Algorithm Design and Data Abstraction

CS 146

Brad Lushman

LaTeXed by Sibelius Peng

Preface

Disclaimer Much of the information on this set of notes is transcribed directly/indirectly from the lectures of CS 146 during Winter 2021 as well as other related resources. I do not make any warranties about the completeness, reliability and accuracy of this set of notes. Use at your own risk.

For any questions, send me an email via https://notes.sibeliusp.com/contact.

You can find my notes for other courses on https://notes.sibeliusp.com/.

Sibelius Peng

Contents

1

Jan 12

1.1 Major themes

Major theme of CS 146

- side-effect ("impurity")
- programs that *do* things
- imperative programming

General outline

- impure Racket
- C
- low-level machine

Why functional programming first? Why not imperative first?

Imperative programming is harder. Side-effects are not easy things to deal with. For example, text is printed to the screen, keystrokes extracted from the keyboard, values of variables change. All these things change the state of the world. Also, the state of the world affects the program.

If we write a racket program like this one,

```
1    (define (f x) (+ x y))
```

That depends on the value of y. However, if the value of y can change because of the side effects, we have to add a word: it depends on *current value* of y.

Thus the semantics of an imperative program must take into account the current state of the world, even while changing the state of the world.

So there is then a temporal component inherent in analysis of imperative programs. It is not "what does this do?", but "what does this do at this point in time?"

Why study imperative programming at all? It seems it doesn't worth it. "The world is imperative". For example, machines work by mutating memory. Even functional programs are eventually executed imperatively.

. . . "or is it?" Is the world constantly mutating, or is is constantly being reinvented? When a character

appears on the screen, does that change the world or create a new one?

Either way, imperative programming matches up with real-world experience, but a functional world view may offer a unique take on side-effects.

1.2 Recursion

Recall from CS 145:

Structural recursion: the structure of the program matches the structure of data.

For example, natural numbers.

```
(define (fact n)                ; A Nat is either
    (if (= n 0) 1               ; 0 or
        (* n (fact (- n 1)))))  ; (+ 1 n) where n is a Nat
```

The cases in the function match the cases in the data definition. The recursive call uses arguments that either stay the same or get one step closer to the base of the data type.

Here is another example on the length of the list.

```
(define (length l)                          ; A (list of X) is empty
    (cond [(empty? l) 0]                    ; or (cons x y) where x
          [else (+ 1 (length (rest l)))]))  ; is an X and y is a (list of X)
```

If the recursion is structural, the structure of the program matches the structure of its correctness by induction.

Claim (length L) produces the length of the list L.

Proof:
Structural induction on L.

Case 1 L is empty. Then (length L) produces 0, which is the length of the empty list.

Case 2 L is `(cons x` L'`)`. Assume that (length L') produces n, which is the length of L'. Then (length L) produces `(+ 1 n)`, which is the length of `(cons x` L'`)`. □

Correctness proof just looks like a restatement of the program itself.

Accumulative recursion one ore more extra parameters that "grow" while the other parameters "shrink".

For example,

```
(define (sum-list L)
    (define (sum-list-help L acc)
        (cond [(empty? L) acc]
              [else (sum-list-help (rest L) (+ (first L) acc))]))
    (sum-list-help L 0))
```

Proof method: induction on an invariant. For example, to prove that `(sum-list L)` sums `L`, suffices to prove `(sum-list-help L 0)` produces the sum of `L`. Let's try to prove by structural induction on `L`.

Case 1 `L` is empty. Then `(sum-list-help L 0)` is `(sum-list-help empty 0)` which gives 0.

Case 2 L = `(cons x` L'`)`. Assume `(sum-list-help` L' `0)` $\Rightarrow$ the sum of L'. Then `(sum-list-help L 0)` is `(sum-list-help (cons x` L'`) 0)` which reduces to `(sum-list-help` L' `(+ x 0))` which is then equal to `(sum-list-help` L' `x)`. Then we are in trouble, because this does not match inductive hypothesis. Proof fails.

So we need a stronger statement about the relationship between `L + acc` that holds throughout the recursion - an invariant.

> Proof:
> We prove the invariant $\forall$ `L`, $\forall$ `acc (sum-list-help L acc)` produces `acc + (sum-list L)` by structural induction on `L`.
>
> **Case 1** `L` is empty. Then `(sum-list-help L acc)` is `(sum-list-help empty acc)` which gives `acc`, which is equal to the sum of the list + `acc`.
>
> **Case 2** `L` is `(cons x` L'`)`. Assume `(sum-list-help` L' `acc)` produces the sum of L' + `acc`. Then `(sum-list-help L acc)` = `(sum-list-help (cons x` L'`) acc)` $\rightsquigarrow$ `(sum-list-help` L' `(+ x acc))` which is equal to `(sum-list` L'`) + (x + acc)` = `(+ (sum-list` L'`) x) + acc` = `(sum-list L) + acc`
>
> Then let `acc = 0`: `(sum-list-help L 0) = (sum-list L)`. □

General recursion: does not follow the structure of the data. Proofs require more creativity.

How do we reason about imperative programs?

1.3 Impure Racket

```
(begin exp_1 ... exp_n)
```

evaluates all of `exp_1, ..., exp_n` in left-to-right order and produces the value of `exp_n`. This is useless in a pure functional setting, but it is useful if `exp_1, ..., exp_(n-1)` are evaluated for their side-effects.

There is an implicit `begin` in the bodies of functions, `lambdas`, `local`, answers of `cond/match`. For example,

```
(define (f x)
    ... ; side-effect 1
    ... ; side-effect 2
    ... ; side-effect 3
    ans
)
```

Reasoning about side-effects: for pure functional programming, we have the substitution model, so-called "stepping rules". Can the substitution model be adapted? we can have the "state of the world" an extra input & extra output at each step. So each reduction step transforms the program & also the "state of the world".

How do we model the "state of the world"? For the simple case, it is just a list of definitions. For more complex cases, we need some kind of memory model (RAM) (won't use yet).

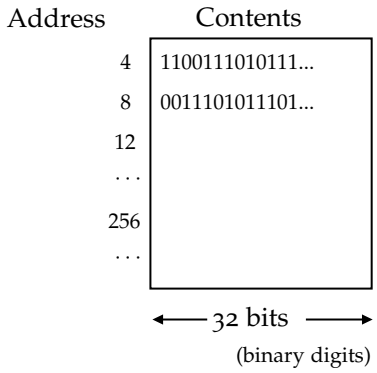
2

Jan 14

2.1 RAM

For now: conceptualization of a RAM (random access machine). Memory is a sequence of "boxes", which are indexed by natural numbers ("addresses"). It contains a fixed size number (say 8 bits or 32 bits). Any box's contains can be fetched $O(1)$ time.

For example, 32-bit RAM:



Will use in a later module, but keep it in mind.

2.2 Modelling Output

It is the simplest kind of side-effect. The "state of the world" here is the sequence of characters that have been printed to the screen. So each step of computation potentially adds characters to this sequence.

> **Note:**
> Every string is just a sequence of characters. Indeed, there is a racket function:
>
> ```
> (string->list "abcd") ===> (list #\a #\b #\c #\d)
> ```

Substitution model $\pi_0 \Rightarrow \pi_1 \Rightarrow \pi_2 \Rightarrow \cdots \Rightarrow \pi_n$ where each π_i is a version of the program obtained by applying one reduction step to π_{i-1}.

In addition to this sequence of programs, now also: $\omega_0 \Rightarrow \omega_1 \Rightarrow \omega_1 \Rightarrow \cdots \Rightarrow \omega_n$ where each ω_i is a

version of the output sequence. Because the sequence of characters can only grow, each ω_i is a *prefix* of ω_{i+1} (can't "unprint" characters).

Therefore, we have a combined version: $(\pi_0, \omega_0) \Rightarrow (\pi_1, \omega_1) \Rightarrow \cdots \Rightarrow (\pi_n, \omega_n)$.

Some program reductions will create definitions, (e.g., `local`), and these defined values will eventually change. So let's separate out the sequence of definitions δ.

So we got a triple now: $(\pi_0, \delta_0, \omega_0) \Rightarrow (\pi_1, \delta_1, \omega_1) \Rightarrow \cdots \Rightarrow (\pi_n, \delta_n, \omega_n)$ where δ_0, ω_0, representing the beginning of the program, are empty.

If $\pi_0 =$ `(define id exp) ...`, then we reduce `exp` according to the usual CS 145 (& new CS 146) rules. This may cause characters to be sent to ω. Now `exp` is reduced to `val`. Then remove `(define id val)` from π and add to δ.

If $\pi_0 =$ `exp ...`, then we reduce `exp` by the usual rules, which may cause characters to be sent to ω. Now `exp` is reduced to `val` which is removed from π. So the characters that make up `val` added to ω.

When π is empty, then we are done. So δ, ω is the *state*, that which changes, other than the program itself. ω here is relatively harmless because changes to ω don't affect the running of the program. What about δ? δ is not a problem yet, because variables are not yet changing. All we are doing now is adding new definitions, which is not really a change of state.

How can we affect ω? In Racket, we can do

- `(display x)` which outputs the value of x with no line break
- `(newline)` gives the line break.
- `(printf "The answer is ~a.\n" x)` which is formatted print. The value of `x` replace `~a`. And `\n` is the new line character. As a Racket character on its own: `#\newline`.

In Racket,

```
> (display "Hello")
Hello
> "Hello"
"Hello"
> (begin (display "Hello") 5)
Hello5
> (define x (begin (display "Hello") 5))
Hello
> x
5
```

But then, what do `display`, `newline`, `printf` return? It looks that they don't return anything. We can try following:

```
> (define y (display "Hello"))
Hello
> y          ; y does have a value
> (list y)   ; by a trick
'(#<void>)
```

They return special value `#<void>` which is not displayed in DrRacket. Basically, for functions, that essentially return nothing, and also the result of evaluating (`void`). Functions that return void are called *statements* or *commands* and that's where imperative programming gets its name.

Recall: a Racket function `map`. `(map f (list l1 ... ln))` produces `(list (f l1) ... (f ln))`. It's reasonable to ask what if `f` is a statement? The idea: it is needed for side-effects and produces `#<void>`. Then `(map f (list l1 ... ln))` produces `(list #<void> ... #<void>)` which is not useful.

Instead, now consider `for-each`: `(for-each f (list l1 l2 ... ln))` *performs* `(f l1)`, `(f l2)` ... `(f ln)` and *produces* `#<void>`. For example, we can use it as follows:

```
(define (print-with-spaces lst)
    (for-each (lambda (x) (printf "~a   " x)) lst))
```

This will print out each item in the list with spaces in between and will produce `void` at the end rather than a list of `void`'s. Let's write `for-each`:

```
(define (for-each f lst)
    (cond [(empty? lst) (void)]
          [else (f (first lst)) ; implicit begin
                (for-each f (rest lst))]))
```

or using `if`:

```
(define (for-each f lst)
    (if (empty? lst)
        (void)
        (begin (f (first lst)) (for-each f (rest lst)))))
```

Doing nothing in one case of an `if` condition is common enough that there is a specialized form:

```
(define (for-each f lst)
    (unless (empty? lst) (f (first lst)) (for-each f (rest lst)))) ; implicit begin
```

It evaluates body expressions if the test is false. Similarly, `(when ...)` evaluates body expressions if test is true.

Before we had output, the order of operations didn't matter (assuming no crashes/non-terminations), but now, the order of evaluation may affect the order of output. Also, before we had output, all non-terminating programs could be considered equivalent (not meaningful), but now non-terminating programs can do interesting things (e.g., print the digits of π).

Semantic model should include the possibility of non-terminating programs. What will be the meaning of the non-terminating programs be? It is what the program would produce "in the limit". Here we let Ω to denote the set of possible values of ω, which would include finite & infinite sequences of characters.

But why do we need output? We never used it in CS 145, and Racket has a REPL (Read-Eval-Print-Loop). We can just call functions and see the result. That's what Racket has, but many languages don't have this. Instead, they have compile/link/execute cycle. Under this cycle, the program is translated (by a *compiler*) to a native machine code and then executed from the command line. Then we will only see output if the program prints it. Below is an example of C program.

```
#include <stdio.h>
int main (void) {
    printf("Hello, world!\n");
    return 0;
}
```

Here we have to ask for it if we want something to show up in the screen (line 3).

What about Racket? Here is a use in Racket: tracing program.

```
(define (fact n)
    (printf "fact applied to argument ~a\n" n) ; implicit begin
    (if (= n 0) 1 (* n (fact (- n 1)))))
```

This can aid debugging.

2.3 Modelling input

Let's now talk about the input. We can imagine an infinite sequence consisting of all characters the user will ever press ι. So the model now is $(\pi, \delta, \omega, \iota)$. Every time we need to accept an input character, is the same as removing a character ι.

Here is a small problem: the sequence may *depend* on the output, so the users decide what to input *in response to* what is displayed on the screen. So a more realistic model of input would perhaps not assume all input is available at one.

The alternative: a request for input yields a function consuming one or more characters and producing the next program π, with the user's characters substituted for the read request. For example, a function `(read-line)`, might be modeled as λ `(line) line`. So if user types `"abc"`, as a result of this, we get `"abc"`. Then the entire program reduces to a big "nesting" of input request functions, basically, one function per "prompt". If we supply user input for each prompt, it yields the final result.

Proof techniques for imperative programs will come much later.

2.4 Input in Racket

`(read-line)` produces a string consisting of all characters pressed until the first newline and the string we get does *not* contain the newline.

```
(read-line) ; pops up a little box and lets us to type
Test.
"Test."     ; and get back the string as the result.
> (string->list (read-line)) ; if we type Test.
(list #\T #\e #\s #\t #\.)
```

To read a list of lines, the question then is how do we know when to stop reading? If we look carefully at the box popped up by `(read-line)`, at the end of the box, there is a yellow button, which says "eof" (end of file). When we press that button, it also ends the search for input. "eof" means there is no more input.

```
(define (read-input)
    (define nl (read-line)) ; nl stands for next line
    (cond [(eof-object? nl) empty]
          [else (cons nl (read-input))]))
```

Note that this implementation of `(read-input)` is not tail-recursive.

A more primitive form of input would be `(read-char)` which extracts one character from the input sequence.

3

Jan 19

3.1 More primitive input reading

`read-char` reads one char from the input sequence. Here is a quick demo.

```
> (read-char)
abcde ; type in the box and press enter
#\a
> (read-char)
#\b
> (read-char)
#\c
> (read-char)
#\d
> (read-char)
#\e
> (read-char)
#\newline
> (read-char) ; now ask for new inputs
```

`peek-char` examines the next char in the sequence, without removing it from the sequence. It does read the character, but does not take that from io, or the input stream.

```
(define (my-read-line)
    (define (mrl-h acc)
        (define ch (read-char))
        (cond [(or (eof-object? ch) (char=? ch #\newline)) (list->string (reverse
    acc))]
              [else (mrl-h (cons ch acc))]))
    (mrl-h empty))

; call it by
(my-read-line)
```

Less primitive input: `read` consumes from input (and produces) an S-expression (no matter how many chars or lines it occupies)

```
> (read) ; type abc
```

```
'abc     ; symbol
> (read)
(a b c   ; not closed
de f ghi ; racket not satisfied
)        ; bracket closed
'(a b c de f ghi)
> (read)
(a b (c d e (f)) g)
'(a b (c d e (f)) g)
```

3.2 Writing DrRacket

The next example is that we write DrRacket: Implementing a Racket REPL

```
(define (repl)
    (define exp (read))
    (cond [(eof-object? exp) (void)]
          [else (display (interp (parse exp)))
                (newline)
                (repl)]))
(repl)
```

`parse` figures out what that S-expression means: function/if... `interp` is do it.

Let's write our own version of `read`. Process typically happens in two steps. The first step is **Tokenization**, which converts sequence of raw characters to a sequence of *tokens* (meaningful "words"). For example, left paren, right paren, id, number... Typically, id's start with a letter, nums start with a digit. Because of that, here is a key observation: peeking at the next character tells us what kind of token we will be getting, and what to look for to complete the token. So this is asking us to build the structure: `(struct token (type value))` where type is the kind of token: `'lp`, `'rp`, `'id`, `'num`; and value is the "value" of the token (numeric value, name, etc).

We gonna make a couple of helpers first:

```
(define (token-leftpar? x) (symbol=? (token-type x) 'lp))
(define (token-rightpar? x) (symbol=? (token-type x) 'rp))
```

```
; read-id: -> (listof char)
(define (read-id)
    (define nc (peek-char))
    (if (or (char-alphabetic? nc) (char-numeric? nc))
        (cons (read-char) (read-id))
        empty))
```

```
; read-number: -> (listof char)
(define (read-number)
    (define nc (peek-char))
    (if (char-numeric? nc)
        (cons (read-char) (read-number))
        empty))
```

Here is our main tokenizer:

```
; read-token: -> token
(define (read-token)
    (define fc (read-char))
    (cond
        [(char-whitespace? fc) (read-token)]
        [(char=? fc #\() (token 'lp fc)]
        [(char=? fc #\)) (token 'rp fc)]
        [(char-alphabetic? fc) (token 'id (list->symbol (cons fc (read-id))))]
        [(char-numeric? fc) (token 'id (list->number (cons fc (read-number))))]
        [else (error "lexical error")]))
```

Note that `list->symbol`, `list->number` don't exist, but it's easy to build them.

Step 2 is **parsing**: are the tokens arranged into a sequence that has the structure of an s-exp? if so, then produce the s-exp. Let's first make a helper.

```
; read-list: -> (listof s-exp)
(define (read-list) ; assumes left-par has already been read
    (define tk (read-token))
    (cond
        [(token-rightpar? tk) empty]
        [(token-leftpar? tk) (cons (read-list) (read-list))]
        [else (cons (token-value tk) (read-list))]))
```

All left is to build `read`:

```
; my-read: -> s-exp
(define (my-read)
    (define tk (read-token))
    (if (token-leftpar? tk) (read-list) (token-value tk)))
```

There are some good exercises:

- expand the set of token types, e.g., strings.
- handle other kinds of brackets, `[ ]`, `{ }` which have to match.

What have we lost by accepting input? We lost *referential transparency*: the same expression has the same value whenever it is evaluated. For example, `(f t)` always produces the same value. If we do `(let ((z (f 4))) body)`, then every (free) `z` in body can be replaced by `(f 4)` and vice versa. "equal can be substituted for equals". It is not true anymore! because `(read)` doesn't produce the same value. That makes it harder to reason about programs, where simple algebraic manipulation is no longer possible.

3.3 Intro to C

C is built from expressions, statements, blocks, functions, program.

3.3.1 Expressions

Example of expressions: `1 + 2` uses infix operators. There is a notion of precedence in C unlike racket. Also a function call, `f(7)`, the name comes first. `printf("%d\n", 5)` is also a function call.

Operator precedence follows usual mathematical conventions. For example, `1 + x * y`, multiplication is done first. If we want plus to do first, then we do `(1 + x) * y`.

We can take function call in a larger expression: `3 + f(x, y, z)`. `printf("%d\n", 5)` is a function call, and C substitutes 5 in place of `%d`, which means display as a decimal number. It's natural to ask, what does `printf` produce? It produces the number of characters printed.

3.3.2 Statements

The easiest way to make a statement (command) is to take an expression and put a semicolon at the end. For example, `printf("%d\n", x);`. Here the value produced by the expression is ignored, so expression is used only for its side-effects. Thus we could do `1 + 2;`, which is legal, but useless. Also, in previous lectures, we have seen `return 0;`, which produces the value 0 as the result of this function and control returns immediately to the caller. `;` is an empty statement, which does nothing. Other statement forms to come.

3.3.3 Blocks

Block is a group of statements treated as one statement.

```
{
    stmt 1
    stmt 2
    ...
    stmt n
}
```

We can think this, sort of,

```
(begin stmt 1 ... stmt n)
```

Note that the difference `begin` has a value, which is the value of `stmt n`, and this is not the case in C. Thus this is not a perfect analogy. A better analogy is that we replace `begin` by `void`, then they will get evaluated but the entire thing has `void` value.

3.3.4 Functions

Here is a function.

```
int f(int x, int y) {
    printf("x = %d, y = %d\n", x, y);
    return x + y;
}
```

In racket, this would be roughly equivalent to

```
; f: Num Num -> Num
(define (f x y)
    (printf "x = ~a, y = ~a\n" x y)
    (+ x y))
```

Function call: `f(4, 3)` is an expression, produces 7. `f(4, 3);` is a statement. Thus in racket, it can be viewed as, `(f 4 3)` and `(void (f 4 3))`.

Note that contracts (type signatures) are required and enforced.

3.3.5 Programs

Program itself is a sequence of functions. The starting point is the special function, known as `main`, and it looks like this

```
int main() { // int main(void) {
    ...
    ...
}
```

For example,

```
int main() {
    f(4, 3);
    return 0;
}
// and we got our f defined before
int f(int x, int y) {
    printf("x = %d, y = %d\n", x, y);
    return x + y;
}
```

If we give it to compiler, it won't compile. Why?

4

Jan 21

4.1 Compile C programs

Recall from last lecture:

```c
int main() {
    f(4, 3);
    return 0;
}
// and we got our f defined before
int f(int x, int y) {
    printf("x = %d, y = %d\n", x, y);
    return x + y;
}
```

won't compile. A C program compiled: there is a program called the compiler that translates the program into the binary which is the only language the computer actually speaks and the computer execute this binary code directly: not through "DrC" like in DrRacket, the program runs natively on the machine on its own.

The way to compile: `gcc myfile.c -Wall -o myfile`. Here `-o myfile` is what we want the output program to be called, name of the output. If we don't do this, the default is `a.out`. `-Wall` stands for "Warn all". To run it, `./myfile` where `.` means the current directory. Without specifying the current directory, it won't know where to find the program to run.

Now back to our problem. The compiler will complain: `main` doesn't know what `f` is. C enforces the rule: declaration-before-use: can't use a function/variable/etc... until we tell C about it. C has this rule because C is old, and it uses one-pass compiler.

Solution 1 Put `f` first.

```c
int f(int x, int y) {
    printf("x = %d, y = %d\n", x, y);
    return x + y;
}

int main() {
    f(4, 3);
```

```
    return 0;
}
```

Ok, but... this doesn't always work. We may want a different order just for the aesthetic of the program. Moreover, reordering the programs does more than C asks.

4.2 Declaration vs. Definitions

```
int f(int x, int y) {
    // ...
}
```

is both *declaration* (tells C the function exists) and *definition* (completely constructs the function).

C only requires *declaration* before use. So what we can do instead is

```
int f(int x, int y); // - function prototype or header
                     // - declaration only.
int main() {
    f(4, 3);
    return 0;
}

int f(int x, int y) {                   // this is the function definition
    printf("x = %d, y = %d\n", x, y);   // also solves the mutual recursion problem
    return x + y;
}
```

However, this still doesn't compile. What is `printf`? no declaration for `printf`. If we knew what it was, in theory we could do

```
int printf(---???---);
int f(int x, int y);

int main() {
    f(4, 3);
    return 0;
}

int f(int x, int y) {
    // ...
}
```

Rather than declare every standard library function header before we use it, C provides "header files". So we write

```
#include <stdio.h>

int f(int x, int y);

int main() {
    f(4, 3);
    return 0;
```

```
}

int f(int x, int y) {
    // ...
}
```

`#include` is not part of the C language. Rather it is a directive to the C preprocessor (which runs before the compiler). It's sort of like macro expansion in Racket. `#include <file.h>` means "drop the contents of `file.h` right here". `stdio.h` contains declarations for `printf`/other IO (input output) functions, and it is located in a "standard place". For example, `/usr/include` directory.

Now until this point, the compiler is satisfied. However, still technically incomplete: where is the code that implements `printf`? `printf` was written once, compiled once, and put in a "standard place", for example, `/usr/lib`.

Code for `printf` must be combined with our code. This step is known as "linking", which is done by a linker, and linker runs automatically. It "knows" to link the code for `printf`. If we write our own modules, then we need to tell the linker about them (later).

Let's go back to `main`: we have the returned value 0. To whom am I returning the zero? The operating system. We can type `echo $?` to check the returned value. Typically, 0 usually means OK. Anything > 0 is some kind of error.

Only in the case of `main`, `return` maybe left out, and in that case, 0 is assumed.

4.3 Variables and input in C

Let's talk about variables.

```
int f(int x, int y) {
    int z = x + y;
    int w = 2;
    return z / w;
}
```

Input:

```
#include <stdio.h>
int main() {
    char c = getchar();
    return c;
}
```

Let's try to read in a number.

```
#include <stdio.h>
// like before, we don't care about the negatives at this point.
int getIntHelper(int acc) {
    char c = getchar();
    if (c >= '0' && c <= '9') return getIntHelper(acc * 10 + c - '0');
    else return acc; // "else" keyword is technically not needed here.
}

// An alternative way: ternary operator
```

```c
int getIntHelper(int acc) {
    char c = getchar();
    return (c >= '0' && c <= '9') ? getIntHelper(acc * 10 + c - '0') : acc;
}

int getInt() {
    return getIntHelper(0);
}
```

We got boolean conditions, like `c >= '0'`, `&&` means "and".

```c
if (test) stmt
else stmt // only needed if there is sth to do in the false case.
```

Typically here, stmt will be a block:

```c
if (test) {
    stmt 1
    ...
    stmt n
}
else {
    ...
}
```

It is recommended to put curly brace for the statement(s). Consider the dangling else problem:

```c
if (condition 1)
    if (condition 2)
        stmt 1
else // this ``else'' actually goes to the second ``if''
    stmt 2
```

Don't fool by the indentation. This is actually

```c
if (condition 1) {
    if (condition 2) {
        stmt 1
    }
    else { stmt 2 }
}
```

Conditional operator `? :` (also called the ternary operator). `if else` is a statement while `? :` creates an expression: `a ? b : c` has value `b` if `a` is `true`, has value `c` if `a` is `false`.

Also note that there is no built-in boolean type in C. 0 means false, and non-zero (often 1) means true. We have boolean type, constants `true, false` in `stdbool.h`.

4.4 Characters

are just restricted form of integer.

`int` varies, but typically occupies 32 bits ($\sim 4 \times 10^9$ distinct values). `char` occupies always 8 bits (256 distinct values). `'0'` is the character 0, numerically it is 48. Similarly, `'9'`, numerically 57.

`char c = '0';` is identical to `char c = 48;` etc. Everything in memory is numbers, so each character must have a numerical code that represents it. The code here is known as ASCII code.

To convert a char `c` to its numeric value: `c - '0'` (`c - 48`). Convert a number (0 - 9) to ASCII: `c + '0'`.

Let's take a second look at `getchar`: `char c = getchar();` not match the prototype: `int getchar();` Why `int` if it's supposed to produce a `char`? What if there are no `char`s? (EOF?) If `getchar` returned any character in this case, there would be no way to indicate EOF (every possible returned value denotes a valid character).

If there are no `char`s (EOF), `getchar` produces an `int` can't possibly be a `char` (not in the range 0..255). The constant EOF denotes the value `getchar` produces an eof (often, `EOF = -1`).

Next question: `getInt` burns a character after reading an `int`. Does C has a function like Racket's `peek-char`? No, but it has `ungetc` which stuffs a `char` back into the input stream.

```
1 int peekchar() {
2     int c = getchar();
3     return c == EOF ? EOF : ungetc(c, stdin);
4 }
```

Here we have equality operator `==`. The reason we return EOF here is because we don't want to stuff a `char` if we didn't receive a `char`. `stdin` is the keyboard stream (or redirected). `ungetc` returns the `char` that was stuffed.

5

Jan 26

5.1 Improved `getInt`

An improved `getInt`, one doesn't burn a character.

```
#include <stdio.h>
#include <ctype.h> // character predicates

int getIntHelper(int acc) {
    int c = peekchar();
    return (isdigit(c)) ? getIntHelper(10 * acc + getchar() - '0') : acc; //
    predicate here will be false for non-digits, EOF.
}
```

This is simpler, but not efficient because we call `peekchar` and `getchar`. To be more efficient, we don't need to call `getchar` twice per character:

```
int getIntHelper(int acc) {
    int c = getchar();
    return (isdigit(c)) ? getIntHelper(10 * acc + c - '0') : (ungetc(c, stdin), acc
    );
}
```

comma operator

`a, b` evaluates `a`, then evaluates `b`, result is the value of `b`. It is equivalent `(begin a b)` in racket. Use is sparingly, otherwise it will affect the readability of the code.

What if there is whitespace before we reach the `int`? So we can write a function skip the whitespace, but we don't want it return anything.

```
void skipws() {
    int c = getchar();
    if (isspace(c)) {
        skipws();
    }
    else ungetc(c, stdin);
}
```

Here is our first example of a `void` function. The idea is that it returns nothing, therefore, cannot

be used in an expression. For example, `void x = skipws();` is illegal. There are no `void` variables, thus only good for side-effects. To return from `void` functions, either reach the end like in `skipws`, or `return;` with no expression.

With that in place, `getInt` becomes simpler:

```
int getInt() {
    skipws();
    return getIntHelper(0);
}
```

5.2 Mutation (in Racket)

Basic mutation: `set!` which is pronounced as "set bang", instead of impolite way "set" (with the extremely high volume).

```
(define x 3)
(set! x 4)  ; produces (void), changes delta
```

Now x is 4. Note x must have been previously defined. So we can now change the value of a variable. What can we do with that? For example,

```
> (lookup 'Brad)
false
> (add 'Brad 36484)
> (lookup 'Brad)
36484
```

This is not possible in pure Racket because same expression can't produces different results. How do we implement this in impure Racket:

```
(define address-book empty) ; global variable, and is visible throughout the entire
    program
(define (add name number)
    (set! address-book (cons (list name number) address-book)))
```

Global data is good for defining constants to be used repeatedly. *But* not good with mutation because any part of the program could change a global variable, thus it affects the entire program. So we got hidden dependencies between different parts of the program, and therefore, it's harder to reason about the program.

5.2.1 Application: Memoization

Caching: saving the result of a computation to avoid repeating it.

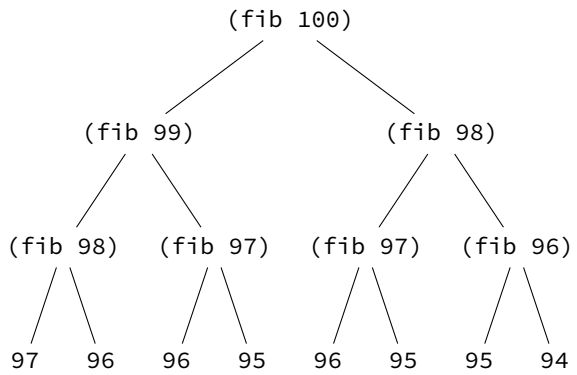
Memoization: maintaining a list or table of cached values.

Consider

```
(define (fib n)
    (cond [(= n 0) 0]
          [(= n 1) 1]
          [else (+ (fib (- n 1))
                   (fib (- n 2)))]))
```

Note that this is inefficient because recursive calls are repeated.

So if want `fib (100)`, it will be expanded as follows:



Note that `(fib 98)` called twice, `(fib 97)` called 3 times, `(fib 96)` called 5 times ... Thus `(fib n)` is $\Theta(F_n) \approx \varphi^n$ where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$. So we can avoid repetition by keeping an association list of paris (n, F_n).

```
(define fib-table empty)

(define (memo-fib n)
    (define result (assoc n fib-table))
    (cond [result => second]
          [else (define fib-n
                    (cond [(<= n 1) n]
                          [else (+ (memo-fib (- n 1))
                                   (memo-fib (- n 2)))]))
                (set! fib-table (cons (list n fib-n) fib-table)) fib-n]))
```

Few notes here.

- `assoc` is a builtin function for association list lookup. In particular, `(assoc x lst)` returns the pair `(x y)` from lst or false if it fails.
- Any value can be used as a test. In racket, `false` is false, anything else is true.
- `(cond [x => f]) ...` if `x` passes (i.e., is not false), then produces `(f x)`.

  `(cond [result => second] ...)` is equivalent to `(cond [(list? result) (second result )] ...)`

Now calls to `(fib n)` now happen only once. But our global variable `fib-table` is accessed by anyone. Can we hide it? Can we arrange that only `memo-fib` has access to this global variable? Here is a way to do it:

```
(define memo-fib
    (local [(define fib-table empty)
            (define (memo-fib n) ...)]
        memo-fib))
```

Equivalently, use `let`:

```
(define memo-fib
    (let ((fib-table empty))
         (lambda (n) ...)))
```

This doesn't quite work for the address-book because we have two functions which need access to it.

5.3 Mutation in C

In C, we have an operator = performing mutation ("assignment operator"). For example,

```c
int main() {
    int x = 3;
    printf("%d\n", x); // 3
    x = 4;
    printf("%d\n", x); // 4
}
```

Note that = is an operator. `x = y` is an expression, thus it has a value as well as an effect: its value is the value assigned. Then `x = 4` sets x to 4, and has value 4. For example,

```c
int main() {
    int x = 3;
    printf("%d\n", x);     // 3
    printf("%d\n", x = 4); // 4
}
```

It really has no advantages... and it has many disadvantages. Because assignment is an expression, C allows us to do `x = y = z = 7;` which sets all of `x, y, z` to 7. Now consider the following,

```c
int main() {
    int x = 5;
    if (x = 4) { // this assigns x to 4, and has value 4, non-zero, true.
        printf("x is 4\n"); // Thus always prints x is 4
    }
    x = 0;
    if (x = 0) { // this assigns x to 0, and has value 0, false.
        printf("x is 0\n"); // Thus never prints x is 0
    }
}
```

It is easy to confuse assignment with equality check: `if (x == 4) ...` Thus usually best to use assignment only as a statement.

One thing we can do is that we can leave variables uninitialized and assign them later. For example,

```c
int main() {
    int x; // uninitialized
    x = 4;
    ...
}
```

This is actually not a good idea. Do only with a good reason. For example,

```c
int x;
if (x == 0) {
    ... // will this run or not?
}
```

The answer to the question above is *we don't know*, because x's value is not known! The value of an uninitialized variable is undefined. Typically, it's whatever value was in the memory from before.

6

Jan 28

6.1 Global variables in C

```c
int c = 0; // global variable

int f() {       // returns 0, then 1, then 2, etc.
    int d = c;
    c = c + 1;
    return d;
}

int main() {
    printf("%d\n", f()); // 0
    printf("%d\n", f()); // 1
    printf("%d\n", f()); // 2
}
```

Be careful:

```c
int main() {
    printf("%d\n%d\n%d\n", f(), f(), f());
}
```

This could produce $\begin{matrix}0\\1\\2\end{matrix}$ or $\begin{matrix}2\\1\\0\end{matrix}$ or others! Order of argument evaluation is *unspecified*.

As with the Racket `fib` example, we can interfere with `f` by mutating `c`.

Can we protect `c` from access by functions other than `f`? Yes by using a magic keyword: `static`.

```c
int f() {
    static int c = 0;
    int d = c;
    c = c + 1;
    return d;
}
```

Here `c` is still a global variable, but it's a global variable that only `f` can see. In terms of variables, there are two notions which we tend to group them together because they are often the same. One is scope, one is extent or lifetime. A traditional global variable has global scope, thus everyone can see it. But more importantly here, its extent: how long it is alive, so it has a global extent. Static variable `c` has a local scope: only `f` can see it, but a global extent: it does not go away when `f` goes away.

6.2 Repetition

Let's say I write a function like this:

```
void sayHiNTimes(int n) {
    if (n > 0) {
        printf("Hi\n");
        sayHiNTimes(n-1);
    }
}
```

This is tail recursion: the recursion call is the last thing the function does. In C, with mutation, we can express this more idiomatically as

```
void sayHiNTimes(int n) {
    while (n > 0) {
        printf("Hi\n");
        n = n - 1;
    }
}
```

This is known as a *loop*, basically shorthand for tail-recursive computation. The body of the loop is executed repeatedly, as long as the condition remains true. In general, if we have

```
void f(int c) {
    if (cont(c)) { // continuation condition
        body(c);
        f(update(c));
    }
}
```

then it becomes

```
void f(int c) {
    while (cont(c)) {
        body(c);
        c = update(c);
    }
}
```

So in the latter version, `f` is not needed, i.e., the things inside may not need to be its own function anymore, if used only once. If we have accumulators, we can still do that.

```
int f(int c, int acc) {
    if (cont(c)) {
        body(c);
        return f(update1(c), update2(c, acc));
    }
```

```
    return g(acc);
}

f(acc, 0);
```

Then it becomes

```
int acc = acc0;
while (cont(c)) {
    body(c);
    acc= update2(c, acc);
    c = update1(c);
}
acc = g(acc);
```

Let's do a concrete example.

```
int getIntHelper(int acc) {
    char c = getchar();
    if (isdigit(c)) {
        return getIntHelper(10*acc+c-'0');
    }
    return acc;
}
int getInt() {
    return getIntHelper(0);
}
```

How might we change it? We can do:

```
int acc = 0;
char c = getchar();
while (isdigit(c)) {
    acc = 10 * acc + c - '0';
    c = getchar();
}
```

which is also shorter. We notice some common patterns, which we can emerge:

```
(initialize variables)
while (condition) {
    (body)
    (update variables)
}
```

It's common to forget the "update step", then we might have infinite loop. There is an alternative format which forces you to do all important things upfront, then much harder to forget them.

```
for (init; condition; update) {
    (body)
}
```

With for loop, we might do

```c
int acc = 0;
char c;
for (c = getchar(); isdigit(c); c = getchar()) {
    acc = 10 * acc + c - '0';
}
```

or even

```c
int acc = 0;
for (char c = getchar(); isdigit(c); c = getchar()) {
    acc = 10 * acc + c - '0';
}
```

So in the latter version, we put the initialization in the part of the loop. Is there a difference between doing these two things? And a related question to that: couldn't I also put initialization of `acc` in the loop as well? The answers to both questions have to do with the scope. When I declare the variable outside the loop, the scope is outside the loop. By putting the definition of `c` right in the loop, the scope of `c` is confined to the loop. Once the loop is done, there is no such `c` anymore. Or even, if we are inclined, we can write the loop as so:

```c
int acc = 0;
for (char c = getchar(); isdigit(c); acc=10*acc+c-'0', c=getchar());
```

Note the usage of comma operator here, which makes this legible. Also, the loop body is empty, which is indicated by `;`, an empty statement, or `{ }`.

What about the `peekchar` version?

```c
int acc = 0;
for (char c = peekchar(); isdigit(c); acc=10*acc+getchar()-'0', c=peekchar());
```

or even

```c
int acc = 0;
for (char c = peekchar(); isdigit(c);c=(getchar(),peekchar())) {
    acc = 10 * acc + getchar() - '0';
}
```

Often loop is controlled by counters, then we update counters. For example, here are some very common patterns:

```c
c = c + 1;
c = c - 2;
c = 10 * c;
c = c / 2;
c = c + d;
```

C has some specialized syntax, which is equivalent to above:

```c
c += 1;
c -= 2;
c *= 10;
c /= 2;
c += d;
```

If we want to increment/decrement by 1, then `++c` increments `c`; `--i` decrements `i`.

These are expressions, thus they have a value as well as an effect. `++c` increments `c` and produces the value of `c` and `--i` similarly. Which value? the old one or the new one? We can try these out.

```
int i = 1;
printf("%d\n", ++i); // 2
```

Thus the new value. There is also a postfix versions `i++`, `i--`, which people seem to like better. These postfix versions increment/decrement `i`, but produce the old value of `i`. So it implies the old value must be remembered.

For the most cases, prefix is simpler. The one possible reason for people prefer postfix is because the name of C++, which is not called ++C. If we use increment/decrement operators, and there is no good reason for postfix, we should use prefix version.

6.3 More on Global Data

Global variables likes `int i = 0;`, we should avoid where possible because it creates hidden dependencies. However, Global *constants* are still useful. We can force a variable to remain constant in C. We can say

```
const int passingGrade = 50; // cannot be mutated.
```

6.4 Intermediate Mutation (Racket)

What if we want to work with multiple address books?

```
(define work '(("Manager" 12345)
               ("Director" 23456)))
(define home '())

(define (add-entry abook name number)
    (set! abook (cons (list name number) abook)))

(add-entry home "Neighbour" 34567)

> home
'()
```

`home` is still empty, no change! Code doesn't work! Not clear how to make it work. What does substitution model say? `(add-entry home "Neighbour" 34567)` says substitute `'()` for `abook` in body. Then it becomes `(set! '() (cons (list name number) '()))`. The latter part makes sense. However, `(set! '() ...)` doesn't makes sense: we are mutating an empty list; also based on this statement, Racket has no idea we are mutating `home`.

To make this work... Recall from CS 145 (??), simulation of `structs` using `lambda`. Do the same thing to create a struct with one field, called a box. A box has two operations: get the value in the box; set the value to a new value.

```
(define (make-box v)
    (lambda (msg)
        (cond [(equal? msg 'get) v])))

```

```
(define (get b) (b 'get))

(define b1 (make-box 7))
(get b1)
; becomes
(define b1 (lambda msg) (cond [(equal? msg 'get) 7]))
(get b1)
; becomes
(get (lambda (msg (cond [(equal? msg 'get) 7]))))
; becomes
((lambda (msg) (cond [(equal? msg 'get) 7])) 'get)
; becomes
(cond [(equal? 'get 'get) 7])
; becomes
7
```

To support `set`, we can introduce a local copy of `v`.

```
(define (make-box v)
    (define val v)
    (lambda (msg)
        (cond [(equal? msg 'get) val])))

(define (get b) (b 'get))

(define b1 (make-box 7))
(get b1)
; becomes
(define val_1 7)
(define b1 (lambda (msg)
    (cond [(equal? msg 'get) val_1])))
; and eventually it will give us
7
```

Now how do we add `set`? It requires an extra parameter. We can achieve this by having the box return a function.

```
(define (make-box v)
    (define val v)
    (lambda (msg)
        (cond [(equal? msg 'get) val]
              [(equal? msg 'set) (lambda (newv) (set! val newv))])))

(define (get b) (b 'get))
(define (set b v) ((b 'set) v))

(define b1 (make-box 7))
(set b1 4)
; becomes
(define val_1 7)
(define b1 (lambda (msg) ... val_1 ...))
(set b1 4)
```

```
; becomes
(define val_1 7)
...
(set (lambda (msg) ... val_1 ...) 4)

; becomes
(define val_1 7)
...
(((lambda (msg) ... val_1 ...) 'set) 4)

; becomes
(define val_1 7)
((cond [(equal? 'set 'get) val_1]
       [(equal? 'set 'set) (lambda (newv) (set! val_1 newv))] 4 )

; becomes
(define val_1 7)
((lambda (newv) (set! val_1 newv)) 4)

; becomes
(define val_1 7)
(set! val_1 4)

; becomes
(define val_1 4)
(void)
```

7

Feb 3

7.1 Intermediate Mutation (Racket) cont'd

Why (now) does this fix the problem?

Before we had this:

```
(define home '())
(add home ... ...)

; reduces to
(add '() ... ...) ; value of the home substituted

; reduces to
(set! '() ... ...) ; can't update home, how do we know this empty list is home not
    sth else?
```

Now we have

```
(define home (make-box '())) ; this creates a variable, local define

(define (add abook name num)
    (set abook (cons (list name num) (get abook)))) ; 'get' fetch the var

(add home ... ...) ; this is a function that can update the created variable
```

Boxes are actually built into Racket. The syntax for boxes:

```
exp :: = ... ; anything before
        | (box exp)
        | (unbox exp)
        | (set-box! exp exp) ; first exp is a box, but doesn't have to be an id;
    second exp is a value.
```

Then address book example using built-in box could have been written:

```
(define home (box '(("Neighbour" 34567))))
(define work (box '(("Manager" 12345) ("Director" 23456))))
```

```
3
4  (define (add abook name num)
5      (set-box! abook (cons (list name num) (unbox abook))))
```

The semantics for box:

```
1  (box v)          ; v is a value
2  ; becomes
3  (define _u v)    ; u is a fresh name
4  _u               ; then (box v) becomes _u
```

Convention: When we write an underscore before a variable name it means the variable's value is not looked up during the expression evaluation, unless `(unbox __)` is called on it. (Note that this is for the stepping rule, no particular meaning in Racket).

If we have `(unbox _n)`, then we want to find `(define _n v)`, then `(unbox _n)` produces `v`. If we have `(set-box! _n v)`, then find `(define _n ...)` and replace that with `(define _n v)`, then `(set-box! _n v)` produces `(void)`.

Let's see an example on how we step on boxes.

```
1  (define box1 (box 4))
2  (unbox box1)
3  (set-box! box1 true)
4  (unbox box1)
5
6  ; becomes
7  (define _u1 4)
8  (define box1 _u1)
9  (unbox box1)
10 (set-box! box1 true)
11 (unbox box1)
12
13 ; becomes
14 (define _u1 4)
15 (define box1 _u1)
16 (unbox _u1)
17 (set-box! box1 true)
18 (unbox box1)
19
20 ; becomes
21 (define _u1 4)
22 (define box1 _u1)
23 4
24 (set-box! _u1 true)
25 (unbox box1)
26
27 ; becomes
28 ; (define _u1 4) no longer here
29 (define _u1 true)
30 (define box1 _u1)
31 4
32 (void)
```

```
(unbox box1)
; becomes
(define _u1 true)
(define box1 _u1)
4
(unbox _u1)

; becomes
(define _u1 true)
(define box1 _u1)
4
true
```

This is a bit messy, and it is one of the challenges of mutation.

7.2 The same problem in C

Suppose we want to write a function,

```
void inc(int x) {
    x = x + 1;
}
int main() {
    int x = 1;
    inc(x);
    printf("%d\n", x);
}
```

What we want is 2, but what we get is 1. Racket solution is putting the variable in a box. What is the C equivalent? one-field structure?

Structures in C:

```
struct Posn {
    int x;
    int y
}; // we got this cute/weird/curious semicolon here.
```

This semicolon is not optional, *needed*. The "reason" they designed this cuteness/shortcut is for the following:

```
struct Posn {
    int x;
    int y
} p1, p2, p3;
```

where we can define the `struct` and declare the variables at the same time. However, we tend to use `struct` variables locally, and `struct` definitions tend to be global...

Then we can use `struct` as follows:

```
int main() {
    struct Posn p;
    p.x = 3;
```

```
    p.y = 4;
    printf("p=(%d, %d)\n", p.x, p.y);
}
```

or we can initialize all at once:

```
int main() {
    struct Posn p = {3, 4};
    printf("p = (%d, %d)\n", p.x, p.y);
}
```

But watch out the following, which is not allowed:

```
int main() {
    struct Posn p;
    p = {3, 4};
    printf("p = (%d, %d)\n", p.x, p.y);
}
```

Let's try to write a function mutate the `struct`:

```
void swap (struct Posn p) {
    int temp = p.x;
    p.x = p.y;
    p.y = temp;
}

int main() {
    struct Posn p = {3, 4};
    swap(p);
    printf("p = (%d, %d)\n", p.x, p.y);
}
```

What we want is `p = (4, 3)`, but what we get is `p = (3, 4)`. Still doesn't work.

The problem: C (and also Racket) passes parameters by a mechanism called *call-by-value*. The idea is that the function operates on a *copy* of the argument, not the argument itself. Note that the Racket substitution model naturally implements call-by-value. For example, if we do,

```
(define x 3)
(f x) => (f 3) ; the value of x, not x itself
```

In C, what we have is

```
void inc(int x) {
    ++x;
}
```

Here `x` really does get mutated, but it's a copy of `x`, not the original from the caller. Therefore, the original remains the same. Similarly, in `swap` we wrote, the entire structure is copied into the function, thus the original structure does not change.

So there is something special about boxes. They are not equal to the value they hold, but they tell (know) you how to *find* the value: `unbox`/`get` to find the value. What does that look like in C? To find a value, we are asking where it is located: it is in memory (RAM). As we said, every value in memory

has an address. If given the address, we can "find" the value. Thus addresses could function as boxes. Instead of passing a value to a function, we can pass an address. For example,

```
int main() {
    int x = 1;
    inc(&x);  // & is known as ‘‘address-of’’ operator, which passes x’s address,
    not its value.
    printf("%d\n", x);
}

void inc(int x) {   // this is wrong now. We didn’t
    x = x + 1;      // get an int, we get an address.
}
```

Maybe: `void inc(address x) {...}` which is also wrong. We need more info than address: what type of data is stored at that address? We need to say `x` is the address of an `int`.

```
void inc(int *x) {  // x is called a pointer to an int
    x = x + 1;      // (i.e., the address of an int)
}
```

This is still wrong because of its body: we don't want to add `1` to the address, but we want to add `1` to the value stored at the address. So what we want is:

```
void inc(int *x) {
    *x = *x + 1;  // * = dereference operator
}
```

and the dereference operator `*` = fetch the value stored at this address (`unbox` in Racket). LHS of assignment: `*x = expr` = store the value of expr at address `x` (`set-box!` in Racket). Thus in Racket, it's equivalent to `(set-box! x (+ (unbox 1) 1))`.

When we see `int *x`, it's like `x` is a pointer to an `int`, but it's intended to be read is "`*x` is an `int`".

Alternatively we can write:

```
void inc(int *x) {
    *x += 1;
}
// OR
void inc(int *x) {
    ++*x;
}
// OR?
void inc(int *x) {
    *x++; // WRONG - why?
}
```

When we say `*x++`, we got operators on both left and right. Which of these two actually happens first? the `*` or the `++`? In C, postfix always takes precedence over prefix. So `*x++` means `*(x++)`. The address is incremented and the old value of the address is fetched (and thrown away). Thus no change to the original variable. If we cannot give up the postfix habit, we can do:

```
void inc(int *x) {
    (*x)++;
}
```

Now consider `swap`. The first version didn't work. Just like `inc`, we can fix this by passing a pointer:

```
void swap(struct Posn *p) {
    int temp = *p.x;
    *p.x = *p.y;
    *p.y = temp;
}
```

which is wrong, and won't even compile. Same problem as before: postfix before prefix. `*p.x = *p.y` means `*(p.x) = *(p.y)` and `p.x`, `p.y` aren't pointers. We need parentheses:

```
void swap(struct Posn *p) {
    int temp = (*p).x;
    (*p).x = (*p).y;
    (*p).y = temp;
}
```

However, this is clunky. `(*p).x` is common enough that it has its own notation: `p->x`. Thus the previous code can be written as:

```
void swap(struct Posn *p) {
    int temp = p->x;
    p->x = p->y;
    p->y = temp;
}
```

Thus we have more sophisticated user input: `scanf`. Note that `scanf("%d",x)` is wrong. It should read `x` as a decimal integer and skip leading whitespace. Here `scanf` is a function which can't modify `x`. Instead, we can do `scanf("%d", &x)`. We can also do `scanf("%d %d", &x, &y)`. The space between `%d`'s means to skip *any* amount of whitespace between the two `int`s (including zero). Zero space is possible if the second `int` is negative.

Note that `scanf` returns the number of arguments actually read. `scanf` has lots of options, very complicated.

8

Feb 4

8.1 Advanced Mutation

It means mutating structures and lists.

In Scheme, we can mutate parts of a `cons` with `set-car!` and `set-cdr!`.

In Racket, `cons` fields are immutable, cannot be mutated. For mutable paris, Racket provides `mcons`. To mutate fields, `mset-car!`, `mset-cdr!`. For `struct`s, it is also immutable. But Racket provides the option `#:mutable`. It will look like this:

```
(struct pos (x y) #:mutable)
(define p (posn 3 4))
(set-posn-x! p 5)
(posn-x p) ; => 5
```

This has an impact on our semantics: from CS 145, we said `(make-posn v1 v2)` is a value. However, now, `(posn v1 v2)` cannot be a simple value if it is mutable. It has to behave more like a `box`. How would a `struct` behave like a `box`? Here are the two ways. Is a `struct` automatically `boxed`? or is a `struct` a `box`? We can find that out by some experiments.

```
(struct posn (x y) #:mutable #:transparent)
(define (mutate-posn p)
    (set-posn-x! p (+ 1 (posn-x p))))

(define (mutate-posn2 p)
    (set! p (posn (+ 1 (posn-x p)) (posn-y p))))

(define p (posn 1 2))
(define q (posn 1 2))

(mutate-posn p)
(mutate-posn2 q)
p ; (posn 2 2)
q ; (posn 1 2)
```

So a `struct` is not automatically `boxed`, but it does `box` its contents. So we can write `(posn v1 v2)` as

```
(define _val1 v1)   ; recall, no expansion
```

```
(define _val2 v2)
(posn _val1 _val2)
```

`(posn-x p)` where `p` is `(posn _val1 val2)`, we find the definition for `_val1`, fetch the value.

`(set-posn-x! p v)` where `p` is `(posn _val1 val2)`, we find `(define _val1 ...)`, replace it with `(define _val1 v)`.

So now we are ready to do some stepping. For example,

```
(define p1 (posn 3 4))
(set-posn-x! p1 5)

; becomes
(define _v1 3)
(define _v2 4)
(define p1 (posn _v1 _v2))
(set-posn-x! p1 5)

; becomes
(define _v1 3)
(define _v2 4)
(define p1 (posn _v1 _v2))
(set-posn-x! (posn _v1 _v2) 5)

; becomes
(define _v1 5)
(define _v2 4)
(define p1 (posn _v1 _v2))
(void)
```

These rules generalize to any mutable `struct`, `mcons`.

Now consider

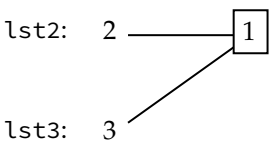
```
(define lst1 (cons (box 1) empty)) ; note that we are using cons, not mutable
(define lst2 (cons 2 lst1))
(define lst3 (cons 3 lst1))
(set-box! (first (rest lst2)) 4)
(unbox (first (rest lst3))) ; gives 4
```

By the CS 145 understanding, this should produce 1. But in fact, it gives 4. Why?

`lst2` = `(cons 2 (cons (box1) empty))`

`lst3` = `(cons 3 (cons (box1) empty))`

In this case, the two `(box 1)`'s are actually the same object. When we define `lst2` and `lst3`, these two lists actually share the same tail. What we actually have is



We could never tell that this was true in CS 145 because there is no way to observe a difference in

terms of whether these two lists are completely distinct lists or actually they are sharing the tail unless we can perform mutation. Thus we do need mutation to observe this.

Nevertheless, we could actually deduce they are actually sharing the tail. We can do

```
(define lst1 '(1 2 ... 100000000)) ; which takes O(n) time to build
(define lst2 (cons 0 (rest lst1))) ; O(1) time
```

If the second line is fast, it's not possible for Racket to recreate an entire list, thus it must be reusing that list. From that perspective, `lst2` is sharing the tail with `lst1`.

Under the old substitution rules, we will get the wrong answer `1`. Under the new substitution rules, boxes are rewritten as a separate define with deferred lookup. We then end up with

```
(define _val 1)
lst2 = (cons 2 (cons _val1 empty))
lst3 = (cons 3 (cons _val1 empty))
```

Here the shared item reflected in the rewrite.

All these force us to rethink what we mean by `define`. If we have

```
(define x 3)
(set! x 7)
x
; becomes
(define x 3)
(void)
x
; becomes
7
```

We cannot replace all occurrence of `x` with `3`, otherwise we could have gotten `3` at the end. So `x` is not just a value, but something we can mutate, it's an entity we can access. Therefore `x` must denote a *location*, and the location contains the value. So we don't just have one lookup δ : `var` $\to$ `value`, instead we have *two* lookups: `var` $\to$ `location`, `location` $\to$ `value` where the second lookup is carried out by RAM.

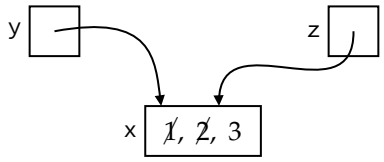
`set!` changes the `location` $\to$ `value` map, but not the `var` $\to$ `location` map (nothing changes that, at least for now). Similarly, `set-box!` changes the `location` $\to$ `value` mapping. `(define ...)` creates a *location*, fills it with a `value`. Keep this in mind.

8.2 Aliasing in C

Does this happen in C as well? Yes. Consider

```
int main() {
    int x = 1;
    int *y = &x;
    int *z = &x; // or int *z = y;
    *y = 2;
    *z = 3;
    printf("%d %d %d\n", x, *y, *z);
}
```

The output is `3 3 3`. Why? `y` is initialized to `x`'s address, the `y` points to the location where `x` resides, and `z` is initialized to `y` (or `&x`), thus `z` also equals to `x`'s address. Therefore, `*y = 2` stores `2` at `x`'s location: `x == *y == *z == 2`. Similarly, `*z = 3` stores `3` at `x`'s location. We can picture this visually:



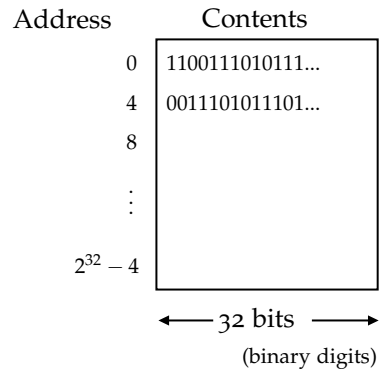
Therefore, `x`, `*y`, `*z` are three different names for the same data. This phenomenon is called *aliasing*: accessing the same data by different names. Aliasing is tricking business, and it can be subtle. Consider the following :

```c
void f(int *x, int *y) {
    *y = *x + 1;
    if (*y == *x) {
        printf("How could this every print?\n"); // (*)
    }
}

int main() {
    int z = 1;
    f(&z, &z);          // makes x and y aliases, thus (*) DOES print
    printf("%d\n", z);  // print 2
}
```

Hence it makes programs very difficult to understand.

8.3 Memory and vectors

Recall from the beginning of the course: Memory is a set of numbered "slots":



Each box is 8 bits (one byte), but they are usually treated in groups of 4-byte *words*.

Primitive data structure: the *array*, it's a "slice" of memory, and a sequence of consecutive memory locations. We will discuss at length when we return to C.

In Racket (also Scheme), it's known as the vector. It is used much like a traditional array. Unlike arrays, the slot of the vector can hold items of any size. Thus we can have unlimited integers, strings, whatever.

8.4 Vectors in Racket

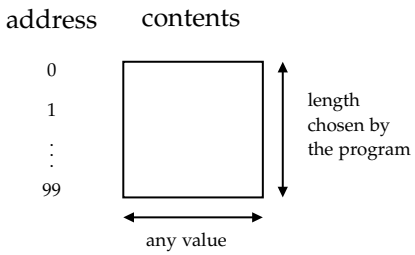
```
(define x (vector 'blue true "you"))
```

is a 3-item vector. From here we know that we can put whatever we want in `vector`. We can also do

```
(define y (make-vector 100))
y = '#(0 0 0 ... 0 0 0)
```

which creates a `vector` of length 100.

Unlike a `list`, it's like a many slice of memory. So it looks like,



We can also create a vector like

```
(define z (build-vector 100 sqr))
=> 0 1 4 9 ... 99^2
(define y (make-vector 100 5))
=> 5 5 5 5 ... 5 ; a hundred 5's
```

What is different about `vectors` versus `lists` is the way we work with them. We already know we work with a `list` by taking the first item and taking the rest. We process the list recursively. Vectors are quite different. They are not accessed by `first` and `rest`. Vector says which item do you want?

```
(define y (make-vector 100 5))
(vector-ref y 7)    ; gives 5
(vector-set! y 7 42)
(vector-ref y 42)   ; gives 42
```

Thus we access/mutate items by index. Hence the main advantage of `vectors` over `lists` is `vector-ref` and `vector-set!` run in $O(1)$ time. This is a consequence of the way `vector` is stored versus the way `list` is stored. With a `list`, we can't get to a second item until we have been through the first item. If we want the 100th item in the list, we have to `rest` 99 times and `first`. On the other hand, with `vectors`, which is a slice of memory, if we want 100th item of the `vector`, all we need to say is where it's start and add 100 $\times$ the slot size, which gives us the exact address where the item is located. Thus we can fetch any item in the `vector` in constant time.

It turns out these things are cranky to work with. What's wrong with `vectors`? It has several disadvantages:

1. *The size is fixed.* A `list` can very easily grow: all we need to do is to put `cons` at the front. However, with a `vector`, we have made a choice to designate a particular slice of memory of being part of that `vector`. The memory before/after it could very well be used for other things. Thus we cannot just take a `vector` and make it bigger.

2. *It's difficult to add or remove elements.* Everything is stored consecutively. If we want to take something out of the middle, that leaves a gap, we then need to shuffle everything down to close the gap. Similar situation if we want to add something into the middle.

3. `vector-set!` *tends to force an imperative style.* Once we start to work with `vectors` instead of `lists`, we will find our Racket code doesn't work so well functionally anymore, it kinda force us to the imperative Racket.

9

Feb 9

9.1 Vectors in Racket cont'd

Let's first write `build-vector`:

```
(define (my-build-vector n f)
    (define res (make-vector n))
    (define mbv-h i) ; my build vector helper
        (cond [(= i n) res]
              [else (vector-set! res i (f i)) (mbv-h (+ i 1))])
    (mbv-h 0))
```

Vectors work well with imperative-style algorithms. Racket provides macros `for`, `for/vector` that facilitate this. Thus we could write

```
(define (my-build-vector n f)
    (define res (make-vector n))
    (for ([i n]) ; i goes from 0 to n
        (vector-set! res i (f i)))
    res)
```

Or in `for/vector` form,

```
(define (my-build-vector n f)
    (for/vector ([i n]) (f i)))
```

Let's do another example with `vectors`. For example, sum the elements of a `vector`,

```
(define (sum-vector vec)
    (define (sv-h i acc)
        (cond [(= i (vector-length vec)) acc]
              [else (sv-h (+ i 1) (+ acc (vector-ref vec i)))]))
    (sv-h 0 0))
```

It's very look-like, then we can use `for`:

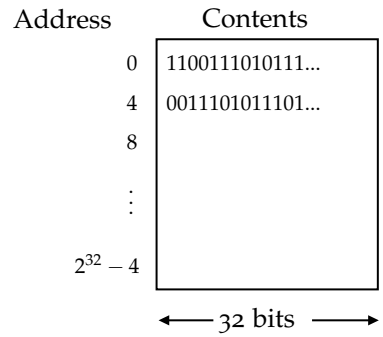
```
(define (sum-vector vec)
    (define sum 0)
    (for [(i (vector-length vec))]
```

```
        (set! sum (+ sum (vector-ref vec i))))
    sum)
```

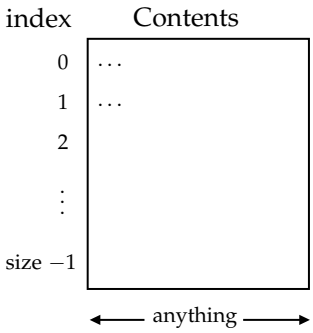
It's not pure functional because it uses mutation. But in some sense, it *looks* pure functional. And the use of mutation is confined to the internals of `sum-vector`. It can't be detected outside the function. Thus outsiders could consider it pure functional.

This provides a strategy for keeping the problems with mutation under control: hide it behind a pure functional interface.

Recall our model for computer's memory: huge lookup table,



Racket `vectors` model that



How does this work? Remember that memory slots only hold fixed size data, and yet Racket has unlimited numbers? strings?

Let's go back to `struct`. Recall

```
(define (mutate-posn p)
    (set-posn-x! p (+ 1 (posn-x p))))

(define p (posn 3 5))
(mutate-posn p)
(posn-x p) ; gives 4
```

Does that happen in C? The equivalent in C would be

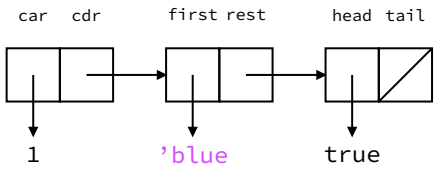
```
void mutate (struct Posn p) { p.x += 1; }
int main() {
    struct Posn p = {3, 5};
    mutate(p);
    printf("%d\n", p.x); // gives 3
}
```

So from here, we learnt that Racket `structs` aren't like C `structs`: the `struct` is coped in Racket, but changes to the field still persists. As we conclude last time, the fields of a Racket `struct` are boxed, i.e., they are pointers.

Similarly, the items in Racket `vector` are *addresses* that point to the actual contents (which can then be of any size). Similarly, the fields of a `cons` are pointers. When we say

```
(cons 1 (cons 'blue (cons true empty)))
```

we can represent this as a box-and-pointer diagram:



Also, since Racket is dynamically typed, the values `1`, `'blue`, `true` must include type information. More later.

9.2 "Vectors" in C: Arrays

An `array` is a sequence of consecutive memory locations. For example,

```
int main() {
    int grades [10]; // Array of 10 ints
    for (int i = 0; i < 10; ++i) {
        scanf("%d", &grades[i]);
    }
    int acc = 0;
    for (int i = 0; i < 10; ++i) {
        acc += grades[i];
    }
    printf("%d\n", acc/10);
}
```

When we see `a[i]`, this accesses the i-th element of array `a`.

`int grades[10];` tells us valid entries are `grades[0], ..., grades[9]`.

What happens if we go out of bounds? It's undefined behaviour.

Will it stop us? No. The program may or may not crash; no way to detect that if it didn't crash.

We can also give the bound implicitly:

```
int main() {
    int grades [] = {0, 0, 0, 0, 0};
    printf("%zd\n", sizeof(grades)/sizeof(int)); // gives 5
}
```

`sizeof` operator tells us the amount of memory `grades`/`int` occupy, 20 and 4 bytes respectively.

`int` in our implementation of C occupies 32 bits but we are running on 64 bits machine. So the amount of memory available might well be larger than what 32 bits can hold. So what `sizeof` creates is a value of type, not `int`, but `size_t`, which is a type that the compiler supplies. First, it is unsigned, i.e., no

negative `size_t`. Second, it's large enough to hold any amount of memory. `z` in `"%zd"` here ensures that when printing out, it is not interpreted as `int`, indicates `size_t`.

Now let's talk about functions on `arrays`.

```
int sum(int array[], int size) { // size is not part of the type of array, works on
     arrays of any size
    int res = 0;
    for (int i = 0; i < size; ++i) res += arr[i];
    return res;
}
```

If we pass `arrays` by value, we copy the whole array, which is expensive. So C will not pass `array` by value. Consider

```
int main() {
    int myArray [100];
    // ...
    int total = sum(myArray, 100); // looks like a copy, how is this not copy?
    // ...
}
```

This leads to the most confusing rule in all of C: The name of an `array` is shorthand for a pointer to its first element. So in fact, `myArray` is shorthand for `&myArray[0]`. Therefore, `sum(myArray, 100)` passes a pointer, not a whole `array`, into the function. But `sum` was expecting an `array`, not a pointer.

Why not `int sum(int *arr, int size)` then? The answer is we could have. `int *arr`, `int arr[]` are identical in meaning, in parameter declarations. Now let's use pointers to write `sum` this time:

```
int sum(int *arr, int size) {
    int res = 0;
    for (int i = 0; i < size; ++i) res += arr[i]; // is that OK to do this to a
    pointer?
    return res;
}
```

Yes, it actually does work.

9.3 Pointer Arithmetic

Let `t` ne a type. If we declare an array `t arr[10];`, then we know `sizeof(arr) = 10 * sizeof(t)`, and `arr` is shorthand for `&arr[0]`. Therefore, `*arr` is equivalent to `arr[0]`. Then what expressions produces a pointer to `arr[1]`?

> `arr + i` is shorthand for `&arr[i]` for `i = 1, 2, ...`

Numerically, `arr + n` produces the address equal to `arr + n * sizeof(t)`. If `arr + k` is shorthand for `&arr[k]`, then `*(arr+k)` means `arr[k]`. Therefore, `sum` is equivalent to

```
int sum(int *arr, int size) {
    int res = 0;
    for (int i = 0; i < size; ++i) res += *(arr + i);
    return res;
}
```

In fact `a[k]` is just shorthand for `*(a+k)`. Actually, to confuse other people, we can do

$$\texttt{a}[\texttt{k}] \equiv *(\texttt{a} + \texttt{k}) \equiv *(\texttt{k} + \texttt{a}) \equiv \texttt{k}[\texttt{a}]$$

We can push the pointer version of `sum` slightly further:

```
int sum(int *arr, int size) {
    int res = 0;
    for (int *cur = arr; cur < arr + size; ++cur) res += *cur;
    return res;
}
```

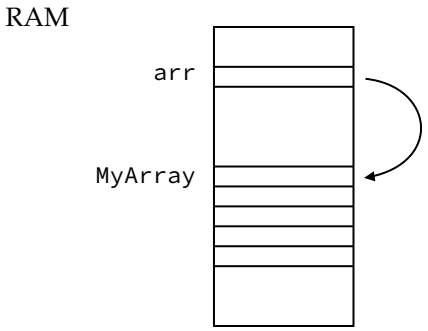
Note that the usage of `arr+size`, which is a pointer outside the array: it is valid to construct a pointer that goes one slot past the end of the array, but it is not valid to dereference that pointer (UB).

`cur < arr + size` is a pointer comparison: return true if `cur` points to an earlier element in the same array than `arr + size`. Comparing pointers in different arrays or not in arrays at all, it UB.

Any pointer can be thought of as pointing to the beginning of an array. We have the same syntax for accessing items through an array as through a pointer. So are arrays and pointers the same thing? No! Let's see an evidence:

```
void f(int arr[]) {
    printf("%zd\n", sizeof(arr));
}
int main() {
    int myArray[10];
    printf("%d\n", sizeof(myArray));
    f(myArray);
}
```

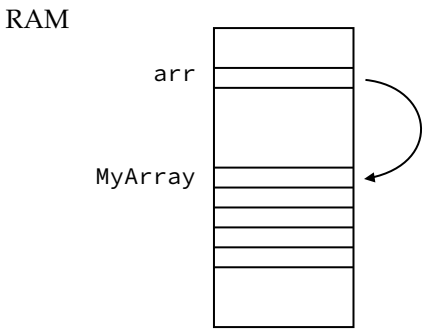
Outputs: `40` (size of array), `8` (size of a pointer to the array's first item). If we take a look at RAM, they are quite different:



10

Feb 11

Continue from last time:



What does the compiler do? When we say `myArray[i]`, it will fetch `myArray` location from environment, and add `i*sizeof(int)`, and then fetch this address from the store (RAM).

What happens when you call `arr[i]`? It will fetch `arr` location from environment, then fetch `myArray` address from store, then add `i*sizeof(int)` to address, then fetch the value from this address in store.

These two operations, `myArray[i]` and `arr[i]` may look the same, but do slightly different things.

We saw that a Racket `struct`, e.g., `(struct posn (x y))`, is like a C `struct` whose fields are pointers. How can we achieve this in C?

> Aside:
>
> ```
> int *x;
> int* x;
> int * x;
> ```
>
> They all mean the same thing. The first one is more idiomatic C, the second one is often favored in C++, and the third one, the instructor is not sure there's too many people favor the third one, but it exists.
>
> But there's one technical thing, if we do
>
> ```
> int *x, y;
> ```
>
> to declare two pointers. Here `x` is a pointer, `y` is not a pointer, just an `int`. If we want `y` also to be a

> pointer, we have to do
>
> ```
> int *x, *y;
> ```

```
struct Posn {
    int *x;
    int *y;
};

int main() {
    struct Posn p;
    // what are p.x, p.y pointing at?
    *p.x = 3;
    *p.y = 4;
}
```

Very likely, this is gonna crash. `p.x` and `p.y` are uninitialized pointers, thus they point at arbitrary locations, dictated by whatever value they happen to hold.

Racket must do something more than this: `(posn 3 4)` must also reserve memory for `x` and `y` to point at, to hold the `3` and `4`. Therefore, we need to do the same in C:

```
#include <stdlib.h>
struct Posn makePosn(int x, int y) {
    struct Posn p;
    p.x = malloc(sizeof(int));
    p.y = malloc(sizeof(int));
    *p.x = x;
    *p.y = y;
    return p;
}
```

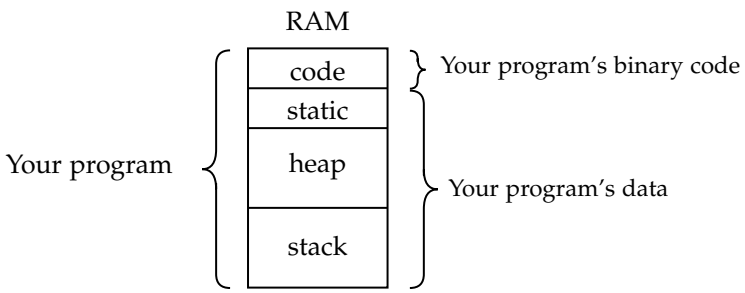
`malloc(n)` says request `n` bytes of memory. Then in `main`, we can do

```
int main() {
    struct Posn p = makePosn(3, 4); // OK
    ...
}
```

We need to understand exactly what's happening.

10.1 Memory Management

Memory layout (applies to C and Racket)



Static area is where global/static variables are stored. The lifetime of these variables is the entire program.

What is a stack? It is abstract data type (ADT) with LIFO (last in first out) semantics. In LIFO, we can only remove the most recently-inserted item. For stacks, operations are

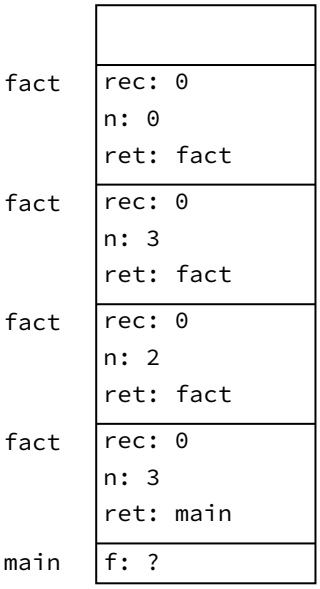
- Push - add an item to the stack;
- Top - what is the most recently inserted item?
- Pop - remove the most recently inserted item.
- Empty? - is the stack empty?

Racket lists are stacks: Push = `cons`, Pop = `rest`, Top = `first`.

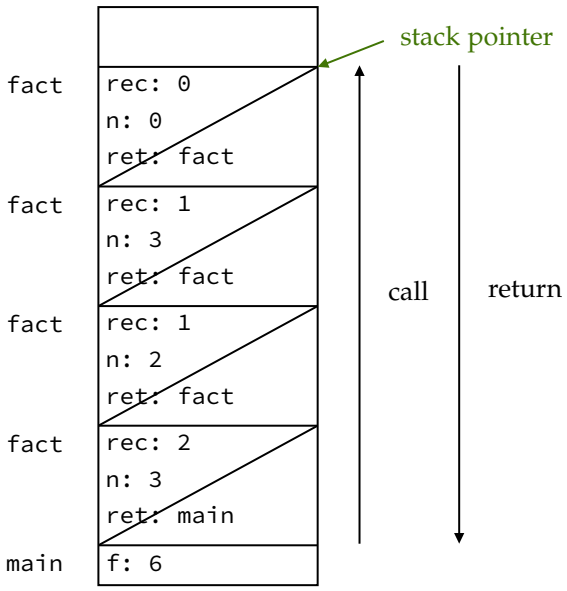
Program stack stores local variables. Let's see an example.

```
int fact(int n) {
    int rec = 0;
    if (n == 0) return 1;
    rec = fact(n-1);
    return n * rec;
}

int main() {
    int f = fact(3);
}
```

`ret` here denotes what do I return to.



Then after all recursive calls are done, we have



Each function call gets a *stack frame*:

- local variables are pushed onto the stack,
- also the return address: where to go when the function returns.
- each invocation of the function gets its own version of local variables.

When a function returns, its stack frame is popped. This means all local variables in that frame are released. They are not typically erased: the program keeps track of where the top of the stack is. This is what's known as *stack pointer*. The "top-of-stack" pointer moved to top of the next frame, then the old frame will be overwritten next time a frame is pushed onto the stack.

The stack holds local variables. The lifetime of variables on the stack will be scope-based.

So what if you have data that must persist after a function returns. We might try the following. What's wrong with this?

```
struct Posn makePosn(int x, int y) {
    struct Posn p;
    int a = x;
    int b = y;
    p.x = &a;
    p.y = &b;
    return p;
}
```

Here we have initialized `p.x` and `p.y`, but we have initialized them to these local variables `a` and `b`'s address. Wait no... This is bad. If in `main`, we do

```
int main() {
    struct Posn p = makePosn(3, 4);
}
```

When the function returns, local variables `a` and `b` are released; pointers `p.x` and `p.y` are thus pointing to dead memory.

Never return a pointer to data stored on local stack. If a function is to return a pointer, the pointer should point to either static, heap, or non-local stack data.

11

Feb 23

11.1 Memory Management cont'd

Some of the material is repeated here because the twitch stream was broken last time...

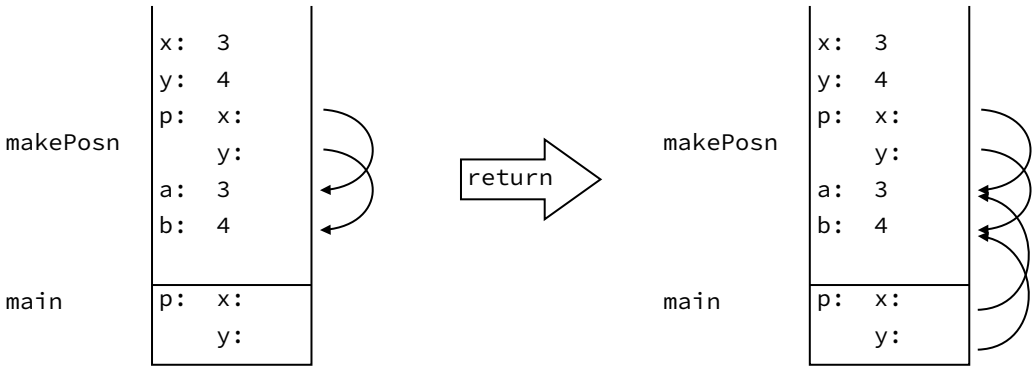
So what if you have data that must persist after a function returns. We might try the following. What's wrong with this?

```
struct Posn makePosn(int x, int y) {
    struct Posn p;
    int a = x;
    int b = y;
    p.x = &a;
    p.y = &b;
    return p;
}
```

This is very BAD!! If in `main`, we do

```
int main() {
    struct Posn p = makePosn(3, 4);
}
```

Let's imagine what's happen in stack:



When the function returns, local variables `a` and `b` are released; pointers `p.x` and `p.y` are thus pointing to dead memory. Returned `p` contains pointers to local stack-allocated data. Don't do this! `x + y` (or

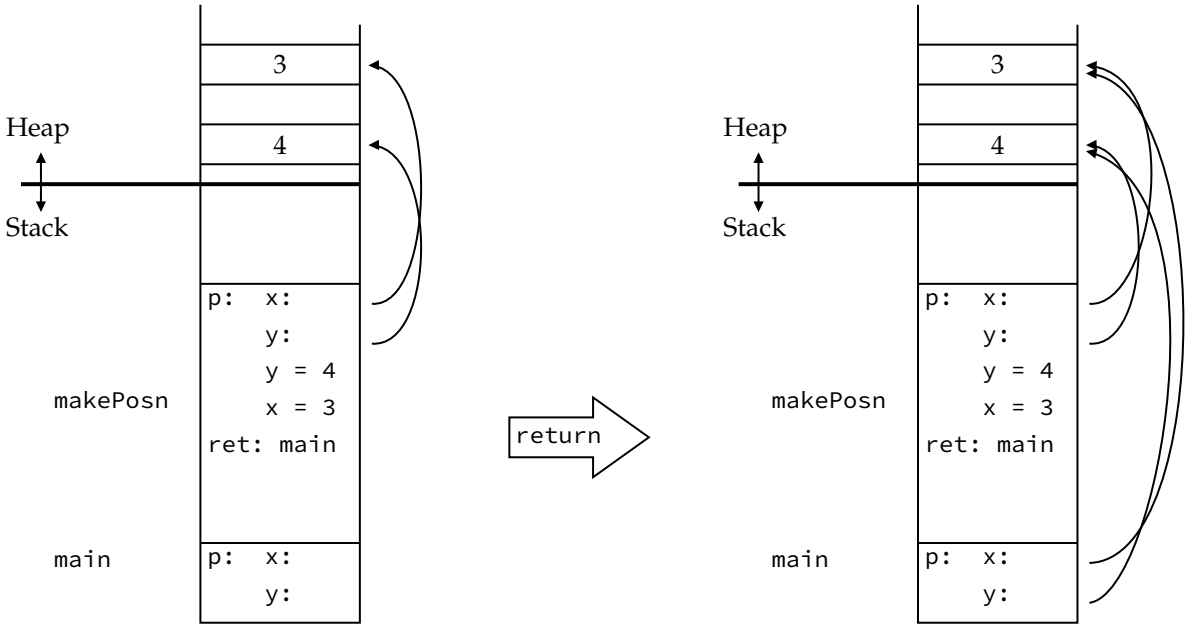
`a + b`) won't survive past the end of `makePosn`.

This is then the difference between pointers to stack memory and what `malloc` does. `malloc` requests memory from the *heap*. Heap is a pool of memory from which we can explicitly request "chunks".

The lifetime of stack memory is until the variable's scope ends. (e.g., end of the function). The life of heap memory is arbitrary. Let's recall

```
struct Posn makePosn(x, y) {
    struct Posn p; // one the stack
    p.x = malloc(sizeof(int)); // points to the heap
    p.y = malloc(sizeof(int)); // points to the heap
    *p.x = x;
    *p.y = y;
    return p;
}

int main() {
    struct Posn p = makePosn(3, 4);
    ...
}
```

Now let's consider the memory layout.



When `makePosn` returns, `p` (including `p.x`, `p.y`) is popped off the stack, which is then no longer live. However `3` and `4` are on the heap, which are still live. So `p` from `makePosn` is copied back to `main`'s frame. `main` then has access to `3` and `4` on the heap, and these outlive `makePosn`. Note that `make-posn` (or `posn`) in Racket would do the same thing.

What is the lifetime of heap-allocated data? As we discussed, it is arbitrarily long. If heap-allocated data *never* gets away, the program will eventually run out of memory, even if most of the data in memory is no longer in use. Racket solution to this problem: there is a run-time process detects memory that is no longer accessible. For example,

```
(define (f x)
    (define p (posn 3 4)) ; certainly is not needed after f returns
```

```
    ...
    (+ x 1))
```

and automatically reclaims it. This is a process known as *garbage collection*.

On the other hand, C solution: Heap memory is `freed` when we `free` it. For example,

```
int *p = malloc(...);
...
free(p); // release p's memory back to the heap.
```

What happens if we don't call `free`? Failing to `free` all allocated memory is called a *memory leak*. Programs that leak memory will eventually fail, if they run long enough.

Consider the following program:

```
int *p = malloc(sizeof(int));
free(p);
*p = 7;
```

Will this program crash? Almost certainly, w.h.p. not. `free(p)` doesn't change `p`. `p` still points to that memory, therefore, storing something at that memory probably still works, but `p` is not pointing at a valid location. And that location may be assigned to another pointer by another `malloc` call. This is called a *dangling pointer*, which is bad.

So a better solution would be: after `free(p)`, we assign `p` to point to a guaranteed-invalid location:

```
int *p = malloc(sizeof(int));
free(p);
p = NULL;
```

This is called *null pointer*, which points to nothing. `NULL` us bit really part of the C language. There are certain headers that define `NULL` as constant equal to `0`. We could equally well say `p = 0;` Now what happens if we deference `NULL`? It's an undefined behavior. The program *may* crash.

If `malloc` fails to allocate memory, it returns `NULL`. Moreover, `free`ing a `NULL` pointer is guaranteed to be safe, which does nothing.

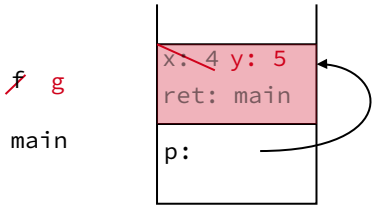
Let's consider a situation we discussed before, but in a slightly different way:

```
int *f() {
    int x = 4;
    return &x;
}

int g() {
    int y = 5;
    return y;
}

int main() {
    int *p = f();
    g();
    printf("%d\n", *p);
}
```

Let's compile it with `clang`. Then we do get a warning:

```
warning: address of stack memory associated with local variable returned [-Wreturn-
    stack-address]
    return &g;
           ^
```

If we run the program, we get `5`, not `4`. Let's take a look at memory:



This is precisely why the program is so dangerous. This is not guaranteed, but this could happen. Let's compile with `gcc`. We then get another warning of the same kind of thing:

```
warning: function returns address of local variable [-Wreturn-local-addr]
    return &g;
           ^~
```

This time when we run it, we see the program crashes with "Segmentation fault (core dumped)". `gcc` is exercising an amazing amount of caution here , which saves us from ourselves. `gcc` is taking an approach which says: anytime I think you are returning a variable that is a pointer to a local, I am going to instead return `NULL`.

`p` points to a dead memory. By the time `f` returns, `x` is no longer a valid location. This is another instance of *dangling pointer*. The program (probably? depending on the compiler) will not crash, but will behave badly. When `g` is called, it occupies `f`'s old stack frame. `y` now occupies `x`'s old spot. `*p` now is `5` (still a dangling pointer).

The lesson we learnt from this: NEVER return a pointer to a local variable.

If we want to return a pointer, it should point to static, heap, or non-local stack data. For example,

```
int *pickOne(int *x, int *y) {
    return ... ? x : y;
}
```

This is fine because `x` and `y` are not pointing to my local stack. It's fine to point to other's stack. Let's also show an example of pointing to heap:

```
struct Posn *getMeAPtr() {
    struct Posn *p = malloc(sizeof(struct Posn));
    return p;
}
```

Finally, let's consider a pointer to static:

```
int z = 5;
int *f() { return &z; }
```

This is also fine because the lifetime of global variable is the entire program, thus it is not pointing to a dead memory.

In general, when should we use heap? Three situations come into mind:

1. For data that should outlive the function that created it.
2. For data whose size is not known at compile-time.
3. For large local arrays.

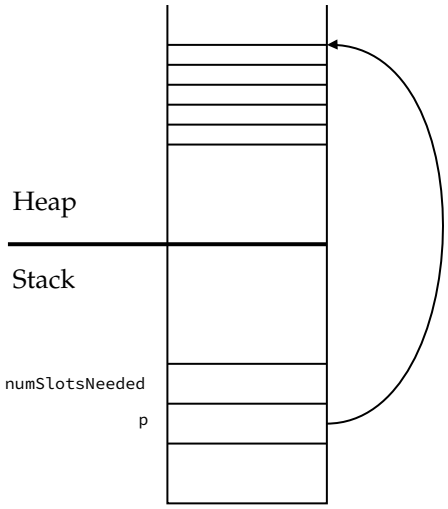
Let's see some examples. The first situation is as above. For the second situation,

```
int *p = malloc(sizeof(int));
```

Is that useful? Why not just `int n`? But what if we ask for more memory? Consider

```
int numSlotsNeeded;
scanf("%d", &numSlotsNeeded);
int *p = malloc(numSlotsNeeded * sizeof(int));
...
free(p);
```

Now we can access `p[0], p[1], ..., p[numSlotsNeeded-1]`, which is dynamic array (heap-allocated). Let's take a look at the memory layout.



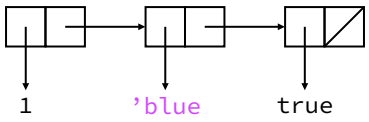
Finally, let's consider the third case. Programs typically have more heap memory available than stack memory. Consider

```
int recursiveFunction(int n) {
    ...
    int HeapArray[10000]; // this eats up stack space for each recursive call
    ...
    recursiveFunctionFunction(n-1);
}
```

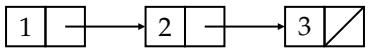
11.2 Linked list

C arrays mimic Racket vectors. Can we get the behavior of a Racket list? We know that `(cons x y)` produces a pair 

Recall that Racket is dynamically typed, then we could have



List items can have different types. On the other hand, C is statically typed, which means list items would need to have the same type. If not the same type, then it leads to headaches. So there's no real need for pointers to the data fields. We could write like this:



This suggests us a `struct` definition like

```
struct Node {
    int data;
    struct Node *next;
};
```

Also, we can write `cons`:

```
struct Node *cons(int data, struct Node *next) {
    struct Node *result = malloc(sizeof(struct Node));
    result->data = data;
    result->next = next;
    return result;
}
```

Then in `main`:

```
int main() {
    struct Node *lst = cons(1, cons(2, cons(3, 0)));
    ...
}
```

This is called a *linked list*.

12

Feb 24

12.1 More linked list

Now let's process a linked list.

```
int length(struct Node *lst) {
    if (!lst) return 0;
    return 1 + length(lst->next);
}
```

Or using iteration/loops, we can write like this

```
int length(struct Node *lst) {
    int res = 0;
    for (struct Node *cur = lst; cur; cur=cur->next) {
        ++res;
    }
    return res;
}
```

Can we write `map`? The idea would be

```
int f(int n) {...}
int main() {
    struct Node lst = ...;
    struct Node lst2 = map(f, lst);
}
```

Well, are we allowed to pass a function as an argument to another function in C? Technically no, because functions in C are not first class values, thus we cannot take them and store them in data structures and so on. However, we can pass a pointer to a function in C. In `map(f,lst)`, the name of a function is shorthand for a pointer to its code. This is how we use `map`, how do we write it? What type do we use for `f`? We know it's a pointer to a function.

```
struct Node *map(int *f(int), struct Node *lst);
// this is wrong, because postfix before prefix. This is a function returning a
    pointer to an int.

```

```
// correct way
struct Node *map(int (*f)(int), struct Node *lst) {
    if (!lst) return 0;
    return cons(f(lst->data), map(f, lst->next));
}
```

Now it's time to `free` the list. Say we have created a list:

```
int main() {
    struct Node *lst = cons(1, cons(2, cons(3, 0)));
    ...
    free(lst);
}
```

This will create memory leak because 2 and 3 get leaked. Our second attempt:

```
int main() {
    ...
    for (struct Node *cur=lst; cur; cur=cur->next) free(cur);
}
```

This is still wrong, because `cur=cur->next` happens after `free(cur)`, then `cur` is dangling. To fix this, we need to grab the next pointer before we `free`. Now let's write a loop doing properly:

```
for (struct Node *cur=lst; cur;) {
    struct Node *tmp = cur;
    cur = cur->next;
    free(tmp);
}
```

This can be also done recursively:

```
void freeList(struct Node *lst) {
    if (lst) {
        freeList(lst->next);
        free(lst);
    }
}
```

12.2 Application of Vectors

12.2.1 ADT Map/Dictionary (Mutable version)

Here are several operations it has. When we are specifying these operations, we should give *preconditions* (what do I need to satisfy in order to be valid when I call this function) and *postconditions* (if we call this function, having met the preconditions, what then does the function guarantee to us) for these operations.

`make-map`: it has no parameters. Pre: true[1]. It produces a new `map`.

`map`: params: map M, key k, value v. Pre: true. It produces no value. Post: if $\exists v'$ such that $(k, v') \in M$, then $M \leftarrow M \setminus \{(k, v')\} \cup \{(k, v)\}$. else $M \leftarrow M \cup \{(k, v)\}$.

[1] this means we can always call this function because true is `true`

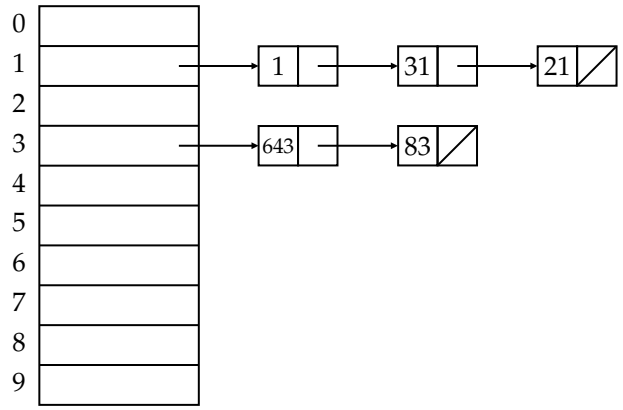
`remove`: 2 params: map M, key k. Pre true. It produces no value. Post: if $\exists v$ such that $(k, v) \in M$, $M \leftarrow M \setminus \{(k, v)\}$. Otherwise M is unchanged.

`search`: params: map M, key k. Pre: true. Value produced is v such that $(k, v) \in M$, otherwise something outside the value domain.

To implement these, we start by assuming keys are integers (for simplicity, we omit values).

If we use an association list, then accessing an item takes time proportional to its position in the list ($O(\texttt{len}(L))$ worst case). If we use a BST, then we do have the same worst case running time because there is no guarantee the BST will be shaped well. If we use a balanced BST (e.g., AVL tree), then $O(\log n)$ is the worst case time, where $n = |M|$, we pay for a difficult implementation.

If we use vectors instead, we have the advantage of $O(1)$ for any index-based access, but how big should the vector be? To have size of vector equal to maximum key? This will waste a lot of space. We can instead combine these two: vector of association lists, which is called a *hash table*.




```
(define (create-hashtable size) (make-vector size empty))
```

To which association list should we add (k, v)? We need to map k to a vector index. Mapping called a *hash function*. For simplicity, we use remainder of k by the length of the vector. For this idea to work well, the hash function must distribute keys evenly over the indices.

```
(define (ht-search table key)
    (define index (modulo key (vector-length table)))
    (define hashlist (vector-ref table index))
    (define lookup (assoc key hashlist))
    (if lookup (second lookup) false))
```

13

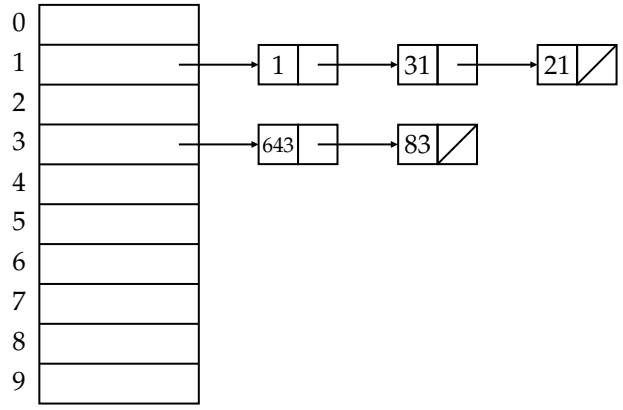
Feb 25

13.1 Hash tables cont'd

Now let's do add.

```
(define (ht-add table key val)
    (define index (modulo key (vector-length table)))
    (define hashlist (vector-ref table index))
    (define lookup (assoc key hashlist))
    (if lookup
        (when (not (equal? (second lookup) val))
              (vector-set! table index
                  (cons (list key val) (remove lookup hashlist))))
        (vector-set! table index (cons (list key val) hashlist))))
```

Quick note: if the keys are not numbers, then we need a hash function that maps them to numbers. Then we end up with a structure like this:



The running time to fetch values from one of these slots is going to be $O(n/\ell)$ where n is the number of items in the collection and ℓ is the length of the vector. The hope is that ℓ is somehow chosen based on the number of items that we expect to have in our table. For example, if $\ell = kn$, then the running time is $O(n/\ell) = O(n/kn) = O(1/k) = O(1)$.

13.2 ADT's in C: Sequence

First thing to note is that C doesn't have modules. C has files. We are going to implement an ADT in C, sequence. The operations:

- empty sequence
- `insert(s, i, e)`: insert e at index i in s. Pre: $0 \leq i \leq \texttt{size}(s)$.
- `size(s)`: number of elements in s. Pre: true.
- `remove(s, i)`: remove item from index i in s. Pre: $0 \leq i \leq \texttt{size}(s) - 1$.
- `index(s, i)`: return ith element of s. Pre: $0 \leq i \leq \texttt{size}(s) - 1$.

We want no limits on size. The sequence can grow as needed. Now let's talk about implementation options: *linked list*, which is easy to grow, but slow to index. The other option is *array*, which is fast to index, but hard to grow.

Our approach would be partially-filled heap array.

```
struct Sequence {
    int size; // how many items in use?
    int cap;  // how much can we hold?
    int *theArray;
};
```

How do we structure this as a C module? So we would have a header file, interface, `sequence.h`:

```
struct Sequence {
    int size, cap, *theArray;
}; // bad style, but to save space

struct Sequence emptySeq();
int seqSize(struct Sequence s);
void add(struct Sequence *s, int i, int e);
void remove(struct Sequence *s, int i);
int index(struct Sequence s, int i);
void freeSeq(struct Sequence s);
```

We have `freeSeq` because our `Sequence` is heap allocated. Rather than forcing the user to understand they need to `free theArray`, we provide them with `freeSeq`. Also we can see that some functions take a pointer to `struct Sequence` and the others are not. This is because when we `add` or `remove` from a `Sequence`, the size will change, then we will need to change `s`.

The implementation will go into `sequence.c` file:

```
#include "sequence.h"
/*
    First, this includes struct definition, we then have context for anything else.
    Second, the use of quotes (instead of <>) told us that the file is located in
    this directory. <> standard libraries.
*/

struct Sequence emptySeq() {
    struct Sequence res;
    res.size = 0;
```

```
    res.theArray = malloc(10 * sizeof(int));
    res.cap = 10;
    return res;
}

int seqSize(struct Sequence s) {
    return s.size;
}

void add(struct Sequence *s, int i, int e) {
    for (int n = s->size; n > i; --n) {
        s->theArray[n] = s->theArray[n-1];
    }
    ++s->size;
    s->theArray[i] = e;
}

void remove(struct Sequence *s, int i) {
    for (int n = i; n < s->size-1; ++n) {
        s->theArray[n] = s->theArray[n+1];
    }
    --s->size;
}

int index(struct Sequence s, int i) {
    return s.theArray[i];
}

void freeSeq(struct Sequence s) {
    free(s.theArray);
}
```

Now let's see `main.c`:

```
int main() {
    struct Sequence s = emptySeq();
    add(s, 0, 4);
    add(s, 1, 7);
    ...
}
```

This is ok, but not immune to tampering and forgery. Tampering is accessing the internals of the ADT without going through the functions that provided. Forgery is building an instance of the ADT without using a constructor function that we give. Client then can do

```
s.size = 8; // tampering!

// forgery
struct Sequence t;
t.size = 10;
t.cap = 20;
...
```

Can we prevent this? Qualified yes, but C is not really designed for this kind of protection like some modern languages. The idea is somehow keeping the the details of struct Sequence hidden. Can we declare, but not define, the `struct`? So in `sequence.h`, we just declare the `struct`:

```
struct Sequence;

struct Sequence emptySeq();
int seqSize(struct Sequence s);
void add(struct Sequence *s, int i, int e);
void remove(struct Sequence *s, int i);
int index(struct Sequence s, int i);
void freeSeq(struct Sequence s);
```

Then in `sequence.c`, we define `struct`:

```
#include "sequence.h"
struct Sequence {
    int size, cap, *theArray;
};

// as before
```

In `main.c`,

```
#include "sequence.h"

int main() {
    struct Sequence s = emptySeq();
    ...
}
```

This won't compile. The compiler here doesn't know enough about `Sequence`: the compiler needs to know how big this variable it is before creating on the stack. The compiler only knows `Sequence` exists. However, we can provide pointers. In `sequence.h`,

```
struct SeqImpl;

typedef struct SeqImpl *Sequence; // Sequence = struct SeqImpl*

Sequence emptySeq();

void add(Sequence s, int i, int e);
...
```

In `main.c`,

```
#include "sequence.h"

int main() {
    Sequence s = emptySeq(); // s is a pointer -- OK!
    ...
}
```

Now what happens if the array is full?

```
void add(Sequence s, int i, int e) {
    if (s->size == s->cap) {
        // make the array bigger
        s->theArray = realloc(s->theArray, /* new size */ );
        ...
    }
    ...
}
```

`realloc` increase a block of memory to a new size. If necessary, it allocates a new, larger block and frees the old block (data copied over). The question is then how big should we make it? one larger? We must assume that ech call to `realloc` causes a copy $O(n)$. If we have a sequence of `adds` (at then end, so no shuffling cost), the number of steps would be

$$n \ + \ n+1 \ + \ n+2 \ + \ \cdots \ + \ n+k \ + \ \cdots$$

If done n times, $O(n^2)$ total cost, $O(n)$ per `add`. Similarly, two larger, or three larger don't save us much. What if, instead, we double the size? Each add still $O(n)$ worst case. But we can do *amortized analysis*: we place a bound in a sequence of operations, even if an individual operation may be expensive.

If an array has a cap of k and is empty,

- k inserts at a cost of 1 each (k steps taken).
- 1 insert cost $k+1$ - cap now $2k$ ($k+1$ steps)
- $k-1$ inserts cost 1 each ($k-1$ steps)
- 1 insert costs $2k+1$ - cap now $4k$ ($2k+1$ steps)
- $2k-1$ inserts cost 1 each ($2k-1$ steps)
- 1 insert costs $4k+1$ - cap now $8k$ ($4k+1$ steps)
- $\ldots$
- 2^{j-1} inserts cost 1 each ($2^{j-1}k-1$ steps)
- 1 insert cost 2^jk+1 - cap now $2^{j+1}k$ (2^jk+1 steps taken)

Total insertions: $k+1+(k-1)+1+(2k-1)+1+\cdots+(2^{j-1}k-1)+1 = 2^jk+1$.

Total steps:

$$k+(k+1)+(k-1)+(2k+1)+(2k-1)+(4k+1)+\cdots+(2^{j-1}k-1)+(2^jk+1) = 3\cdot 2^jk-k+1$$

Then number of steps per insertion:

$$\frac{3\cdot 2^jk-k+1}{2^jk+1} \approx 3$$

Therefore, doubling capacity provides for $O(1)$ amortized time insertions (at the end)