked list.

We created our C `cons` function to mimic the Racket `cons` function: it is passed an item and a list and returns a "new" list. This approach is not always used in C.

In C, it is often more typical to have an "add" function that is passed a list and an item to be added to the list.

Linked Lists: Add to Front

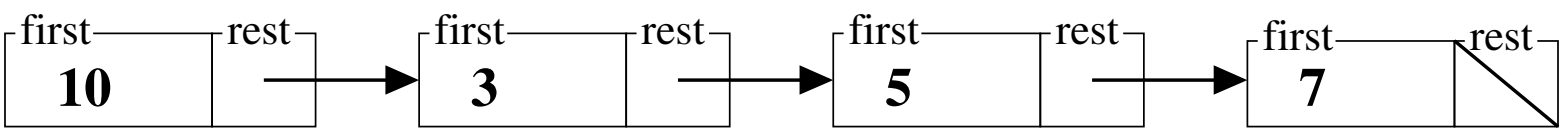
For example, the following function returns `void` and accepts a pointer to a list (which is itself a pointer to a node). We have used the variable name `ptr_to_list` to highlight the difference between it and our `cons` function:

```
void add_to_front(struct node **ptr_to_list, int item) {

  struct node *newnode = malloc(sizeof(struct node));

  if (newnode == NULL) {
    printf("ERROR! add_to_front ran out of memory!\n");
    exit(EXIT_FAILURE);
  }

  newnode->first = item;
  newnode->rest = *ptr_to_list;
  *ptr_to_list = newnode;
}
```

Linked Lists: Using add to front

To use `add_to_front`, we must pass the pointer to (the address of) the list.

```
// recall the signature of add_to_front
void add_to_front(struct node **ptr_to_list, int item);

int main(void) {

  struct node *mylist = NULL; // initialize to an empty list

  add_to_front(&mylist,7);  // note the &
  add_to_front(&mylist,5);
  add_to_front(&mylist,3);
  add_to_front(&mylist,10);
  //...
}
```



Linked Lists: Remove From Front

For the complimentary `remove_from_front`, we will `free` the first node in the list and `return` the item from the first node. This function has the *precondition* that the list is not empty.

```
int remove_from_front(struct node **ptr_to_list) {

  struct node *lst = *ptr_to_list;
  int retval = lst->first;
  *ptr_to_list = lst->rest;
  free(lst);

  return retval;
}
```

filter in C

We can now revisit the `filter` function and write it in C:

```
void filter(bool (*fkeep)(int), struct node **ptrhead) {
  struct node *lst = *ptrhead;
  struct node *prev = NULL;
  struct node *next;
  *ptrhead = NULL; // reset to empty list

  while (lst != NULL) {
    next = lst->rest;
    if (fkeep(lst->first)) {
      if (*ptrhead == NULL) { // first node
        *ptrhead = lst;
      }
      prev = lst;
    } else {
      if (prev != NULL) {
        prev->rest = next; // skip over this node
      }
      free(lst);
    }
    lst = next;
  }
}
```

filter example

```
#include <stdbool.h>

void filter(bool (*fkeep)(int), struct node **ptrhead) {
  // see previous slide
}

bool odd(int i) {
  return (i%2) != 0;
}

int main(void) {
  struct node *mylist = NULL;
  add_to_front(&mylist,7);
  add_to_front(&mylist,5);
  add_to_front(&mylist,3);
  add_to_front(&mylist,10);
  // mylist contains: 10 -> 3 -> 5 -> 7
  filter(odd, &mylist);
  // mylist contains: 3 -> 5 -> 7
  //...
}
```

foldr and foldl in C

Unlike `map` and `filter`, the C equivalents of `foldr/foldl` would not mutate (change) the list. The implementation of `foldr` is naturally recursive:

```
int foldr(int (*fcombine)(int, int),
          int base,
          struct node *lst) {

  if (lst == NULL) {
    return base;
  } else {
    return fcombine(lst->first,
                    foldr(fcombine, base, lst->rest));
  }
}
```

As an exercise, implement `foldl` iteratively.

foldr example in C

The following example shows how `foldr` could be used:

```
int add(int i, int j) {
  return i+j;
}

int sum_list(struct node *lst) {
  return foldr(add, 0, lst);
}
```

Because function generation (`lambda`), lists, `map`, `filter`, and `foldr/foldl` are not built-in features of C, function pointers are not as common as they are in Racket. A C programmer would more likely write `sum_list` as:

```
int sum_list(struct node *lst) {
  int sum = 0;
  while (lst != NULL) {
    sum += lst->first;
    lst = lst->rest;
  }
  return sum;
}
```

The Stack ADT

We will now explore creating ADTs in C, and we will start with the Stack ADT.

Our Stack ADT will use the linked list data structure we are already familiar with:

```
//node.h

struct node {
  int first;
  struct node *rest;
};

void add_to_front(struct node **ptrhead, int item);
int remove_from_front(struct node **ptrhead);
void destroy_list(struct node *lst);
```

Opaque Structures in C

In Racket, we used *structures* in our ADT implementations to take advantage of their *opaque* property to "hide" information from our clients. We will use a similar strategy in C.

In C, all structures are transparent (not opaque) by default. To achieve opaqueness (where only our module can access the fields of a structure) we place the *full* structure declaration in the `"module.c"` file, and we only provide a *partial* declaration in the `"module.h"` file we provide to clients.

```c
// module.c: all fields are known
struct opaque_struct {
  int hidden_field1;
  int hidden_field2;
};

// module.h: only the name is known
struct opaque_struct;
```

Opaque Structures in C

When the client includes `"module.h"`, it only encounters the partial declaration: the client knows the name of the structure type, but it does not know what the fields are.

Because the client does now know the `sizeof` the structure, the client can not declare any variables of the structure type, and can only declare *pointers* to the structure type.

```
// module.h: only the name is known
struct opaque_struct;

// client.c
#include "module.h"

struct opaque_struct *ptr;       // valid
struct opaque_struct mystruct;   // INVALID
```

The Stack Structure

Here is our opaque stack structure:

```
//stack.h
struct stack;

//stack.c
#include "node.h"

struct stack {
  struct node *lst;
};

//client.c
#include "stack.h"
  //...
  struct stack *mystack;
```

To make declarations in the client a little friendlier, we will use the `typedef` C language feature.

typedef

`typedef` is a C language feature that allows you to give alternate names for existing types.

```
typedef int Quantity;
```

After the above statement, you can use `Quantity` just like any other type, and C will know that it's an `int`:

```
Quantity q = 0;
//...
q += 1;
//...
total_cost = item_price * q;
```

This is one method of *abstracting* the type of `Quantity` and making it more portable. If we decide we need a different type to store quantities, we can make the change in one place.

`typedef` simplifications

`typedef`s are often used to simplify long type names such as "`struct stack *`":

```
typedef struct stack *Stack; // C is case sensitive

struct stack *your_stack;    // my_stack and your_stack
Stack my_stack;              // are the same type
```

`typedef`s are *especially* useful when dealing with function pointers. In the following, the type `MapFn` is a pointer to a function that has one `int` parameter and returns an `int`:

```
typedef int (*MapFn)(int);

void your_map(int (*fp)(int), struct node *lst) {...}
void my_map(MapFn fp, struct node *lst)  {...}

int sqr(int i) { return i * i; }

int (*your_fn_ptr)(int) = sqr;
MapFn my_fn_ptr = sqr;
```

Creating a Stack

Because the stack structure is opaque to the client we will need to implement a `new_stack` function and a `delete_stack` function:

```
//stack.c
#include "node.h"
#include "stack.h"

struct stack {
  struct node *lst;
};

Stack new_stack(void) {
  Stack s = malloc(sizeof(struct stack));
  s->lst = NULL;
  return s;
}

void delete_stack(Stack s) {
  destroy_list(s->lst);
  free(s);
}
```

Creating a Stack in the Client

```
//stack.h
struct stack;
typedef struct stack *Stack;

Stack new_stack(void);
// PRE: true
// POST: returns a new, empty Stack

void delete_stack(Stack s);
// PRE: s is a Stack created with new_stack
// POST: s is no longer a valid stack


//client.c
#include "stack.h"
//...
  Stack mystack = new_stack();
  //...
  delete_stack(mystack);
```

Remaining Stack Operations

The following stack operations can be implemented as follows:

```
bool stack_is_empty(Stack s) {
  return s->lst == NULL;
}

void stack_push(Stack s, int item) {
  add_to_front(&s->lst, item);
}

int stack_pop(Stack s) {
  return remove_from_front(&s->lst);
}

int stack_top(Stack s) {
  return s->lst->first;
}
```

One difference between this stack implementation and the one we used in Racket is that this `stack_pop` function returns the value of item popped.

Adding a Length Operation

Now that we have learned how to describe the complexity of a function, it is expected to provide the big-O complexity for each ADT operation. All of the Stack operations are O(1), except for `delete_stack`, which is O(n) (why?).

What if we want to add a `stack_length` operation to our ADT:

```
int stack_length(Stack s) {
  struct node *lst = s->lst;
  int len = 0;
  while (lst != NULL) {
    len += 1;
    lst = lst->rest;
  }
  return len;
}
```

The complexity of the above `stack_length` operation is O(n).

Can we do better?

Augmenting our Stack Structure

One big advantage of using an opaque structure in an ADT implementation is that we can *augment* or *enhance* the structure by adding additional fields. We can even use additional fields to improve the *efficiency* of our ADT operations.

For example, we can add a field that maintains (or "caches") the current length of our stack:

```
struct stack {
  struct node *lst;
  int length; // NEW
};
```

Which operations do we now have to change to support this augmentation?

Augmenting our Stack Structure

```
Stack new_stack(void) {
  //...
  s->length = 0;

void stack_push(Stack s, int item) {
  s->length += 1;
  //...

int stack_pop(Stack s) {
  s->length -= 1;
  //...

int stack_length(Stack s) {
  return s->length;
}
```

This augmentation only added $O(1)$ to the time of each of the operations, and reduced the time of `stack_length` from $O(n)$ to $O(1)$.