CS 136: Elementary Algorithm Design and Data Abstraction

Official calendar entry: This course builds on the techniques and patterns learned in CS 135 while making the transition to use of an imperative language. It introduces the design and analysis of algorithms, the management of information, and the programming mechanisms and methodologies required in implementations. Topics discussed include iterative and recursive sorting algorithms; lists, stacks, queues, trees, and their application; abstract data types and their implementations.
Welcome to CS 136 (Fall 2017)

Instructors: Dan Holtby and Tim Brecht

Web page:
http://www.student.cs.uwaterloo.ca/~cs136/

Other course personnel: ISAs (Instructional Support Assistants), IAs (Instructional Apprentices), ISC (Instructional Support Coordinator): see website for details

Lectures: Tuesdays and Thursdays

Tutorials: Mondays

Be sure to explore the course website: Lots of useful info!
About me (your instructor)
Programming languages

Most of this course is presented in the C programming language.

While time is spent learning some of the C syntax, this is not a “learn C” course.

We present C language features and syntax only as needed.

We continue to use Racket (a dialect of Scheme) to illustrate concepts and highlight the similarities (and differences) between the two languages.

What you learn in this course can be transferred to most languages.
Programming environment (Seashell)

We use our own customized “Seashell” development environment.

• browser-based for platform independence
• works with both C and Racket
• integrates with our submission & testing environment
• helps to facilitate your own testing

See the website and attend tutorials for how to use Seashell.
Course materials

Textbooks:

- “C Programming: A Modern Approach” (CP:AMA) by K. N. King. (strongly recommended)
- “How to Design Programs” (HtDP) by Felleisen, Flatt, Findler, Krishnamurthi (very optional)

Available for free online: http://www.htdp.org

Course notes:

Available on the web page and as a printed coursepack from media.doc (MC 2018).
Several different styles of “boxes” are used in the notes:

**Important information appears in a thick box.**

Comments and “asides” appear in a thinner box. Content that only appears in these “asides” will **not appear on exams**.

Additional **"advanced"** material appears in a “dashed” box.

The advanced material enhances your learning and may be discussed in class and appear on assignments, but you are **not responsible for this material on exams** unless your instructor explicitly states otherwise.
Marking scheme

- 20% assignments (roughly weekly)
- 5% participation
- 25% midterm
- 50% final

To pass this course, you must pass both the assignment component and the weighted exam component.
Class participation

We use i>Clickers to encourage active learning and provide real-time feedback.

- i>Clickers are available for purchase at the bookstore
- Any physical i>Clicker can be used, but we do not support web-based clickers (e.g., i>Clicker Go)
- Register your clicker ID in Assignment 0
- To receive credit you must attend your registered lecture section (you may attend any tutorial section)
- Using someone else’s i>Clicker is an academic offense
Participation grading

- 2 marks for a correct answer, 1 mark for a wrong answer
- Your best 75% responses (from the entire term) are used to calculate your 5% participation grade
- For each tutorial you attend, we’ll increase your 5% participation grade 0.1% (up to 1.2% overall, you cannot exceed 5%)

To achieve a perfect participation mark

- answer 75% of all clicker questions correctly, or
- answer $\approx 40\%$ of all clicker questions correctly, and attend every tutorial
Assignments

Assignments are weekly (approximately 10 per term).

Each assignment is weighted equally (except A0).

Make sure you read the assignment instructions carefully. Each assignment may have different instructions and requirements.

A0 does not count toward your grade, but must be completed before you can receive any other assignment marks.
Assignment *questions* are colour-coded as either “black” or “gold” to indicate if any collaboration is permitted.

For **BLACK** questions, *moderate collaboration* is permitted:

- You can discuss assignment *strategies* openly (including online)
- You can search the Internet for strategies or code examples
• You can discuss your code with *individuals*, but **not** online or electronically (piazza, facebook, github, email, IM, *etc.*)

• You can show your code to others to help them (or to get help), but copying code is not allowed (electronic transfer, copying code from the screen, printouts, *etc.*)

If you submit any work that is not your own, you must still cite the origin of the work in your source code.
For **GOLD** questions, **no collaboration** is permitted:

- **Never share or discuss your code**
- Do not discuss assignment *strategies* with fellow students
- Do not search the Internet for strategies or code examples

You may discuss your code with course staff.

**Academic integrity is strictly enforced for gold questions.**
Assignments: second chances

Assignment deadlines are strict, but some assignment questions may be granted a “second chance”.

- Second chances are granted automatically by an automated “oracle” that considers the quantity and quality of the submissions
- Don’t ask in advance if a question will be granted a second chance; we won’t know
- Second chances are (typically) due 48 hours after the original
- Your grade is: \( \max(\text{original}, \frac{\text{original} + \text{second}}{2}) \)
  (there is no risk in submitting a second chance)
Marmoset

Assignments are submitted to the Marmoset submission system: http://marmoset.student.cs.uwaterloo.ca/

There are two types of Marmoset tests:

- **Public** *(basic / simple)* tests results are available immediately and ensure your program is “runnable”.

- **Private** *(comprehensive / correctness)* tests are available after the deadline and fully assess your code.

Public tests do **not** thoroughly test your code.
• Marmoset uses the best result from all of your submissions (there is never any harm in resubmitting).

• For questions that are *hand-marked*, the most recent submission (before the deadline) with the highest score is marked.

• You can *submit* your assignments via Seashell and view *public* test results.

• Every submission is stored (backed up) for your convenience.

**You must log into Marmoset to view your **private** test results (after the deadline).**
Design recipe

In CS 135 you were encouraged to use the *design recipe*, which included: contracts, purpose statements, examples, tests, templates, and data definitions.

The design recipe has two main goals:

- to help you **design** new functions from scratch, and
- to aid **communication** by providing **documentation**.

In this course, you should already be comfortable designing functions, so we focus on **communication** (through documentation).
Documentation

In this course, every function you write must have:

- a **purpose** statement, and
- a **contract** (including a **requires** section if necessary).

Unless otherwise stated, you are **not** required to provide: templates, data definitions or examples.

Later, we extend contracts to include *effects* and *time* (speed / efficiency).
Hand-marking

Questions that are hand-marked for "style" may be evaluated for:

- documentation and comments
- code readability
- whitespace and indentation
- identifiers (variable & function names)
- appropriate use of helper functions
- testing methodology
The purpose of hand-marking is not to “punish” or “torture” you. It is **formative feedback** to improve your learning.

Unfortunately, we do not have the resources (staff) to hand-mark all assignment questions.

Well formatted and documented code is still expected, even if it is not hand-marked.

We will not provide assistance (office hours or piazza) if your code is poorly formatted or undocumented.

View your formative feedback on MarkUs.
Testing strategies

You are expected to test your own code.

Simply relying on the public marmoset tests is not a viable strategy to succeed in this course.

There are two key testing strategies used in this course:

• assertions

• I/O
Assertion-based testing

In assertion-based testing, the main program is a sequence of function calls, each “asserted” to be correct.

A successful run passes all tests and does “nothing”.

This is used in CS 135:

```
(define (my-sqr n)
  (* n n))
```

```
(check-expect (my-sqr 5) 25)
(check-expect (my-sqr -3) 9)
```

In CS 136 we do not use `check-expect` (alternatives are discussed later).
I/O

This course introduces *Input and Output (I/O)*.

*Informally*, you might say that a function has “inputs” and “outputs” (if you “input” 5 into `my-sqr` it will “output” 25).

We avoid this casual use of “input” and “output”.

The function `my-sqr` is *passed* (or *consumes*) the *argument* 5 and *returns* (or *produces*) the value 25.

In this course, **input** is read from a keyboard (or a file), and **output** is displayed to the screen (or a file).
I/O testing

In I/O-based testing, the main program reads *input* from a *test file* (.in file) and then generates output. The output is then checked against the expected results (.expect file).

For example, consider a program that reads in integers from input and then outputs the square of each integer.

An I/O test for such a program would be:

```
mytest.in     mytest.expect
5             25
-3            9
```

Our Seashell environment supports multiple test files. Each test is run independently.
Getting help

- office hours (see website)
- lab hours (see website)
- tutorials (see website)
- textbook
- piazza

Course announcements are made on piazza and are considered mandatory reading.
Piazza etiquette

• read the official assignment post before asking a question

• search to see if your question has already been asked

• use meaningful titles

• ask clarification questions for assignments
  (do not ask leading questions for GOLD questions)

• do not discuss strategies for GOLD questions

• do not post any of your assignment code publicly

• you can post your commented code privately, and an ISA or Instructor may provide some assistance.
Appendix

The notes also include an appendix, which contains additional content, examples and language syntax details that may not be covered in the lectures. Some of this content will be covered in tutorials.

At this point in the course, you should review:

- Appendix A.1: A Review of CS 135
- Appendix A.2: Full Racket

You are still responsible for content in the Appendix, even if it is not presented in the lectures.
Main topics & themes

- imperative programming style
- elementary data structures & abstract data types
- modularization
- memory management & state
- introduction to algorithm design & efficiency
- designing “medium” sized, “real world” programs with I/O

In this Section we introduce some of the main topics.
Three of the most common programming paradigms are functional, imperative and object-oriented. The first three CS courses at Waterloo use different paradigms to ensure you are “well rounded” for your upper year courses. Each course incorporates a wide variety of CS topics and is much more than the paradigm taught.

CS 135 ⇒ CS 136 ⇒ CS 246

functional    imperative    object-oriented
At the end of each Section there are *learning goals* for the Section (in this Section, we present the learning goals for the entire course). These learning goals clearly state what our expectations are.

Not all learning goals can be achieved just by listening to the lecture. Some goals require reading the text or using Seashell to complete the assignments.
Course learning goals

At the end of this course, you should be able to:

- produce well-designed, properly-formatted, documented and tested programs of a moderate size (200 lines) that can use basic I/O in both Racket and C
- use imperative paradigms (e.g., mutation, iteration) effectively
- explain and demonstrate the use of the C memory model, including the explicit allocation and deallocation of memory
- explain and demonstrate the principles of modularization and abstraction
● implement, use and compare elementary data structures (structures, arrays, lists and trees) and abstract data type collections (stacks, queues, sequences, sets, dictionaries)

● analyze the efficiency of an algorithm implementation
A Functional Introduction to C

Readings: CP:AMA 2.2–2.4, 2.6–2.8, 3.1, 4.1, 5.1, 9.1, 9.2, 10.1, 15.2

- the ordering of topics is different in the text
- some portions of the above sections have not been covered yet
A brief history of C

C was developed by Dennis Ritchie in 1969–73 to make the Unix operating system more portable.

It was named “C” because it was a successor to “B”, which was a smaller version of the language BCPL.

C was specifically designed to give programmers “low-level” access to memory (discussed in Section 05 and Section 06).

It was also designed to be easily translatable into “machine code” (discussed in Section 13).
Today, thousands of popular programs, and portions of all of the popular operating systems (Linux, Windows, Mac OS X, iOS, Android) are written in C.

There are a few different versions of the C standard. In this course, the C99 standard is used.
From Racket to C

To ease the transition from Racket, we will first learn to write some simple C functions using a functional paradigm.

This allows us to become familiar with the C syntax without introducing too many new concepts.

in Section 04 imperative programming concepts are introduced.

Read your assignments carefully: you may not be able to “jump ahead” and start programming with imperative style (e.g., with mutable variables or loops).
Comments

; Racket comment // C comment

#| Racket multi-line /* C multi-line
   comment |#
   comment */

In C, any text on a line after // is a comment.

Any text between /* and */ is also a comment, and can extend over multiple lines. This is useful for commenting out a large section of code.

C’s multi-line comment cannot be “nested”:
/* this /* nested comment is an */ error */
Defining a variable (constant)

; Racket Constant:
(define my-number 42)

// C Constant:
const int my_number = 42;

In C, a semicolon (;) is used to indicate the end of a variable definition.

The `const` keyword indicates the variable is immutable (`constant`).

The `int` keyword declares that `my_number` is an `integer`. 
In this course, the term “variable” is used for both variable and constant identifiers.

In the few instances where the difference matters, we use the terms “mutable variables” (introduced in Section 04) and “constants”.

In this Section, all variables are constants.
C identifiers ("names") are more limited than in Racket. They must start with a letter, and can only contain letters, underscores and numbers (*my*_number instead of *my*-number).

We use _underscore_style_, but _camelCaseStyle_ is a popular alternative. Consistency is more important than the choice of style.

C identifiers can start with a leading underscore (_name_) but their use is restricted and they may interfere with reserved keywords. Avoid them in this course as they may interfere with marmoset tests.
Typing

Racket and C handle types very differently.

- Racket uses **dynamic typing**: the type of an identifier is determined while the program is running.

  ```scheme
  ; dtype can be a number or a string
  (define dtype (cond [(>= x 0) 42]
                     [else "invalid"]
  ))
  ```

- C uses **static typing**: the type of an identifier must be known before the program is run. The type is declared in the definition and cannot change.

  ```c
  // stype is always a const int
  const int stype = 42;
  ```
C Types

To start, we will only work with **integers** in C.

More types will be introduced soon.

Because C uses static typing, there are no functions equivalent to the Racket type-checking functions (e.g., `integer?` and `string?`).
Initialization

// C Constant:
const int my_number = 42;

The “= 42” portion of the above definition is called initialization.

Constants must be initialized.
Expressions

C uses the more familiar *infix* algebraic notation \((3 + 3)\) instead of the *prefix* notation used in Racket \((+ 3 3)\).

; Racket Expressions:  // C Expressions:
(define six (+ 3 3))  const int six = 3 + 3;
(define seven (+ 1 (* 3 2)))  const int seven = 1 + 3 * 2;
(define eight (* (+ 1 3) 2))  const int eight = (1 + 3) * 2;

With infix notation, parentheses are often necessary to control the order of operations.

Use parentheses to clarify the order of operations in an expression.
C operators

C distinguishes between operators (e.g., +, -, *, /) and functions.

The C order of operations ("operator precedence rules") are consistent with mathematics: multiplicative operators (*, /) have higher precedence than additive operators (+, -).

In C there are also non-mathematical operators (e.g., for working with data) and almost 50 operators in total.

As the course progresses more operators are introduced.

The full order of operations is quite complicated (see CP:AMA Appendix A).
In C, each operator is either *left* or *right* associative to further clarify any ambiguity (see CP:AMA 4.1).

The multiplication operators are *left*-associative:

\[ 4 \times 5 \div 2 \text{ is equivalent to } (4 \times 5) \div 2. \]

The distinction in this particular example is important in C.
The / operator

When working with integers, the C division operator (/) truncates (rounds toward zero\(^\dagger\)) any intermediate values, and behaves the same as the Racket **quotient** function.

\[
\text{const int } a = (4 \times 5) / 2; \quad // \; 10 \\
\text{const int } b = 4 \times (5 / 2); \quad // \; 8 !! \\
\text{const int } c = -5 / 2; \quad // \; -2 !!
\]

Remember, use parentheses to clarify the order of operations.

\(^\dagger\) C99 standardized the “(round toward zero)” behaviour.
The % operator

The C remainder operator (%) (also known as the modulo operator) behaves the same as the remainder function in Racket.

```
const int a = 21 % 2; // 1
const int b = 21 % 3; // 0
const int c = 13 % 5; // 3
```

In this course, avoid using % with negative integers.

In C99, \( (i \% j) \) has the same sign as \( i \).
(see CP:AMA 4.1 for more details).
Function terminology

In this course, we use a function terminology that is more common amongst imperative programmers.

In our “functional” CS 135 terminology, we apply a function. A function consumes arguments and produces a value.

In our new terminology, we call a function. A function is passed arguments and returns a value.
Function definitions

; Racket function:  // C function:
; my-sqr: Int -> Int int my_sqr(int n) {
(define (my-sqr n) return n * n;
  (* n n))
}

In C, braces ({}) indicate the beginning and end of the function body (or the function block).

The return keyword is placed before the expression to be returned.

Because C is statically typed, the function return type and the type of each parameter is required.

The return type of my_sqr is an int, and it has an int parameter n.
int my_sqr(int n) {
    return n * n;
}

In the definition of `my_sqr` we did not use the `const` keyword, although we could have in two places:

```c
const int my_sqr(const int n) {
    return n * n;
}
```

C99 ignores the `const int` return type, so we will not use it.

The `const int` parameter is often good style. However, we will avoid using `const` for parameters until we have a deeper understanding of their meaning.
int my_add(int x, int y) {
    return x + y;
}

int my_num(void) {
    return my_add(40, 2);
}

Parameters are separated by a comma (,) in the function definition and calling syntax.

The void keyword is used to indicate a function has no parameters.
If you omit the `void` in a parameterless function definition:

```c
int my_num() {
    // ...
}
```

C will allow it. This is because (`()` is used in a special syntax to indicate an “unknown” or “arbitrary” number of parameters (that is beyond the scope of this course).

It is better style to use `void` to clearly communicate and enforce that there are no parameters.

```c
int my_num(void) {
    // ...
}
```
Coding Style and Formatting

Code is formatted by including white space (tabs, spaces, new lines and blank lines in appropriate places). It is used to make your code easy to read which is EXTREMELY IMPORTANT!!

// Valid but not very pretty (i.e., not properly formatted)
int my_add(int x,
int y){return x+
y;}int my_num(void) {return my_add(40,2);}

Be sure to follow the conventions used in this class when formatting your code. Instructors and TAs will find it very difficult to help you with your code if you do not use proper style when writing your code.
Function documentation

; (my-sqr n) squares n
; my-sqr: Int -> Int

(define (my-sqr n)
  (* n n))

In C, the contract types are part of the function definition. No additional contract documentation is necessary. However, you should still add a requires section if appropriate.

// some_function(n) ....
// requires: n > 0

int some_function(int n) {
  // ...
}

int my_sqr(int n) {
  return n * n;
}
In Racket, the contract types are only documentation. It is possible to violate the contract, which may cause a “type” runtime error.

```racket
;; Racket:
(my-sqr "hello") ; => runtime error !!!
```

In statically typed languages like C, it is impossible to violate the contract type, and “type” runtime errors do not exist.

```c
// C:
my_sqr("hello") // => will not run !!!
```
Getting started

At this point you are probably eager to write your own functions in C. Unfortunately, we do not have an environment similar to DrRacket’s interactions window to evaluate expressions and informally test functions.

Next, we will demonstrate how to run and test a simple C program that can display information.
Entry point

Typically, a program is “run” (or “launched”) by an Operating System (OS) through a shell or another program such as DrRacket.

The OS needs to know where to start running the program. This is known as the entry point.

In many interpreted languages (including Racket), the entry point is simply the top of the file you are “running”.

In C, the entry point is a special function named main.

Every C program must have one (and only one) main function.
main

main has no parameters\(^\dagger\) and an \texttt{int} return type.

\begin{verbatim}
int main(void) {
    //...
}
\end{verbatim}

The return value communicates to the OS if the program is successful (zero) or if an error occurred (non-zero). The return value is \textit{optional}, and defaults to zero.

For this class, \texttt{main} does \textbf{not} require any documentation. However, it is generally a good idea to include a comment explaining what the program does.

\(\dagger\) \texttt{main} has \textit{optional} parameters (discussed in Section 13).
Hello, World

To display **output** in C, we use the **printf** function.

```c
// hello.c : my first C program

int main(void) {
    printf("Hello, World");
    return 0;
}
```

This may not work: we need to “require” or “include” the module that contains the **printf** function.

---

Seashell may be kind and include it for us, but it will give a warning message.
printf is part of the C stdio (standard i/o) module. We will introduce C modules later but for now you can think of stdio (and a module) as a collection of functions. In this case include <stdio.h> is needed to use the function printf.

// hello2.c : my second C program
#include <stdio.h> // <-- require stdio module

int main(void) {
    printf("Hello, World");
    return 0; // <-- can be omitted
} // (if returning 0)

Hello, World

In the slides, we typically omit #includes to save space.
int main(void) {
    printf("Hello, World");
    printf("C is fun!");
}

Hello, WorldC is fun!

The **newline** character (\n) is necessary to properly format your output to appear on multiple lines.

```
printf("Hello, World\n");
printf("C is
fun!");
```

Hello, World
C is
fun!
The first parameter of `printf` must be a "string". Until we discuss strings in Section 08, this is the only place you are allowed to use strings.

We can output other value types by using a `placeholder` within the string and providing an additional parameter.

```
printf("2 plus 2 is: %d\n", 2 + 2);
```

```
2 plus 2 is: 4
```

The "%d" placeholder is replaced with the value of the additional parameter. There can be multiple placeholders, each requiring an additional parameter.

```
printf("%d plus %d is: %d\n", 2, 2, 2 + 2);
```

```
2 plus 2 is: 4
```
C uses different placeholders for each type. The placeholder we use for integers is "%d" (which means “decimal format”).

To output a percent sign (%), use two (%%).

```c
printf("I am %d%% sure you should watch your", 100);
printf("spacing!\n");
```

I am 100% sure you should watch yours.

Similarly, to print a backslash (\), use two (\\).

We will discuss I/O in more detail in Section 07
Many computer languages have a `printf` function and use the same placeholder syntax as C. The placeholders are also known as `format` specifiers (the `f` in `printf`).

The full C `printf` placeholder syntax allows you to control the format and align your output.

```c
printf("4 digits with zero padding: %04d\n", 42);
4 digits with zero padding: 0042
```

See CP:AMA 22.3 for more details.

In this course, simple "\%d" formatting is usually sufficient.
Boolean expressions

In C, “false” is represented by zero (0) and “true” is represented by one (1). Any non-zero value is also considered “true”.

The **equality operator** in C is `==` (note the *double* equals).

The **not equal operator** is `!=`.

\[
\begin{align*}
(3 \ == \ 3) & \Rightarrow \ 1 \ (\text{true}) \\
(2 \ == \ 3) & \Rightarrow \ 0 \ (\text{false}) \\
(2 \ != \ 3) & \Rightarrow \ 1 \ (\text{true}) \\
\end{align*}
\]

Always use a *double* `==` for equality, not a *single* `=`.
The accidental use of a *single* = instead of a *double* == for equality is one of the most common programming mistakes in C. This can be a serious bug (we revisit this in Section 04). It is such a serious concern that it warrants an extra slide as a reminder.
The *not, and and or* operators are respectively `!`, `&&` and `||`.

\[
\begin{align*}
!(3 == 3) & \Rightarrow 0 \\
(3 == 3) && (2 == 3) & \Rightarrow 0 \\
(3 == 3) && !(2 == 3) & \Rightarrow 1 \\
(3 == 3) || (2 == 3) & \Rightarrow 1 \\
(2 && 3 || 0) & \Rightarrow 1 \\
\end{align*}
\]

Similar to Racket, C *short-circuits* and stops evaluating an expression when the value is known. 

\[
(a != 0) && (b / a == 2)
\]

does not produce an error if \( a \) is 0.

A common mistake is to use a single `&` or `|` instead of `&&` or `||`. These operators have a different meaning.
Comparison operators

The operators <, <=, > and >= behave exactly as you would expect.

\[
\begin{align*}
(2 &< 3) \implies 1 \\
(2 &\geq 3) \implies 0
\end{align*}
\]

\!(a < b)\! is equivalent to \!(a \geq b)\!.

It is always a good idea to add parentheses to make your expressions clear.

> has higher precedence than ==, so the expression

\[
1 == 3 > 0 \text{ is equivalent to } 1 == (3 > 0), \text{ but it could easily confuse some readers.}
\]
Conditionals

There is no direct C equivalent to Racket’s `cond` special form.

We can use C’s `if` statement to write a function that has conditional behaviour.

```c
int my_abs(int n) {
    if (n < 0) {
        return -n;
    } else {
        return n;
    }
}
```

There can be more than one `return` in a function.
The cond special form consumes a sequence of question and answer pairs (questions are Boolean expressions).

Racket functions that have the following cond behaviour can be re-written in C using if, else if and else:

```
(define (my-function ...)  
  (cond  
    [q1   a1]  
    [q2   a2]  
    [else a3])))
```

```
int my_function(...) {  
  if (q1) {  
    return a1;  
  } else if (q2) {  
    return a2;  
  } else {  
    return a3;  
  }
}
```
Recursion in C behaves the same as in Racket.
cond produces a value and can be used inside of an expression:

\[
(+ \ y \ (\text{cond} \ [(< \ x \ 0) \ -x] \\
\text{[else} \ \ x])
\]

C’s if statement does not produce a value: it only controls the “flow of execution” and cannot be similarly used within an expression.

We revisit if in Section 05 after we understand how “statements” differ from expressions. For now, only use if as we have demonstrated:

```c
if (q1) {
    return a1;
} else if (q2) {
    return a2;
} else {
    return a3;
}
```
Unlike C’s \textit{if statement}, the C \texttt{? :} \textit{operator} does produce a value and behaves the same as Racket’s \texttt{if} special form.

\begin{verbatim}
;; Racket’s if special form:
(define c (if q a b))
(define abs-v (if (>= v 0) v (- v)))
(define max-ab (if (> a b) a b))
\end{verbatim}

The value of \texttt{(q ? a : b)} is \texttt{a} if \texttt{q} is true (non-zero), and \texttt{b} otherwise.

\begin{verbatim}
// C’s \texttt{?:} operator
const int c = q ? a : b;
const int abs_v = (v >= 0) ? v : -v;
const int max_ab = (a > b) ? a : b;
\end{verbatim}
This example illustrates the importance of ordering (placing) C functions in a file. Because `sum` calls `accsum`, we placed `accsum` before (or “above”) `sum` in the code.
In C, a function (or any identifier) must be declared before (or “above”) any expression it appears in.

A declaration communicates to C the type of an identifier.

Function declarations include both the return type and the parameter type(s).

In C, there is a subtle difference between a definition and a declaration.
Declaration vs. definition

- A declaration only specifies the *type* of an identifier.
- A definition instructs C to “create” the identifier.

However, a definition must also specify the type of the identifier, so

**a definition also includes a declaration.**

An identifier can be declared multiple times, but only defined once.

Unfortunately, not all computer languages and reference manuals use these terms consistently.
A function declaration is a header without a body, with just a semicolon (;) instead of a code block.

**example: function declaration**

```c
int accsum(int k, int acc); // DECLARATION

int sum(int n) {
    return accsum(n, 0); // this is now ok
}

int accsum(int k, int acc) { // DEFINITION
    if (k == 0) {
        return acc;
    } else {
        return accsum(k - 1, k + acc);
    }
}
```

CS 136 Fall 2017 02: A Functional Intro to C
C ignores the parameter names in a function declaration (it is only interested in the parameter *types*).

The parameter names can be different from the definition or not present at all.

```c
// These are all equivalent:

int accsum(int k, int acc);
int accsum(int, int);
int accsum(int ignored, int ignored);
```

It is good style to include the correct parameter names in the declaration to aid communication.
A **variable declaration** starts with the `extern` keyword and is not initialized.

**example: variable declaration**

```c
extern const int fun_number;       // variable DECLARATION

int is_fun(int n) {
    if (n == fun_number) {       // this is now ok
        return 1;
    } else {
        return 0;
    }
}

const int fun_number = 4010;       // variable DEFINITION
```

Variable declarations are uncommon.
bool type

Boolean types are not “built-in” to C, but are available through the stdbool standard module.

stdbool provides a new bool type (can only be 0 or 1) and defines the constants true (1) and false (0).

```c
#include <stdbool.h>

const bool is_cool = true;

bool is_even(int n) {
    return (n % 2) == 0;
}
```
Symbol type

In C, there is no equivalent to the Racket `symbol` type. To achieve similar behaviour in C, you can define a unique integer for each “symbol”. It is common to use an alternative naming convention (such as ALL_CAPS).

; Racket Symbols:  // C’s alternative:
(const int POP = 1;
(const int ROCK = 2;
(define genre1 'pop)  (define int genre1 = POP;
(define genre2 'rock)  (const int genre2 = ROCK;
In C, there are enumerations (enum, CP:AMA 16.5) which allow you to create your own enum types and help to facilitate defining constants with unique integer values.

Enumerations are an example of a C language feature that we do not introduce in this course.

After this course, we would expect you to be able to read about enums in a C reference and understand how to use them.

If you would like to learn more about C or use it professionally, we recommend reading through all of CP:AMA after this course is over.
Floating point types

The C float (floating point) type can represent real (non-integer) values.

```c
const float pi = 3.14159;
const float avagadro = 6.022e23;  // 6.022*10^23
```

Unfortunately, floats are susceptible to precision errors.

C’s float type is similar to **inexact numbers** in Racket (which appear with an **#i** prefix in the teaching languages):

```racket
(sqrt 2)  ; => #i1.4142135623730951
(sqr (sqrt 2))  ; => #i12.0000000000000004
```
example 1: inexact floats

```c
const float penny = 0.01;

float add_pennies(int n) {
    if (n == 0) {
        return 0;
    } else {
        return penny + add_pennies(n-1);
    }
}

int main(void) {
    const float dollar = add_pennies(100);
    printf("the value of one dollar is: %f\n", dollar);
}

the value of one dollar is: 0.999999
```

The `printf` placeholder to display a float is "%f".
example 2: inexact floats

```c
int main(void) {
    const float bil = 1000000000;
    const float bil_and_one = bil + 1;
    printf("a float billion is: %f\n", bil);
    printf("a float billion + 1 is: %f\n", bil_and_one);
}
```

a float billion is: 1000000000.000000
a float billion + 1 is: 1000000000.000000
In the previous two examples, we highlighted the precision errors that can occur with the `float` type.

C also has a `double` type that is still inexact but has significantly better precision.

Just as we use `check-within` with inexact numbers in Racket, we can use a similar technique for testing in floating point numbers C.

Assuming that the precision of a `double` is perfect or “good enough” can be a serious mistake and introduce errors.

Unless you are explicitly told to use a `float` or `double`, you should not use them in this course.
Structures

Structures (compound data) in C are similar to structures in Racket:

```
(struct posn (x y) #:transparent)
```

```
struct posn {
    int x;
    int y;
};
```

```
(define p (posn 3 4))
```

```
const struct posn p = {3,4};
```

Racket generates functions (e.g., `posn-x`) when you define a structure, but C does not.
Because C is statically typed, structure definitions require the type of each field.

The structure type includes the keyword “struct”. For example, the type is “struct posn”, not just “posn”. This can be seen in the definition of p above.

Do not forget the last semicolon (;) in the structure definition.
Instead of selector functions, C has a structure operator \((.)\) which "selects" the value of the requested field.
C99 supports an alternative way to initialize structures:

```c
const struct posn p = { .y = 4, .x = 3};
```

This prevents you from having to remember the “order” of the fields in the initialization.

Any omitted fields are automatically zero, which can be useful if there are many fields:

```c
const struct posn p = {.x = 3}; // .y = 0
```
The equality operator (==) does not work with structures. You have to define your own equality function.

```c
bool posn_equal (struct posn a, struct posn b) {
    return (a.x == b.x) && (a.y == b.y);
}
```

Also, printf only works with elementary types. You have to print each field of a structure individually:

```c
const struct posn p = {3,4};
printf("The value of p is (%d,%d)\n", p.x, p.y);
```

The value of p is (3,4)
The braces ({} ) are part of the initialization syntax and should not be used inside of an expression.

```c
struct posn scale(struct posn p, int f) {
    return {p.x * f, p.y * f};  // INVALID
}
```

In the past, students more familiar with Racket and functional-style structures have found this restriction frustrating.

To avoid this, simply define new constants as required.

```c
struct posn scale(struct posn p, int f) {
    const struct posn r = {p.x * f, p.y * f};
    return r;
}
```
Goals of this Section

At the end of this section, you should be able to:

- demonstrate the use of the C syntax and terminology introduced
- define constants and write expressions in C
- re-write a simple Racket function in C (and vice-versa)
- use the C operators introduced in this module (including `% == != >= && || .)
- explain the difference between a declaration and a definition
• explain the significance of the main function in C
• perform basic testing and I/O in C using assert and printf
• use structures in C
• provide the required documentation for C code
Modularization & ADTs

Readings: CP:AMA 19.1, 10.2-10.5
Modularization

In previous courses we designed small programs with definitions in a single Racket (.rkt) file.

For larger programs, keeping all of the code in one file is unwieldy. Teamwork on a single file is awkward, and it is difficult to share or re-use code between programs.

A better strategy is to use modularization to divide programs into well defined modules.
The concept of modularization extends far beyond computer science. You can see examples of modularization in construction, automobiles, furniture, nature, etc.

A practical example of a good modular design is an electrical outlet.

You don’t have to know how power is provided to the outlet (in fact it has changed over the years) and many devices can be built with a plug that can use the power. A key is a well specified, standard interface that does not change and can be used by a large variety of devices.
We have already seen an elementary type of modularization in the form of *helper functions* that can “help” many other functions.

We will extend and formalize this notion of modularization.

When designing larger programs, we move from writing “helper functions” to writing “helper modules”.

A module provides a collection of functions† that share a common aspect or purpose.

† Modules can provide elements that are not functions (e.g., data structures and variables) but their primary purpose is to provide functions.

For convenience in these notes, we describe modules as providing only functions.
Modules vs. files

In this course, and in much of the “real world”, it is considered **good style** to store each module in a separate file.

While the terms *file* and *module* are often used interchangeably, a file is only a module if it provides functions for use outside of the file.

Some computer languages enforce this relationship (one file per module), while in others it is only a popular *convention*.

There are advanced situations (beyond the scope of this course) where it may be more appropriate to store multiple modules in one file, or to split a module across multiple files.
Terminology

It is helpful to think of a “client” that requires the functions that a module provides.

In practice, the client is a file that may be written by yourself, a co-worker or even a stranger.

Conceptually, it is helpful to imagine the client as a separate entity.
Large programs can be built from many modules.

A module can be a client itself and require functions from other modules.

The *module dependency graph* cannot have any cycles.

There must be a “root” (or *main file*) that acts only as a client. This is the program file that is “run”.
Motivation

There are three key advantages to modularization: re-usability, maintainability and abstraction.

**Re-usability:** A good module can be re-used by many clients. Once we have a “repository” of re-usable modules, we can construct large programs more easily.

**Maintainability:** It is much easier to test and debug a single module instead of a large program. If a bug is found, only the module that contains the bug needs to be fixed. We can even replace an entire module with a more efficient or more robust implementation.
**Abstraction:** To use a module, the client needs to understand what functionality it provides, but it does not need to understand how it is implemented. In other words, the client only needs an “abstraction” of how it works. This allows us to write large programs without having to understand how every piece works.

*Modularization* is also known in computer science as the *Separation of Concerns (SoC).*
C modules

Unlike Racket, there are no “built-in” functions in C.

Fortunately, C provides several \textit{standard modules} (also known as \textit{libraries}) with many useful functions.

We have already seen how the \texttt{stdio} standard module provides the \texttt{printf} function.

We will use several \textit{standard modules} throughout this course, including \texttt{assert, stdbool, limits, string} and \texttt{stdlib}.
To **require** the `stdio` module, we wrote:

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

The angle brackets (`<>`) specify that the module is one of the **standard modules**.

The `.h` stands for header (i.e., we include a header file).

To **require** a “regular” module (i.e., one you have written) the syntax is slightly different. Quotes (`""`) are used instead of angle brackets:

```c
#include "mymodule.h"
```

The different notations (`<stdio.h>` vs. "mymodule.h") tell C **where** to look for the module interface.
Creating a module in C

To create a module in C, we place the **interface** and the **implementation** into separate files.

In the **interface (.h) file** we place **declarations** for the functions (and variables) that the module **provides**. We also place the documentation for the client in the interface file.

In the **implementation (.c) file** we place all of the **definitions**.
// mymodule.h [INTERFACE]
extern const int my_constant;

// my_sqr(n) squares n
int my_sqr (int n);

// mymodule.c [IMPLEMENTATION]
const int my_constant = 7;

// see mymodule.h for details
int my_sqr(int n) {
    return n * n;
}

// client.c [CLIENT]

#include <stdio.h>
#include "mymodule.h"

int main(void) {
    printf("my_constant squared is %d\n", my_sqr(my_constant));
}

my_constant squared is 49
#include

#include is what is known as a preprocessor directive. Directives can modify a source file just before it is run.

#include “inserts” the contents of the interface (.h) file directly into the client source file. This makes all of the provided declarations available to the client.

In Appendix A.4 we discuss #include and C modules in more detail.
One use of `#include` that may not be intuitive is that a module implementation (.c) file can `#include` its own interface.

This is actually very good practice as it ensures there are no discrepancies between the interface and the implementation.

```c
// mymodule.c [IMPLEMENTATION]

#include "mymodule.h" // good practice

const int my_constant = 7;

// see mymodule.h for details
int my_sqr(int n) {
    return n * n;
}
```
The CP:AMA textbook frequently uses the `#define` directive. In its simplest form it performs a *search & replace*.

```c
// replace every occurrence of MY_NUMBER with 42
#define MY_NUMBER 42

int my_add(int n) {
    return n + MY_NUMBER;
}
```

In C99, it is better style to define a variable (constant), but you will still see `#define` in the “real world”.

`#define` can also be used to define *macros*, which are hard to debug, considered poor style, and should be avoided.
assert standard module

A useful standard module is `assert`, which provides the `assert` function.

`assert(e)` stops the program and displays a message if the expression `e` is false. If `e` is true, nothing happens.

`assert` is especially useful for verifying function requirements. You should `assert` your requirements `when feasible`.

```c
#include <assert.h>

// requires: n > 0
int some_function(int n) {
    assert(n > 0);
    //...
}
```
example: test client

It is good practice to create a test client for each module you create. `assert` can be used in a manner similar to Racket’s `check-expect` for assertion-based testing clients.

```
// test-mymodule.c: assertion testing client for mymodule

#include <assert.h>
#include "mymodule.h"

int main(void) {
    assert(my_sqr(0) == 0);
    assert(my_sqr(-1) == 1);
    assert(my_sqr(1) == 1);
    assert(my_sqr(2) == 4);
    assert(my_sqr(3) == 9);
}
```


```c
#include <assert.h>

int main() {
    const int x = 1;
    assert(x == 1);
    assert(x != 0);
    assert(x == 0);
    assert(x == 0);
}
```

Assertion failed: (x == 0), function main, file assert-example.c, line 7.
Scope

Local scope behaviour is the same in both C and Racket. Each C block ({} ) creates a new local environment similar to local in Racket. If an identifier shadows another with the same name, the innermost ("most local") instance is used.

In C, each global identifier has program scope by default.
To declare that a C global function or variable has **module** scope, the **static** keyword is used. **static** indicates that the identifier is “restricted” to the current file.

```c
int pfunction (void) { ... } // program scope
static int mfunction (void) {...} // module scope
const int pvariable = 42; // program scope
static const int mvariable = 23; // module scope
```

In Appendix A.4 we discuss this in more detail.

**private** would have been a better choice than **static**.

The C keyword **static** is not related to **static typing**.
One more scope-related difference between the languages is that in C you cannot have any **top-level expressions**.

The initialization of a global variable can contain a simple expression, but it cannot contain a function call.

```
// C top level:
const int a = 3 * 3; // VALID

const int b = my_sqr(3); // INVALID
3 * 3; // INVALID
my_sqr(3); // INVALID
```

Also, in C99 you cannot define a local function within another function.
example: fun number module

Consider that some integers are more “fun” than others, and we want to create a fun module that provides a fun function.

// fun.h
#include <stdbool.h>
extern bool fun(int n);

// fun.c
#include "fun.h"
bool fun(int n) {
    if (n == 42 || n == 1337 || n == 4010 || n == 8675309) {
        return true;
    } else {
        return false;
    }
}
To use the function(s) provided by the module `fun` we must include the module.

```c
// fun-client.c
#include <stdio.h>
#include "fun.h"

int main() {
    if (fun(4010)) { printf("4010 is fun\n"); } 
    if (fun(5)) { printf("5 is fun\n"); } 
    if (fun(8675309)) { printf("8675309 is fun\n"); } 
    if (fun(100)) { printf("100 is fun\n"); } 
}
```

4010 is fun
8675309 is fun
re-usability: multiple programs can use the fun module.

maintainability: When new numbers become fun (or become less fun), only the fun module needs to be changed.

abstraction: The client does not need to understand what makes an integer fun.
Module interface

The module *interface* is the list of the functions that the module provides.

In practice, the interface also includes the documentation.

The interface is separate from the module *implementation*, which is the code of the module (*i.e.*, function *definitions*).

The interface is everything that a client would need to use the module. The client does not need to see the implementation.
Terminology (revisited)

The **interface** is what is provided to the client.

The **implementation** is hidden from the client.
Interface documentation

Interface documentation includes:

- an **overall description** of the module,
- a **list of functions** it provides, and
- the **contract** and **purpose** for each provided function.

Ideally, the interface should also provide **examples** to illustrate how the module is used and how the interface functions interact.
C module interfaces

For C modules, the interfaces appears in the .h (header) file.

In the implementation, it is not necessary to duplicate the interface documentation for public (program scope) functions that are provided.

For private (module scope) functions, the proper documentation (contract and purpose) should accompany the function definition.
example: sum module

// sum-module.h
// A module for summing numbers [description of module]

// purpose: sums the integers 1..n
// requires: n >= 1
extern int sum_first(int n);

// purpose: sums the squares of integers 1..n
// requires: n >= 1
extern int sum_squares(int n);
// sum-module.c

int sum_first(int n) {
  assert(n >= 1);
  ...
}

int sum_squares(int n) {
  assert(n >= 1);
  ...
}

// purpose: squares an integer (C doesn’t provide this)
// requires: n >= 1
static int square(int n) {
  assert(n >= 1);
  return n * n;
}
Testing

For each module you design, it is good practice to create a *test client* that ensures the *provided* functions are correct.

**example: sum-test.c**

```c
// this is a simple assertion testing client for sum-module
#include "sum-module.h"

int main() {
    assert(sum_first(1) == 1);
    assert(sum_first(2) == 3);
    assert(sum_first(3) == 6);
    assert(sum_first(6) == 21);
    assert(sum_squares(1) == 1);
    assert(sum_squares(2) == 5);
    assert(sum_squares(6) == 91);
}
```
In Section 07 we discuss more advanced testing strategies.

There may be “white box” tests that cannot be tested by a client. These may include implementation-specific tests and tests for module-scope functions.

In these circumstances, you can provide a `module_name_test` function that uses assertions and produces nothing if the tests are successful. If the test fails an assertion will be thrown indicating where the problem was.
Designing modules

The ability to break a big programming project into smaller modules, and to define the interfaces between modules, is an important skill that will be explored in later courses.

Unfortunately, due to the nature of the assignments in this course, there are very few opportunities for you to design any module interfaces.

For now, we will have a brief discussion on what constitutes a good interface design.
Cohesion and coupling

When designing module interfaces, we want to achieve *high cohesion* and *low coupling*.

**High cohesion** means that all of the interface functions are related and working toward a “common goal”. A module with many unrelated interface functions is poorly designed.

**Low coupling** means that there is little interaction *between* modules. It is impossible to completely eliminate module interaction, but it should be minimized.
High coupling  

Low coupling
Interface vs. implementation

We emphasized the distinction between the module interface and the module implementation.

Another important aspect of interface design is information hiding, where the interface is designed to hide any implementation details from the client.

In Racket, the module interface and implementation appear in the same file, so the distinction between interface and implementation may not seem important, but in C (and in many other languages) only the interface is provided to the client.
Information hiding

The two key advantages of information hiding are security and flexibility.

Security is important because we may want to prevent the client from tampering with data used by the module. Even if the tampering is not malicious, we may want to ensure that the only way the client can interact with the module is through the interface. We may need to protect the client from themselves.

By hiding the implementation details from the client, we gain the flexibility to change the implementation in the future.
example: changing fun

// fun.h
bool fun(int n);

// fun.c (original)
#include "fun.h"
bool fun(int n) {
    if (n == 42 || n == 1337 || n == 4010 || n == 8675309) {
        return true;
    } else {
        return false;
    }
}

example: changing fun

// fun.c (new)
#include "fun.h"

bool fun(int n) {
    if (n == 13 || n == 17 || n == 31 || n == 37) {
        return true;
    } else {
        return false;
    }
}
Data structures vs. ADTs

The difference between a *data structure* and an *ADT* is subtle and worth reinforcing.

With a **data structure**, you know how the data is “structured” and you can access the data directly in any manner you desire.

However, with an **ADT** you do not know how the data is structured and you can only access the data through the interface functions (operations) provided by the ADT.

The terminology is especially confusing because the *implementation* of an ADT uses a data structure.
Collection ADTs

A **Collection ADT** is an ADT designed to store an arbitrary number of items. Collection ADTs have well-defined operations and are useful in many applications.

In CS 135 we were introduced to our first *collection ADT*: a dictionary.

In most contexts, when someone refers to an ADT they *implicitly* mean a “collection ADT”.

By some definitions, collection ADTs are the *only* type of ADT.
Dictionary (revisited)

The dictionary ADT (also called a *map, associative array, or symbol table*), is a collection of **pairs** of **keys** and **values**. Each **key** is unique and has a corresponding value, but more than one key may have the same value.

Typical dictionary ADT operations:

- **lookup**: for a given key, retrieve the corresponding value or “not found”
- **insert**: adds a new key/value pair (or replaces the value of an existing key)
- **remove**: *deletes* a key and its value
**example: student numbers**

<table>
<thead>
<tr>
<th>key (student number)</th>
<th>value (student name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234567</td>
<td>&quot;Sally&quot;</td>
</tr>
<tr>
<td>3141593</td>
<td>&quot;Archie&quot;</td>
</tr>
<tr>
<td>8675309</td>
<td>&quot;Jenny&quot;</td>
</tr>
</tbody>
</table>
We can implement a dictionary with an **association list data structure** (a list of key/value pairs with each pair stored as a two-element list).

```
(define al '(((1234567 "Sally") (3141593 "Archie")
  (8675309 "Jenny")))
```

![Diagram of association list with key-value pairs]
Alternatively, we can implement a dictionary with a **Binary Search Tree (BST) data structure.**

```scheme
(define bst (make-node 3141593 "Archie"
         (make-node 1234567 "Sally" empty empty)
         (make-node 8675309 "Jenny" empty empty)))
```

Racket BSTs are briefly reviewed in Appendix A.1.
To *implement* a dictionary, we have a choice: use an association list, a BST or perhaps something else?

This is a *design decision* that requires us to know the advantages and disadvantages of each choice.

You likely have an intuition that BSTs are “more efficient” than association lists.

In Section 09 we explore what it means to be “more efficient”, and introduce a formal notation to describe the efficiency of an implementation.
More collection ADTs

Three additional collection ADTs that will be explored in this course are:

• stack
• queue
• sequence
Stack ADT

The stack ADT is a collection of items that are “stacked” on top of each other. Items are pushed onto the stack and popped off of the stack. A stack is known as a LIFO (last in, first out) system. Only the “top” item is accessible.

Stacks are often used in browser histories ("back") and text editor histories ("undo").

In Section 05 we will see a very practical use for a stack.
Typical stack ADT operations:

- **push**: adds an item to the top stack
- **pop**: removes the top item from the stack
- **top**: returns the top item on the stack
- **is-empty**: determines if the stack is empty
Stack ADT vs. Racket list

The stack ADT is very similar to a Racket list.

The operations: push/pop/top/is-empty are closely related to: cons/rest/first/empty?.

One significant difference is that with a stack ADT, only the top item on the stack is accessible.

With a list data structure, any element is accessible (e.g., second).

One could implement a Stack ADT by using the Racket list data structure.
Queue ADT

A queue is like a “lineup”, where new items go to the “back” of the line, and the items are removed from the “front” of the line. While a stack is LIFO, a queue is FIFO (first in, first out).

Typical queue ADT operations:

- **add-back**: adds an item to the end of the queue
- **remove-front**: removes the item at the front of the queue
- **front**: returns the item at the front
- **is-empty**: determines if the queue is empty
Sequence ADT

The sequence ADT is useful when you want to be able to retrieve, insert or delete an item at any position in a sequence of items.

Typical sequence ADT operations:

- **item-at**: returns the item at a given position
- **insert-at**: inserts a new item at a given position
- **remove-at**: removes an item at a given position
- **length**: return the number of items in the sequence

The **insert-at** and **remove-at** operations change the position of items after the insertion/removal point.
Goals of this Section

At the end of this section, you should be able to:

• explain and demonstrate the three core advantages of modular design: abstraction, re-usability and maintainability

• identify two characteristics of a good modular interface: high cohesion and low coupling

• explain and demonstrate information hiding and how it supports both security and flexibility

• explain what a modular interface is, the difference between an interface and an implementation, and the importance of a good interface design.
• explain the differences between local, module and program scope and demonstrate how \textit{static} and \textit{extern} are used

• write modules in C with implementation and interface files

• implement an ADT module in C

• produce good interface documentation, including the new documentation changes introduced
Introduction to Imperative C

Readings: CP:AMA 2.4, 4.2–4.5, 5.2

- the ordering of topics is different in the text
- some portions of the above sections have not been covered yet
- some previously listed sections have now been covered in more detail
Functional vs. imperative programming

In CS 135 the focus is on functional programming, where functions behave very “mathematically”. The only purpose of a function is to produce a value, and the value depends only on the parameter(s).

The *functional programming paradigm* is to only use constant values that never change. Functions produce new values rather than changing existing ones.

A programming *paradigm* can also be though of as a programming “approach”, “philosophy” or “style”.
With functional programming, \((\text{add1 } n)\) produces a \textbf{new} value, but it does not actually \textit{change} \(n\). Once \(n\) is \textbf{defined}, it is a \textbf{constant} and always has the same value.

Similarly, \((\text{sort } \text{lon})\) produces a \textbf{new} list that is sorted, but the original list \(\text{lon}\) does not change.
In this course, the focus is on *imperative* programming and in this section we introduce imperative concepts.

In the English language, an imperative is an instruction or command: “Give me your money!”

Similarly, in imperative programming we use a sequence of statements (or “commands”) to give instructions to the computer.

To highlight the difference, we will consider an imperative special form within Racket (which will seem odd).

Many modern languages are “**multi-paradigm**”. Racket was primarily designed as a functional language but also supports imperative language features.
Side effects

In the *imperative* paradigm, an expression can also have a *side effect*.

An expression with a side effect does *more* than produce a value: it also changes the *state* of the program (or “the world”).

Functions (or programs) can also have side effects.

We have already seen a C function with a side effect: `printf`.

The side effect of `printf` is that it displays “output”. In other words, `printf` changes the *state* of the output.

```c
printf("Hello, World!\n");
```
The presence of side effects is a significant difference between imperative and functional programming.

In functional programming there are no side effects.

Some purists insist that a function with a side effect is no longer a “function” and call it a “procedure” (or a “routine”).

The “imperative programming paradigm” is also known as the “procedural programming paradigm”.

In this course we are more relaxed and a function can have side effects.
Documentation: side effects

Add an **effects**: section to a contract if there are any side effects.

```c
// purpose: simulates taking qty pills
// requires: qty > 0
// effects: displays a message

void take_headache_pills(int qty) {
    if (qty > 3) {
        printf("Nausea\n");
    }
    printf("Headache gone!\n");
}
```

For **interfaces**, you should only describe side effects in enough detail so that a client can use your module. Avoid disclosing any implementation-specific side effects.
In C, we used `void` to declare that a function returns “nothing”.

It is also used in `void` to declare that a function has no parameters.

```c
// purpose: displays a friendly message
// effects: displays a message

void say_hello(void) {
    printf("hello!\n");
    return; // this is optional
}
```

In a `void` function, `return` has no expression and it is optional.

As mentioned earlier, `main` is the only non-`void` function where the `return` is optional (`main` returns an `int`).
Expression statements

C’s `printf` is not a `void` function.

`printf` returns an `int` representing the number of characters printed.

`printf("hello!\n")` is an expression with the value of 7.

An expression statement is an expression followed by a semicolon (`;`).

`printf("hello!\n");`
In an expression statement, the value of the expression is ignored.

3 + 4;
7;
printf("hello!\n");

The three values of 7 in the above expression statements are never used.

The purpose of an expression statement is to produce a side effect.

Seashell may give you a warning if you have an expression statement without a side effect (e.g., 3 + 4;).
Block statements

A *block* ( `{ }` ) is also known as a **compound statement**, and contains a *sequence of statements*†.

† Blocks can also contain local scope *definitions*, which are not statements.

```c
{  
  const int x = 42; // scope is limited to enclosing braces  
  if (x == 10) {     
    printf("x is 10\n");  
  } else {          
    printf("x is not 10\n");  
  }
}
```
Earlier, we stated that in imperative programming we use a
sequence of statements.

We have seen two types of C statements:

• **compound statements** (a sequence of statements)

• **expression statements** (for producing side effects)

The only other type are **control flow statements**.
Control flow statements

As the name suggests, *control flow statements* change the “flow” of a program and the order in which other statements are executed.

We have already seen two examples:

- the *return* statement leaves a function to return a value.

- the *if* (and *else*) statements execute statements *conditionally*.

We will discuss control flow in more detail and introduce more examples later.


C terminology (so far)

```c
#include <stdio.h> // preprocessor directive

int add1(int x); // function declaration

int add1(int x) {
    // and block statement
    const int y = x + 1; // local definition
    printf("add1 called\n"); // expression statement
    // (with a side effect)
    2 + 2 == 5; // expression statement
    // (useless: no side effect)
    return y; // control flow statement
}
```
State

The biggest difference between the imperative and functional paradigms is the existence of *side effects*.

We described how a side effect changes the *state* of a program (or “the world”). For example, *printf* changes the state of the output.

The defining characteristic of the *imperative programming paradigm* is to *manipulate state*.

However, we have not yet discussed state.
**State** refers to the value of some data (or “information”) **at a** moment in time.

For an example of state, consider your bank account balance. At any moment in time, your bank account has a specific balance. In other words, your account is in a specific “*state*”. When you withdraw money from your account, the balance *changes* and the account is in a *new “state”*. State is related to **memory**, which is discussed in Section 05.
In a program, each variable is in a specific state (corresponding to its value).

In functional programming, each variable has only one possible state.

In imperative programming, each variable can be in one of many possible states.

The value of the variable can change during the execution of the program (hence, the name “variable”).
int main(void) {

    int n = 5;

    printf("the value of n is: %d\n", n);

    n = 6;

    printf("the value of n is: %d\n", n);
}

the value of n is: 5
the value of n is: 6

Note that \texttt{n} is \textbf{not} defined as a \texttt{const int}.
Mutation

When the value of a variable is changed it is known as *mutation*.

For most imperative programmers, mutation is second nature and not given a special name. They rarely use the term “*mutation*” (the word does not appear in the CP:AMA text).

Ironically, imperative programmers often use the oxymoronic terms “immutable variable” or “constant variable” instead of simply “constant”.
Mutable variables

The `const` keyword is explicitly required to define a constant. Without it, a variable is by default `mutable`.

```c
const int c = 42;  // constant
int m = 23;        // mutable variable
```

It is **good style** to use `const` when appropriate, as it:

- communicates the intended use of the variable,
- prevents ‘accidental’ or unintended mutation, and
- may allow the compiler to optimize (speed up) your code.
Assignment Operator

In C, mutation is achieved with the assignment operator (=).

```c
int m = 5; // initialization
m = 28;    // assignment operator
```

The assignment operator can be used on structs.

```c
struct posn p = {1,2};
struct posn q = {3,4};
p = q;
p.x = 23;
```
The = used in initialization is not the assignment operator.

Some initialization syntaxes are invalid with the assignment operator.

```c
struct posn p = {3,4}; // VALID INITIALIZATION
p = {5,7}; // INVALID ASSIGNMENT
p = {.x = 5}; // INVALID ASSIGNMENT
```

This is especially important when we introduce arrays and strings in Section 08.
The assignment operator is not symmetric.

\[ x = y; \]

is **not the same** as

\[ y = x; \]

Some languages use

\[ x := y \]

or

\[ x <- y \]

to make it clearer that it is an assignment and not symmetric.
Side effects

Clearly, an assignment operator has a side effect.

A function that mutates a global variable also has a side effect.

```c
int count = 0;

int increment(void) {
    count = count + 1;
    return count;
}

int main(void) {
    printf("%d\n", increment());
    printf("%d\n", increment());
    printf("%d\n", increment());
}
```

1
2
3
Even if a function does not have a side effect, its behaviour may depend on other mutable global variables.

```c
int n = 10;

int addn(int k) {
    return k + n;
}

int main(void) {
    printf("addn(5) = %d\n", addn(5));
    n = 100;
    printf("addn(5) = %d\n", addn(5));
}
```

```
addn(5) = 15
addn(5) = 105
```
In the functional programming paradigm, a function cannot have any side effects and the value it produces depends only on the parameter(s).

In the imperative programming paradigm, a function may have side effects and its behaviour may depend on the state of the program.

Testing functions in an imperative program can be challenging.
Racket supports mutation as well. The set! special form (pronounced “set bang”) “re-defines” the value of an existing identifier. set! can even change the type of an identifier.

(define n 5)
(n) ; => 5
(set! n "six")
(n) ; => "six"

The ! in set! is a Racket convention used to express “caution” that the functional paradigm is not being followed.
Racket structures can become mutable by adding the #:mutable option to the structure definition.

For each field of a mutable structure, a set-structname-fieldname! function is created.

```
(struct posn (x y) #:mutable #:transparent)

(define p (posn 3 4))

(set-posn-x! p 23)
(set-posn-y! p 42)
```
More assignment operators

In addition to the mutation side effect, the assignment operator (=) also produces the value of the expression on the right hand side.

This is occasionally used to perform multiple assignments.

\[ x = y = z = 0; \]

Avoid having more than one side effect per expression statement.

\[
\text{printf("y is %d\n", y = 5 + (x = 3)); // don’t do this!}
\]
\[
z = 1 + (z = z + 1); // or this!
\]
The value produced by the assignment operator is the reason using a single = instead of double == for equality is so dangerous!

```c
x = 0;
if (x = 1) {
    printf("disaster!\n");
}
```

$x = 1$ assigns 1 to $x$ and produces the value 1, so the if expression is always true, and it always prints disaster!

Pro Tip: some programmers get in the habit of writing $(1 == x)$ instead of $(x == 1)$. If they accidentally use a single = it causes an error.
The following statement forms are so common

\[
\begin{align*}
  x &= x + 1; \\
  y &= y + z;
\end{align*}
\]

that C has an addition assignment operator (\(+=\)) that combines the addition and assignment operator.

\[
\begin{align*}
  x &== 1; & \text{\// equivalent to } x &= x + 1; \\
  y &== z; & \text{\// equivalent to } y &= y + z;
\end{align*}
\]

There are also assignment operators for other operations.

\(-=, *=, /=, %=\).

As with the simple assignment operator, do not use these operators within larger expressions.
As if the simplification from \( x = x + 1 \) to \( x += 1 \) was not enough, there are also *increment* and *decrement* operators that increase and decrease the value of a variable by one.

\[
++x; \\
--x; \\
// or, alternatively \\
x++; \\
x--; \\
\]

It is best not to use these operators within a larger expression, and only use them in simple statements as above.

The difference between \( x++ \) and \( ++x \) and the relationship between their values and their side effects is tricky (see following slide).

The language C++ is a pun: one bigger (better) than C.
The *prefix* increment operator (\(++x\*) and the *postfix* increment operator (\(x++\)) both increment \(x\), they just have different *precedences* within the *order of operations*.

\(x++\) produces the “old” value of \(x\) and then increments \(x\).

\[++x\) increments \(x\) and then produces the “new” value of \(x\).

```c
x = 5;
j = x++;  // j = 5, x = 6
```

```c
x = 5
j = ++x;  // j = 6, x = 6
```

\;++x\) is preferred in most circumstances to improve clarity and efficiency.
Goals of this Section

At the end of this section, you should be able to:

- explain what a side effect is
- document a side effect in a contract with *effects* section
- use the new terminology introduced, including: expression statements, control flow statements, compound statements ({} )
- use the assignment operators
C Model: Memory & Control Flow

Readings: CP:AMA 6.1–6.4, 7.1–7.3, 7.6, Appendix E

Course Notes: Appendix A.5

- the ordering of topics is different in the text
- some portions of the above sections have not been covered yet
Models of computation

In CS 135, we modelled the computational behaviour of Racket with substitutions (the “stepping rules”).

To apply a function, all arguments are evaluated to values and then we substitute the body of the function, replacing the parameters with the argument values.

```
(define (my-sqr x) (* x x))
```

```
(+ 2 (my-sqr (+ 3 1)))
=> (+ 2 (my-sqr 4))
=> (+ 2 ( * 4 4))
=> (+ 2 16)
=> 18
```
In this course, we model the behaviour of C with

- memory and

- control flow.
Memory review

One bit of storage (in memory) has two possible states: 0 or 1.

A byte is 8 bits of storage. Each byte in memory is in one of 256 possible states.

Review Appendix A.5
Accessing memory

The smallest accessible unit of memory is a byte.

To access a byte of memory, you have to know its position in memory, which is known as the address of the byte.

For example, if you have 1 MB of memory (RAM), the address of the first byte is 0 and the address of the last byte is \(2^{20} - 1\).

**Note:** Memory addresses are usually represented in hex, so with 1 MB of memory, the address of the first byte is 0x0, and the address of the last byte is 0xFFFFFFFF.
If you can’t remember what $2^{20}$ is (especially on an exam) don’t panic.

You can figure it out by spending a little bit of time to write out:

\[ 2^0 = 1, \quad 2^1 = 2, \quad 2^2 = 4, \quad 2^3 = 8, \ldots, \quad 2^{10} = 1024. \]

You can write out more values if needed. To get to $2^{20}$

\[ 2^{20} = 2^{10} \times 2^{10} = 1024 \times 1024 = 1 \text{ MB} \]
You can visualize computer memory as a collection of “labeled mailboxes” where each mailbox stores a byte.

<table>
<thead>
<tr>
<th>address (1 MB of storage)</th>
<th>contents (one byte per address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000</td>
<td>00101001</td>
</tr>
<tr>
<td>0x000001</td>
<td>11001101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xFFFFFE</td>
<td>00010111</td>
</tr>
<tr>
<td>0xFFFFFF</td>
<td>01110011</td>
</tr>
</tbody>
</table>

The \textit{contents} in the above table are arbitrary values.
Defining variables

When C encounters a variable definition, it

- reserves (or “finds”) space in memory to store the variable

- “keeps track of” the address of that storage location

- stores the initial value of the variable at that location (address).

For example, with the definition

```c
int n = 0;
```

C reserves space (an address) to store `n`, “keeps track of” the address `n`, and stores the value `0` at that address.

A variable definition reserves space but a declaration does not.
In our CS 135 substitution model, a variable is a “name for a value”. When a variable appears in an expression, a substitution occurs and the name is replaced by its value.

In our new model, a variable is a “name for a location” where a value is stored.

When a variable appears in an expression, C “fetches” the contents at its address to obtain the value stored there.
sizeof

When we define a variable, C reserves space in memory to store its value – but **how much space** is required?

It depends on the type of the variable.

It may also depend on the environment (the machine and compiler).
The **size operator** (sizeof), produces the number of bytes required to store a type (it can also be used on identifiers). `sizeof` looks like a function, but it is an operator.

```c
int n = 0;
printf("the size of an int is: %zd\n", sizeof(int));
printf("the size of n is: %zd\n", sizeof(n));
```

```
the size of an int is: 4
the size of n is: 4
```

In this course, the size of an **integer** is 4 bytes (32 bits).

The placeholder for a size is "\%zd" (the type is `size_t`).
In C, the size of an `int` depends on the machine (processor) and/or the operating system that it is running on.

Every processor has a natural “word size” (e.g., 32-bit, 64-bit). Historically, the size of an `int` was the word size, but most modern systems use a 32-bit `int` to improve compatibility.

In C99, the `inttypes` module (#include `<inttypes.h>`) defines many types (e.g., `int32_t`, `int16_t`) that specify exactly how many bits (bytes) to use.

In this course, you should only use `int`, and there are always 32 bits in an `int`. 
**example: variable definition**

```c
int n = 0;
```

For this variable definition C reserves (or “finds”) 4 consecutive bytes of memory to store `n` (*e.g.*, addresses `0x5000`...`0x5003`) and then “keeps track of” the first (or “starting”) address.

<table>
<thead>
<tr>
<th>identifier</th>
<th>type</th>
<th># bytes</th>
<th>starting address</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>int</td>
<td>4</td>
<td>0x5000</td>
</tr>
</tbody>
</table>

C updates the contents of the 4 bytes to store the initial value (0).

<table>
<thead>
<tr>
<th>address</th>
<th>0x5000</th>
<th>0x5001</th>
<th>0x5002</th>
<th>0x5003</th>
</tr>
</thead>
<tbody>
<tr>
<td>contents</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>
**Integer limits**

Because C uses 4 bytes (32 bits) to store an `int`, there are only $2^{32}$ (4,294,967,296) possible values that can be represented.

The range of C `int` values is $-2^{31} \ldots (2^{31} - 1)$ or $-2,147,483,648 \ldots 2,147,483,647$.

If you `#include <limits.h>`, the constants `INT_MIN` and `INT_MAX` are defined with those limit values.

```
unsigned int variables represent the values 0 \ldots (2^{32} - 1)
but we do not use them in this course.
```
Overflow

If we try to represent values outside of the integer limits, **overflow** occurs.

For example, when you add one to 2,147,483,647 the result is 
\[-2,147,483,648.\]

By carefully specifying the order of operations, you can sometimes avoid overflow.

You are not responsible for calculating overflow, but you should understand why it occurs and how to avoid it.

In CS 251 / CS 230 you will learn more about overflow.
example: overflow

```c
int bil = 1000000000;
int four_bil = bil + bil + bil + bil;
int nine_bil = 9 * bil;

printf("the value of 1 billion is: %d\n", bil);
printf("the value of 4 billion is: %d\n", four_bil);
printf("the value of 9 billion is: %d\n", nine_bil);

the value of 1 billion is: 1000000000
the value of 4 billion is: -294967296
the value of 9 billion is: 410065408
```
Racket can handle arbitrarily large numbers, such as 
(expt 2 1000).

Why did we not have to worry about overflow in Racket?

Racket does not use a fixed number of bytes to store numbers. Racket represents numbers with a *structure* that can use an arbitrary number of bytes (imagine a *list* of bytes).

There are C modules available that provide similar features (a popular one is available at gmplib.org).
The char type

The char type is also used to store integers, but C only allocates **one byte** of storage for a char (an int uses 4 bytes).

There are only $2^8$ (256) possible values for a char and the range of values is $(-128 \ldots 127)$ in our Seashell environment.

Because of this limited range, chars are rarely used for calculations. As the name implies, they are often used to store **characters**.
ASCII

Early in computing, there was a need to represent text (characters) in memory.

The American Standard Code for Information Interchange (ASCII) was developed to assign a numeric code to each character.

Upper case A is 65, while lower case a is 97. A space is 32.

ASCII was developed when teletype machines were popular, so the characters 0 . . . 31 are teletype “control characters” (e.g., 7 is a “bell” noise).

The only control character we use in this course is the line feed (10), which is the newline \n character.
/*
   32 space 48 0 64 @ 80 P 96 ‘ 112 p
   33 ! 49 1 65 A 81 Q 97 a 113 q
   34 " 50 2 66 B 82 R 98 b 114 r
   35 # 51 3 67 C 83 S 99 c 115 s
   36 $ 52 4 68 D 84 T 100 d 116 t
   37 % 53 5 69 E 85 U 101 e 117 u
   38 & 54 6 70 F 86 V 102 f 118 v
   39 ' 55 7 71 G 87 W 103 g 119 w
   40 ( 56 8 72 H 88 X 104 h 120 x
   41 ) 57 9 73 I 89 Y 105 i 121 y
   42 * 58 : 74 J 90 Z 106 j 122 z
   43 + 59 ; 75 K 91 [ 107 k 123 {
   44 , 60 < 76 L 92 \ 108 l 124 |
   45 - 61 = 77 M 93 ] 109 m 125 }
   46 . 62 > 78 N 94 ^ 110 n 126 ~
   47 / 63 ? 79 O 95 _ 111 o
*/
ASCII worked well in English-speaking countries in the early days of computing, but in today’s international and multicultural environments it is outdated.

The **Unicode** character set supports more than 100,000 characters from all over the world.

A popular method of **encoding** Unicode is the UTF-8 standard, where displayable ASCII codes use only one byte, but non-ASCII Unicode characters use more bytes.
C characters

In C, **single** quotes (’’) are used to indicate an ASCII character.

For example, ’a’ is equivalent to 97 and ’z’ is 122. C “translates” ’a’ into 97.

In C, there is **no difference** between the following two variables:

```c
char letter_a = 'a';
char ninety_seven = 97;
```

Always use **single** quotes with characters:

"a" is **not** the same as ’a’.
The `printf` placeholder to display a *character* is "\%c".

```c
char letter_a = 'a';
char ninety_seven = 97;

printf("letter_a as a character: \%c\n", letter_a);
printf("ninety_seven as a char: \%c\n", ninety_seven);

printf("letter_a in decimal: \%d\n", letter_a);
printf("ninety_seven in decimal: \%d\n", ninety_seven);

letter_a as a character: a
ninety_seven as a char: a

letter_a in decimal: 97
ninety_seven in decimal: 97
```
Character arithmetic

Because C interprets characters as integers, characters can be used in expressions to avoid having “magic numbers” in your code.

```c
bool is_lowercase(char c) {
    return (c >= 'a') && (c <= 'z');
}

// to_lowercase(c) converts upper case letters to lowercase letters, everything else is unchanged
char to_lowercase(char c) {
    if ((c >= 'A') && (c <= 'Z')) {
        return c - 'A' + 'a';
    } else {
        return c;
    }
}
```
Structures in the memory model

For a structure *definition* no memory is reserved:

```c
struct posn {
    int x;
    int y;
};
```

Memory is only reserved when a `struct` variable is defined.

```c
struct posn p = {3,4};
```
The amount of space reserved for a `struct` is **at least** the sum of the `sizeof` each field, but it may be larger.

```c
struct mystruct {
    int x; // 4 bytes
    char c; // 1 byte
    int y; // 4 bytes
};

printf("sizeof(struct mystruct) = %zd\n", sizeof(struct mystruct));

sizeof(struct mystruct) = 12
```

*You must use the `sizeof` operator to determine the size of a structure.*
The size may depend on the *order* of the fields:

```c
struct s1 {
    char c;
    int i;
    char d;
};

struct s2 {
    char c;
    char d;
    int i;
};
```

```c
printf("The sizeof s1 is: %zd\n", sizeof(struct s1));
printf("The sizeof s2 is: %zd\n", sizeof(struct s2));
```

The sizeof s1 is: 12
The sizeof s2 is: 8

C may reserve more space for a structure to improve *efficiency* and enforce *alignment* within the structure.
Floats

A **double** has more precision than a **float** because it uses more memory.

Just as we might represent a number in decimal as $6.022 \times 10^{23}$, a **float** uses a similar strategy.

A 32 bit **float** uses 24 bits for the **mantissa** and 8 bits for the **exponent**.

A 64 bit **double** uses $(53 + 11)$.

**Floats** and their internal representation are discussed in CS 251 / 230 and in detail in CS 370 / 371.
Sections of memory

In this course we model five *sections* (or “regions”) of memory:

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Only Data</td>
</tr>
<tr>
<td>Global Data</td>
</tr>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>Stack</td>
</tr>
</tbody>
</table>

Other courses may use alternative names.

The *heap* section is introduced in Section 10.
Sections are combined into memory segments, which are recognized by the hardware (processor).

When you try to access memory outside of a segment, a segmentation fault occurs (more on this in CS 350).
When evaluating C expressions, the intermediate results must be temporarily stored.

\[ a = f(3) + g(4) - 5; \]

In the above expression, C must temporarily store the value returned from \( f(3) \) “somewhere” before calling \( g \).

In this course, we do not discuss this “temporary” storage, which is covered in CS 241.
The code section

When you program, you write *source code* in a text editor using ASCII characters that are “human readable”.

To “run” a C program the *source code* must first be converted into *machine code* that is “machine readable”.

This machine code is then placed into the *code section* of memory where it can be executed.

Converting source code into machine code is known as *compiling*. It is briefly discussed in Section 13 and covered extensively in CS 241.
The read-only & global data sections

Earlier we described how C “reserves space” in memory for a variable definition. For example:

```c
int n = 0;
```

The location of memory depends on whether the variable is global or local.

First, we discuss global variables. We discuss local variables and the stack section after discussing control flow.

All global variables are placed in either the read-only data section (constants) or the global data section (mutable variables).
Global variables are available throughout the entire execution of the program, and the space for the global variables is reserved before the program begins execution.

First, the code from the entire program (all of the modules) is scanned and all global variables are identified. Next, space for each global variable is reserved. Finally, the memory is properly initialized. This happens before the main function is called.

The read-only and global memory sections are created and initialized at compile time.
Control flow

In our C model, we use **control flow** to model how programs are executed.

During execution, we keep track of the **program location**, which is “where” in the code the execution is currently occurring.

When a program is “run”, the **program location** starts at the beginning of the **main** function.

In hardware, the **location** is known as the **program counter**, which contains the **address** within the machine code of the current instruction (more on this in CS 241).
int g(int x) {
    return x + 1;
}

int f(int x) {
    return 2 * x + g(x);
}

int main(void) {
    int a = f(2);
    //...
}

When a function is called, the program location “jumps” to the start of the function. The return keyword “returns” the location back to the calling function.
The return address

For each function call, we need to “remember” the program location to “jump back to” when we `return`. In other words, when a return statement is reached what address should we “return to, to resume execution in the calling function”. This location is known as the \textit{return address}.

In this course, we use the name of the function and a line number (or an arrow) to represent the return address.
The call stack

Suppose the function `main` calls `f`, then `f` calls `g`, and `g` calls `h`.

As the program flow jumps from function to function, we need to “remember” the “history” of the return addresses. When we return from `h`, we jump back to the return address in `g`. The “last called” is the “first returned”.

This “history” is known as the **call stack**. Each time a function is called, a new entry is *pushed* onto the stack. Whenever a `return` occurs, the entry is *popped* off of the stack.
**Stack frames**

The “entries” pushed onto the *call stack* are known as *stack frames*.

Each function call creates a *stack frame* (or a “frame of reference”).

Each *stack frame* contains:

- the *argument values*

- any *local variables* that appear within the function *block* (including any sub-blocks), and

- the *return address*. 
As with Racket, **before** a function can be called, all of the arguments must be values.

C makes a copy of each argument value and **places the copy in the stack frame**.

This is known as the “pass by value” convention.
Whereas space for a *global* variable is reserved *before* the program begins execution, space for a *local* variable is only reserved *when the function is called*.

The space is reserved within the newly created stack frame.

When the function *returns*, the variable (and the entire frame) is popped and effectively “disappears”.

In C, local variables are known as *automatic* variables because they are “automatically” created when needed. There is an *auto* keyword in C but it is rarely used.
example: stack frames

```c
int h(int i) {
    int r = 10 * i;
    return r;
}

int g(int y) {
    int c = y * y;
    return c;
}

int f(int x) {
    int b = 2*x + 1;
    return g(b + 3) + h(b);
}

int main(void) {
    int a = f(2);
    //...
}
```

---

g:
  y: 8
  c: 64
  return address: f:13

---

f:
  x: 2
  b: 5
  return address: main:17

---

main:
  a: ???
  return address: OS

---
char func(int i, char c) {
    if (i > 10) {
        return c;
    } else if (i < 0) {
        return c + 2;
    } else {
        return c + 5;
    }
}

int main() {
    char k = func(7, 'a');
}

The return address is the location in the program where the function was called. There will be one and only one return address on each stack frame. NOTE: the return address is NOT the location of the return statement with a function.
A **void** function does not require a return statement.

```c
void func(int i) {
    printf("i = %d\n", i);
}

int main() {
    func(7);
    func(9);
}
```

In practice, the *return address* is the address of the machine instruction following the function call.
Recursion in C

Now that we understand how stack frames are used, we can see how *recursion* works in C.

In C, each recursive call is simply a new *stack frame* with a separate frame of reference.

The only unusual aspect of recursion is that the *return address* is a location within the same function.

In this example, we will also see control flow with the *if* statement.
```c
int sum_first(int n) {
    if (n == 0) {
        return 0;
    } else {
        return n + sum_first(n-1);
    }
}

int main(void) {
    int a = sum_first(2);
    //...
}
```
Stack section

The call stack is stored in the stack section, the fourth section of our memory model. We refer to this section as “the stack”.

In practice, the “bottom” of the stack (i.e., where the main stack frame is placed) is placed at the highest available memory address. Each additional stack frame is then placed at increasingly lower addresses. The stack “grows” toward lower addresses.

If the stack grows too large, it can “collide” with other sections of memory. This is called “stack overflow” and can occur with very deep (or infinite) recursion.
Uninitialized memory

In most situations, mutable variables *should* be initialized, but C will allow you to define variables without any initialization.

```c
int i;
```

For all **global** variables, C will automatically initialize the variable to be zero.

Regardless, it is good style to explicitly initialize a global variable to be zero, even if it is automatically initialized.

```c
int g = 0;
```
A local variable (on the stack) that is uninitialized has an arbitrary initial value.

```c
define mystery(void) {
    int k;
    printf("the value of k is: %d\n", k);
}
```

Seashell gives you a warning if you obtain the value of an uninitialized variable.

In the example above, the value of k will likely be a leftover value from a previous stack frame.
## Memory sections (so far)

<table>
<thead>
<tr>
<th>low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
</tr>
<tr>
<td>Read-Only Data</td>
</tr>
<tr>
<td>Global Data</td>
</tr>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>↑</td>
</tr>
<tr>
<td>Stack</td>
</tr>
</tbody>
</table>

| high                    |

CS 136 Fall 2017 05: C Model
Model

We now have the tools to model the behaviour of a C program.

At any moment of execution, a program is in a specific state, which is the combination of:

- the current program location, and
- the current contents of the memory.

To properly interpret a program’s behaviour, we must keep track of the program location and all of the memory contents.

For the remainder of this Section we will discuss the control flow mechanisms in C.
Calling a function

Calling a function is control flow. When a function is called:

• a *stack frame* is created (“pushed” onto the Stack memory area)

• a *copy* of each of the arguments is placed in the stack frame

• the current program location is placed in the stack frame as the *return address*

• the program location is changed to the start of the new function

• the initial values of local variables are set when their definition is encountered
We have already seen the `return` control flow statement.

When a function **returns**: 

- the current program location is changed back to the `return address` (which is retrieved from the stack frame)
- the stack frame is removed (“popped” from the Stack memory area)

The return **value** (for non-**void** functions) is stored in a temporary memory area we are not discussing in this course. This will be discussed further in CS 241.
if statement

We briefly introduced the if control flow statement in Section 03. We now discuss if in more detail.

The syntax of if is

```
if (expression) statement
```

where the statement is only executed if the expression is true (non-zero).

```
if (n < 0) printf("n is less than zero\n");
```

Remember: the if statement does not produce a value. It only controls the flow of execution.
The `if` statement only affects whether the *next* statement is executed. To conditionally execute more than one statement, braces (`{}`) are used to insert a compound statement block (a sequence of statements) in place of a single statement.

```c
if (n <= 0) {
    printf("n is zero\n");
    printf("or less than zero\n");
}
```

Using braces is **strongly recommended** even if there is only one statement. It makes the code easier to follow and less error prone. (*In the notes, we omit them only to save space.*)

```c
if (n <= 0) {
    printf("n is less than or equal to zero\n");
}
```
Statement A;
if (exp) {
 Code Block;
}
Statement Z;
As we have seen, the if statement can be combined with else statement(s) for multiple conditions.

```c
if (expression) {
    statement(s)
} else if (expression) {
    statement(s)
} else if (expression) {
    statement(s)
} else {
    statement(s)
}
```
Statement A;
if (exp1) {
    Code Block 1;
} else if (exp2) {
    Code Block 2;
} else {
    Code Block 3;
}
Statement Z;
If an `if` condition **returns**, there may be no need for an `else`.

```c
int sum(int k) {
    if (k <= 0) {
        return 0;
    } else {
        return k + sum(k - 1);
    }
}

// Alternate equivalent code
int sum(int k) {
    if (k <= 0) {  // Added curly braces for clarity
        return 0;
    } else {  // Added curly braces for clarity
        return k + sum(k - 1);
    }
}
```
Braces are sometimes necessary to avoid a “dangling” else.

```c
if (y > 0)
    if (y != 5)
        printf("you lose");
else
    printf("you win!"); // when does this print?
```
The C `switch` control flow statement (see CP:AMA 5.3) has a similar structure to `else if` and `cond`, but very different behaviour.

A `switch` statement has “fall-through” behaviour where more than one branch can be executed.

In our experience, `switch` is very error-prone for beginner programmers.

Do not use `switch` in this course.
The C `goto` control flow statement (CP:AMA 6.4) is one of the most disparaged language features in the history of computer science because it can make “spaghetti code” that is hard to understand.

Modern opinions have tempered and most agree it is useful and appropriate in some circumstances.

To use `gotos`, you must also have `labels` (code locations).

```c
if (k < 0) goto mylabel;
  //...
mylabel:
  //...
```

Do not use `goto` in this course.
Looping

With mutation, we can control flow with a method known as \textit{looping}.

\begin{verbatim}
while (expression) statement
\end{verbatim}

\textbf{while} is similar to \textbf{if}: the \texttt{statement} is only executed \texttt{if} the \texttt{expression} is true.

The difference is, \textbf{while} \textbf{repeatedly} "loops back" and executes the \texttt{statement} \textbf{until the expression is false}.

Like with \texttt{if}, you should always use braces (\{\}) for a \textit{compound statement}, even if there is only a single statement.
Statement A;
while (exp) {
    Code Block;
}
Statement Z;
example: while loop

<table>
<thead>
<tr>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ \Rightarrow \text{int } i = 2; \]
\[ \text{while } (i >= 0) \{ \]
\[ \quad \text{printf("%d\n", i);} \]
\[ \quad --i; \]
\[ \}
\]

OUTPUT:
Iteration vs. recursion

Using a loop to solve a problem is called iteration.

Iteration is an alternative to recursion and is much more common in imperative programming.

// recursion
int sum(int k) {
    if (k <= 0) {
        return 0;
    }
    return k + sum(k - 1);
}

// iteration
int sum(int k) {
    int s = 0;
    while (k > 0) {
        s += k;
        --k;
    }
    return s;
}
When first learning to write loops, you may find that your code is very similar to using *accumulative recursion*.

```c
int accsum(int k, int acc) {
    if (k == 0) return acc;
    return accsum(k - 1, k + acc);
}

int recursive_sum(int k) {
    return accsum(k, 0);
}

int iterative_sum(int k) {
    int acc = 0;
    while (k > 0) {
        acc += k;
        k--;
    }
    return acc;
}
```

Looping is very “imperative”. Without mutation (side effects), the while loop condition would not change, causing an “endless loop”.

Loops can be “nested” within each other.

```c
int i = 5;
while (i >= 0) {
    int j = i;
    while (j >= 0) {
        printf(" * ");
        --j;
    }
    printf("\n");
    --i;
}
```

```
******
*****
****
***
**
*
```
Changing parameter values

Earlier, we saw this example of an iterative function:

```c
int sum(int k) {
    int s = 0;
    while (k > 0) {
        s += k;
        --k;
    }
    return s;
}
```

In this code, we mutate `k` within the loop, which may seem odd because `k` is a parameter.

Remember that a copy of each argument is passed to the function, so the function `sum` is free to mutate its own copy of `k`. 
while errors

A simple mistake with `while` can cause an “endless loop” or “infinite loop”. Each of the following examples will produce an endless loop.

```c
while (i >= 0) // missing {}
    printf("%d\n", i);
    --i;
```

```c
while (i >= 0); { // extra ;
    printf("%d\n", i);
    --i;
}
```

```c
while (i = 100) { ... } // assignment typo
```

```c
while (1) { ... } // constant true expression
```
Do while

The do control flow statement is very similar to while.

    do statement while (expression);

The difference is that statement is always executed at least once, and the expression is checked at the end of the loop.

    int i = 0;
    bool success; // an uninitialized var (rare!)
    do {
        ++i;
        success = guess_pin(i);
    } while (!success);
Statement A;

Code Block;

\[ \text{exp?} \]

\text{true}

\text{false}

Statement Z;

do {
  Code Block;
} while (exp);

Statement Z;
The **break** control flow statement is useful when you want to exit from the *middle* of a loop.

**break** immediately terminates the current (innermost) loop.

**break** is often used with a (purposefully) infinite loop.

```c
while (1) {
    // stuff
    if (early_exit_condition) break;
    // more stuff
}
```
The `continue` control flow statement skips over the rest of the statements in the current block ( `{}` ) and “continues” with the loop.

```c
// only concerned with fun numbers
int i = 0;
while (i <= 9999) {
    ++i;
    if (!is_fun(i)) continue;
    //...
}
```
Statement A;
while (exp) {
    Code Block;
}
Statement Z;
for loops

The final control flow statement we introduce is `for`, which is often referred to as a “for loop”.

`for` loops are a “condensed” version of a `while` loop. The format of a `while` loop is often of the form:

```
setup statement
while (expression) {
    body statement(s)
    update statement
}
```

which can be re-written as a single `for` loop:

```
for (setup; expression; update) { body statement(s) }
```
for vs. while

Recall the for syntax.

```c
for (setup; expression; update) { body statement(s) }
```

This while example
```c
i = 100; // setup
while (i >= 0) { // expression
    printf("%d\n", i);
    --i; // update
}
```

is equivalent to
```c
for (i = 100; i >= 0; --i) {
    printf("%d\n", i);
}
```
Statement A;
for (setup; exp; update) {
    Code Block;
}
Statement Z;
Statement A;
for (setup; exp; update) {
    Code Block;
}
Statement Z;

exp?
true
Code Block
false
Most for loops follow one of these forms (or “idioms”).

// Counting up from 0 to n-1
for (i = 0; i < n; ++i) {...}

// Counting up from 1 to n
for (i = 1; i <= n; ++i) {...}

// Counting down from n-1 to 0
for (i = n-1; i >= 0; --i) {...}

// Counting down from n to 1
for (i = n; i > 0; --i) {...}

It is a common mistake to be “off by one” (e.g., using < instead of <=). Sometimes re-writing as a while is helpful.
In C99, the setup statement can be a definition.

This is very convenient for defining a variable that only has local (block) scope within the for loop.

```c
for (int i = 100; i >= 0; --i) {
    printf("%d\n", i);
}
```

The equivalent while loop would have an extra block.

```c
{
    int i = 100;
    while (i >= 0) {
        printf("%d\n", i);
        --i;
    }
}
```
You can omit any of the three components of a `for` statement. If the expression is omitted, it is always “true”.

```
for (; i < 100; ++i) {...} // i was setup previously
for (; i < 100;) {...} // same as a while(i < 100)
for (;;) {...} // endless loop
```

You can use the `comma operator` (,) to use more than one expression in the `setup` and `update` statements of a `for` loop. See CP:AMA 6.3 for more details.

```
for (i = 1, j = 100; i < j; ++i, --j) {...}
```
A for loop is not always equivalent to a while loop.

The only difference is when a continue statement is used.

In a while loop, continue jumps back to the expression.

In a for loop, the “update” statement is executed before jumping back to the expression.
Goals of this Section

At the end of this section, you should be able to:

- explain why C has limits on integers and why overflow occurs
- use the `char` type and explain how characters are represented in ASCII
- explain how C execution is modelled with memory and control flow, as opposed to the substitution model of Racket
• describe the 4 areas of memory seen so far: code, read-only data, global data and the stack

• identify which section of memory an identifier belongs to

• explain a stack frame and its components (return address, parameters, local variables)

• explain how C makes copies of arguments for the stack frame
• use the introduced control flow statements, including (return, if, while, do, for, break, continue)

• re-write a recursive function with iteration and vice versa

• trace the execution of small programs by hand, and draw the stack frames at specific execution points
Introduction to Pointers in C

Readings: CP:AMA 11, 17.7
Address operator

C was designed to give programmers “low-level” access to memory and expose the underlying memory model.

The **address operator** (&) produces the starting address of where the value of an identifier is stored in memory.

```c
int g = 42;

int main(void) {
    printf("the value of g is:  %d\n",  g);
    printf("the address of g is:  %p\n",  &g);
}
```

the value of g is:  42
the address of g is:  0x68a9e0

The `printf` placeholder to display an address (in hex) is "%p".
Pointers

In C, there is also a type for storing an address: a pointer.

A pointer is defined by placing a star (*) before the identifier (name). The * is part of the declaration syntax, not the identifier itself.

```c
int i = 42;
int *p = &i; // p "points at" i
```

The type of `p` is an “int pointer” which is written as “int *”.

For each type (e.g., `int`, `char`) there is a corresponding pointer type (e.g., `int *`, `char *`).
This definition:

```c
int *p = &i; // p "points at" i
```

is equivalent to the following definition and assignment:

```c
int *p; // p is defined (not initialized)
p = &i; // p now "points at" i
```

The `*` is part of the definition/declaration of `p` and is not part of the variable name. It is part of the type specification. The name of the variable is simply `p`, not `*p`.

As with any variable, its value can be changed.

```c
p = &i; // p now "points at" i
//...
p = &j; // p now "points at" j
```
The **value** of a pointer is an **address**.

```c
int i = 42;
int * p = &i;

printf("value of i (i) = %d\n", i);
printf("address of i (&i) = %p\n", &i);
printf("value of p (p) = %p\n", p);

value of i (i) = 42
address of i (&i) = 0xf020
value of p (p) = 0xf020
```

To make working with pointers easier in these notes, we often use shorter, simplified (“fake”) addresses.
```
int i = 42;
int *p = &i;
```

<table>
<thead>
<tr>
<th>identifier</th>
<th>type</th>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>int</td>
<td>0xf020</td>
<td>42</td>
</tr>
<tr>
<td>p</td>
<td>int *</td>
<td>0xf024</td>
<td>0xf020</td>
</tr>
</tbody>
</table>

When drawing a *memory diagram*, we rarely care about the value of the address, and visualize a pointer with an arrow (that “points”).
sizeof a pointer

In most $k$-bit systems, memory addresses are $k$ bits long, so pointers require $k$ bits to store an address.

In our 64-bit Seashell environment, the sizeof a pointer is always 64 bits (8 bytes).

The sizeof a pointer is always the same size, regardless of the type of data stored at that address.

$$
\text{sizeof(int *) } \Rightarrow 8 \\
\text{sizeof(char *) } \Rightarrow 8 
$$
Indirection operator

The *indirection operator* (*), also known as the dereference operator, is the inverse of the address operator (&).

*p* produces the value of what pointer p “points at”.

```c
int i = 42;
int *p = &i;    // pointer p points at i

printf("value of p (p) = %p\n", p);
printf("value of what p points at (*p) = %d\n", *p);
```

value of p (p) = 0xf020
value of what p points at (*p) = 42

The value of *&i* is simply the value of i.
The **address operator** (\&) can be thought of as: 
“*get the address of this box*”.

The **indirection operator** (*) can be thought of as: 
“*follow the arrow to the next box*”.

\[ *p \Rightarrow 42 \]
The * symbol is used in three different ways in C:

- as the *multiplication operator* between expressions
  
  $k = i \times i$;

- in pointer *declarations* and pointer *types*
  
  `int *p = &i;`
  
  `sizeof(int *)`

- as the *indirection operator* for pointers
  
  $j = *p$;
  
  `$p = 5$;

$(p \times p)$ is a confusing but valid C expression.
Note that the pointer’s identifier (the variable name) can be any valid identifier.

```c
int i = 2;
int j = 3;
int *pi = &i; // pi is a pointer to i
int *pj = &j; // pj is a pointer to j
int *ptr = &i; // ptr also points at i
int *pk = &i; // pk is a pointer to i
int *jenny = &j; // jenny is a pointer to j

pi = &j; // the variable name does not restrict its use
pj = &i;
```
C mostly ignores white space, so these are equivalent

```c
int *pi = &i; // style A
int * pi = &i; // style B
int* pi = &i; // style C
```

There is some debate over which is the best style. Proponents of style B & C argue it’s clearer that the type of `p` is an “`int *`”.

However, *in the declaration* the `*` “belongs” to the `p`, not the `int`, and so style A is used in this course and in CP:AMA.

This is clear with multiple declarations: (not encouraged)

```c
int i = 42, j = 23;
int *pi = &i, *pj = &j; // VALID
int * pi = &i, pj = &j; // INVALID: pj is not a pointer
```
Pointers to pointers

A common question is: "Can a pointer point at itself?"

```c
int * p = &p; // pointer p points at p ???
```

This is actually a type error:

- p is declared as `(int *)`, a pointer to an `int`, but
- the type of `&p` is `(int **)`, a pointer to a pointer to an `int`. 
In C, we can declare a **pointer to a pointer**:

```c
int i = 42;
int * pi = &i;  // pointer pi points at i
int ** ppi = &pi;  // pointer ppi points at pi
```

C allows any number of pointers to pointers. More than two levels of “pointing” is uncommon.

\[(**ppi \times **ppi)\] is a confusing but valid C expression.

A **void** pointer (\texttt{void \*}) can point at anything, including a **void** pointer (itself).
The NULL pointer

NULL is a special pointer value to represent that the pointer points to “nothing”, or is “invalid”. Some functions return a NULL pointer to indicate an error. NULL is essentially “zero”, but it is good practice to use NULL in code to improve communication.

If you dereference a NULL pointer, your program will likely crash.

Most functions should require that pointer parameters are not NULL.

assert (p != NULL);
assert (p); // <-- because NULL is not true...
    // this is equivalent and common

NULL is defined in the stdlib module (and several others).
Function pointers

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

For example, Racket functions are values that can be stored in variables and data structures, passed as arguments and returned by functions.

In C, functions are not first-class values, but *function pointers* are.

A significant difference is that **new** Racket functions can be created during program execution, while in C they cannot.

A function pointer can only point to a function that already exists.
A *function pointer* stores the starting address of a function.

A function pointer **declaration** includes the *return type* and all of the *parameter types*, which makes them a little messy.

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

int main(void) {
    int (*fp)(int) = add1; // OR = &add1;
    printf("add1(3) = %d\n", fp(3));
}

add1(3) = 4
```

The syntax to declare a function pointer with name `fpname` is:

```
return_type (*fpname)(param1_type, param2_type, ...)
```
examples: function pointer declarations

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

char functionB(int i, int j) {...}  
char (*fpB)(int, int) = functionB;

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

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

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

In an exam, we would not expect you to remember the syntax for declaring a function pointer.
Pointer assignment

Consider the following code

```c
int i = 5;
int j = 6;

int *p = &i;
int *q = &j;

p = q;
```

The statement `p = q;` is a `pointer assignment`. It means “change `p` to point at what `q` points at”. It changes the `value` of `p` to be the `value` of `q`. In this example, it assigns the `address` of `j` to `p`.

It does not change the value of `i`. 
p = q;
Using the same initial values,

```c
int i = 5;
int j = 6;
int * p = &i;
int * q = &j;
```

the statement

```c
*p = *q;
```

does not change the value of \( p \): it changes the value of what \( p \) points at. In this example, it changes the value of \( i \) to 6, even though \( i \) was not used in the statement.

This is an example of aliasing, which is when the same memory address can be accessed from more than one variable.
*p = *q;
example: aliasing

```c
int i = 2;

int *p1 = &i;
int *p2 = p1;

printf("i = %d\n", i);
*p1 = 7; // i changes...
printf("i = %d\n", i);
*p2 = 100; // without being used directly
printf("i = %d\n", i);
i = 2
i = 7
i = 100
```
Mutation & parameters

Consider the following C program:

```c
void inc(int i) {
    ++i;
}

int main(void) {
    int x = 5;
    inc(x);
    printf("x = %d\n", x); // 5 or 6 ?
}
```

It is important to remember that when `inc(x)` is called, a `copy` of `x` is placed in the stack frame, so `inc` cannot change `x`.

The `inc` function is free to change it's own copy of the argument (in the stack frame) without changing the original variable.
void inc(int i) {
    ++i;
}

int main(void) {
    int x = 5;
    inc(x);
    printf("x = %d\n", x);
}
In the “pass by value” convention of C, a copy of an argument is passed to a function.

The alternative convention is “pass by reference”, where a variable passed to a function can be changed by the function. Some languages support both conventions.

What if we want a C function to change a variable passed to it? (this would be a side effect)

In C we can emulate “pass by reference” by passing the address of the variable we want the function to change. This is still considered “pass by value” because we pass the value of the address.
By passing the *address* of \( x \), we can change the *value* of \( x \).

It is also common to say “pass a pointer to \( x \)”.

```c
void inc(int * p) {
    * p += 1;
}

int main(void) {
    int x = 5;
    inc(&x); // note the &
    printf("x = %d\n", x); // NOW it’s 6
}
```

\( x = 6 \)

To pass the address of \( x \) use the **address operator** (\&\( x \)).

The corresponding parameter type is an **int pointer** (\texttt{int *}).
void inc(int *p) {
    *p += 1;
}

int main(void) {
    int x = 5;
    inc(&x);
    inc(&x);
    printf("x = %d\n", x);
}
void inc(int * p) {
    *p += 1;
}

Note that instead of *p += 1; we could have written (*p)++;
The parentheses are necessary.

Because of the order of operations, the ++ would have
incremented the pointer p, not what it points at (*p).

C is a minefield of these kinds of issues: the best strategy is to
use straightforward code.
example: mutation side effects

// effects: swaps the contents of *px and *py
void swap(int *px, int *py) {
    int temp = *px;
    *px = *py;
    *py = temp;
}

int main(void) {
    int a = 3;
    int b = 4;
    printf("a = %d, b = %d\n", a, b);
    swap(&a, &b); // Note the &
    printf("a = %d, b = %d\n", a, b);
}

a = 3, b = 4
a = 4, b = 3
In the *functional paradigm*, there is no observable difference between “pass by value” and “pass by reference”.

In Racket, simple values (e.g., numbers) are passed by *value*, but structures are passed by *reference*.

Mutable structures can be modified by a function.

```racket
(struct mposn (x y) #:mutable #:transparent)

(define (swap! mp)
  (define oldx (mposn-x mp))
  (set-mposn-x! mp (mposn-y mp))
  (set-mposn-y! mp oldx))

(define my-posn (mposn 3 4))
(swap! my-posn)
my-posn ;; => (mposn 4 3)
```
Returning more than one value

Like Racket, C functions can only return a single value.

Pointer parameters can be used to *emulate* “returning” more than one value.

The addresses of several variables can be passed to the function, and the function can change the value of the variables.
example: “returning” more than one value

This function performs division and “returns” both the quotient and the remainder.

```c
void divide(int num, int denom, int *quot, int *rem) {
    *quot = num / denom;
    *rem = num % denom;
}

int main(void) {
    int q; // this is a rare example where
    int r; // no initialization is necessary

    divide(13, 5, &q, &r);

    assert(q == 2 && r == 3);
}
```
This “multiple return” technique is useful when it is possible that a function could encounter an error.

For example, the previous `divide` example could return `false` if it is successful and `true` if there is an error (i.e., division by zero).

```c
bool divide(int num, int denom, int *quot, int *rem) {
    if (denom == 0) return true;
    *quot = num / denom;
    *rem = num % denom;
    return false;
}
```

Some C library functions use this approach to return an error. Other functions use “invalid” sentinel values such as `-1` or `NULL` to indicate when an error has occurred.
The return type of a function can also be an address (pointer).

```c
int *ptr_to_max(int *a, int *b) {
    if (*a >= *b) return a;
    return b;
}

int main(void) {
    int x = 3;
    int y = 4;

    int *p = ptr_to_max(&x, &y); // note the &
    assert(p == &y);
}
```

Returning addresses become more useful in Section 10.
A function must **never** return an address within its stack frame.

```c
int *bad_idea(int n) {
    return &n; // NEVER do this
}

int *bad_idea2(int n) {
    int a = n*n;
    return &a; // NEVER do this
}
```

As soon as the function *returns*, the stack frame “disappears”, and all memory within the frame should be considered **invalid**.
Passing structures

Recall that when a function is called, a \textbf{copy} of each argument value is placed into the stack frame.

For structures, the \textit{entire} structure is copied into the frame. For large structures, this can be inefficient.

```
struct bigstruct {
    int a; int b; int c; ... int y; int z;
};
```

Large structures also increase the size of the stack frame. This can be \textit{especially} problematic with recursive functions, and may even cause a \textit{stack overflow} to occur.
To avoid structure copying, it is common to pass the *address* of a structure to a function.

```c
int sqr_dist(struct posn *p1, struct posn *p2) {
    int xdist = (*p1).x - (*p2).x;
    int ydist = (*p1).y - (*p2).y;
    return xdist * xdist + ydist * ydist;
}
```

```c
int main(void) {
    struct posn p1 = {2,4};
    struct posn p2 = {5,8};
    assert(sqr_dist(&p1, &p2) == 25); // note the &
}
```
The parentheses ( ) in the expression (*p1).x are used because the structure operator (.) has higher precedence than the indirection operator (*).

Without the parentheses, *p1.x is equivalent to *(p1.x) which is a “type” syntax error because p1 does not have a field x.

Writing the expression (*ptr).field is a awkward. Because it frequently occurs there is an additional selection operator for working with pointers to structures.
The **arrow selection operator** (`->`) combines the indirection and the selection operators.  

```
ptr->field is equivalent to (*ptr).field
```

The arrow selection operator can only be used with a **pointer to a structure**.

```c
int sqr_dist(struct posn *p1, struct posn *p2) {
    int xdist = p1->x - p2->x;
    int ydist = p1->y - p2->y;
    return xdist * xdist + ydist * ydist;
}
```
Passing the address of a structure to a function (instead of a copy) also allows the function to mutate the fields of the structure.

// scale(p, f) scales the posn *p by f
// requires: p is not null
// effects: changes the field values of p

void scale(struct posn *p, int f) {
    p->x *= f;
    p->y *= f;
}

If a function has a pointer parameter, the documentation should clearly communicate whether or not the function can mutate the pointer’s destination (“what the pointer points at”).

While all side effects should be properly documented, documenting the absence of a side effect may be awkward.
const pointers

Adding the `const` keyword to a pointer definition prevents the pointer’s destination from being mutated through the pointer.

```c
void cannot_change(const struct posn *p) {
    p->x = 5;  // INVALID
}
```

The `const` should be placed first, before the type (see the next slide).

It is **good style** to add `const` to a pointer parameter to communicate (and enforce) that the pointer’s destination does not change.
The syntax for working with pointers and `const` is tricky.

```c
int * p; // p can change, can point at any int

const int * p; // p can change, but must point at a const int

int * const p = &i; // p must always point at i, but i can change

const int * const p = &i; // p is constant and i is constant
```

The rule is “`const` applies to the type to the left of it, unless it’s first, and then it applies to the type to the right of it”.

Note: the following are equivalent and a matter of style.

```c
const int i = 42;
int const i = 42;
```
const parameters

As we just established, it is good style to use const with pointer parameters to communicate that the function will not (and can not) mutate the contents of the pointer.

```c
void can_change(struct posn *p) {
    p->x = 5;  // VALID
}

void cannot_change(const struct posn *p) {
    p->x = 5;  // INVALID
}
```

What does it mean when const is used with simple (non-pointer) parameters?
For a simple value, the `const` keyword indicates that the parameter is immutable \textit{within the function.}

```c
int my_function(const int x) {
  // mutation of x here is invalid
  // ...
}
```

It does not require that the argument passed to the function is a constant.

Because a \texttt{copy} of the argument is made for the stack, it does not matter if the original argument value is constant or not.

A `const` parameter communicates that \texttt{the copy} will not be mutated.
For simple parameters, \texttt{const} is meaningless in a function declaration.

The caller (client) does not need to know if the function will mutate the \texttt{copy} of the argument value.

\begin{verbatim}
int my_function(int x); // DECLARATION

// (no const)

int my_function(const int x) { // DEFINITION
  // mutation of x here is invalid // (with const)
  // ...
}
\end{verbatim}

It is good style to use \texttt{constant} parameters in \texttt{definitions} to improve communication.

\begin{quote}
In the notes, we often omit \texttt{const} in parameters to save space.
\end{quote}
Opaque structures in C

C supports **opaque structures** through *incomplete declarations*, where a structure is *declared* without any fields. With incomplete declarations, only *pointers* to the structure can be defined.

```c
struct posn; // INCOMPLETE DECLARATION

struct posn my_posn; // INVALID
struct posn *posn_ptr; // VALID
```

If a module only provides an *incomplete declaration* in the *interface*, the client can not directly access any of the fields.

The module must provide a function to *create* an instance of the structure. This will be explored more in Section 10.
Goals of this Section

At the end of this section, you should be able to:

- declare and de-reference pointers
- use the new operators (&, *, ->)
- explain when a pointer parameter should be `const`
- use function pointers
- describe aliasing
- use pointers to structures as parameters and explain why parameters are often pointers to structures
I/O & Testing

Readings: CP:AMA 2.5

Course Notes: Appendix A.6
I/O

Input & Output (I/O for short) is the term used to describe how programs interact with the “real world”.

A program may interact with a human by receiving data from an input device (like a keyboard, mouse or touch screen) and sending data to an output device (like a screen or printer).

A program can also interact with non-human entities, such as a file in secondary storage (e.g., a hard drive) or even a different computer (e.g., a website).
Output

We have already seen the printf function (in both Racket and C) that prints formatted output via placeholders.

In C, we have seen the placeholders \%d(ecimal integer), \%c(haracter), \%f(loat) and \%p(ointer / address).

In Racket, we have seen \~a(ny). The \~v(alue) placeholder is useful when debugging as it shows extra type information (such as the quote for a 'symbol).

In this course, we only output "text", and so printf is the only output function we need.
Writing to **text files** directly is almost as straightforward as using `printf`. The `fprintf` function (`file printf`) has an additional parameter that is a file pointer (`FILE *`). The `fopen` function opens (creates) a file and return a pointer to that file.

```c
#include <stdio.h>

int main(void) {
    FILE *file_ptr;
    file_ptr = fopen("hello.txt", "w"); // w for write
    fprintf(file_ptr, "Hello World!\n");
    fclose(file_ptr);
}
```

See CP:AMA 22.2 for more details.
Debugging output

Output can be very useful to help **debug** our programs.

We can use **printf** to output intermediate results and ensure that the program is behaving as expected. This is known as **tracing** a program. **Tracing** is especially useful when there is mutation.

A global variable can be used to turn tracing on or off.

```c
const bool TRACE = true; // set to false to turn off tracing
if (TRACE) printf("The value of i is: %d\n",i);
```

In practice, tracing is commonly implemented with **macros** (**#define**) that can be turned on & off (CP:AMA 14).
C input: `scanf`

In C, the `scanf` function is the counterpart to the `printf` function.

```c
scanf("%d", &i); // read in an integer, store in i
```

`scanf` requires a **pointer** to a variable to store the value read in from input.

Just as with `printf`, you use multiple placeholders to read in more than one value.

However, in this course **only read in one value per `scanf`**.

This will help you debug your code and facilitate our testing.
The return value of `scanf` is the number (count) of values successfully read.

The return value can also be the special constant value `EOF` to indicate that the End Of File (EOF) has been reached.

In Seashell, when you run (not test), a Ctrl-D (“Control D”) keyboard sequence sends an EOF.

In this course, a return value of one is “success”.

```c
count = scanf("%d", &i); // read in an int, store in i
if (count != 1) {
    printf("Fail! I could not read in an integer!\n");
}
```
\texttt{scanf("\%d", \&i)} will \textbf{ignore whitespace} (spaces and newlines) and read in the next integer.

If the next non-whitespace input to be read is not a valid integer (e.g., a letter), it will stop reading and return zero.

When reading in a \texttt{char}, you may or may not want to ignore whitespace.

\begin{verbatim}
  // reads in next character (may be whitespace character)
  count = scanf("\%c", \&c);

  // reads in next character, ignoring whitespace
  count = scanf(" \%c", \&c);
\end{verbatim}

The extra leading space in the second example indicates that whitespace should be ignored.
```c
int main(void) {
    int num = 0;
    int i = 0;
    int sum = 0;

    printf("how many numbers should I sum?\n");
    if (scanf("%d", &num) != 1) {
        printf("bad input!\n");
        return 1;
    }

    for (int j=0; j < num; ++j) {
        printf("enter #%d:\n", j+1);
        if (scanf("%d", &i) != 1) {
            printf("bad input!\n");
            return 1;
        }
        sum += i;
    }

    printf("the sum of the %d numbers is: %d\n", num, sum);
}
```
```c
int main(void) {
    int num = 0;
    int i = 0;
    int sum = 0;

    printf("keep entering numbers, press Ctrl-D when done.\n");
    while (1) {
        printf("enter #%d: \n", num + 1);
        if (scanf("%d", &i) != 1) {
            break;
        }
        sum += i;
        ++num;
    }
    printf("the sum of the %d numbers is: %d\n", num, sum);
}
```
Tips for testing in C

Here are some additional tips for testing in C:

- check for “off by one” errors in loops
- consider the case that the initial loop condition is not met
- make sure every control flow path is tested
- consider large argument values (\texttt{INT\_MAX} or \texttt{INT\_MIN})
- test for special argument values (\texttt{-1}, \texttt{0}, \texttt{1}, \texttt{NULL})
Goals of this Section

At the end of this section, you should be able to:

• use the I/O terminology introduced

• use the input function `scanf` in C to make interactive programs

• use the Seashell testing environment effectively
Arrays & Strings

Arrays

C only has two built-in types of “compound” data storage:

- structures
- arrays

```c
int my_array[6] = {4, 8, 15, 16, 23, 42};
```

An array is a data structure that contains a fixed number of elements that all have the same type.

Because arrays are built-in to C, they are used for many tasks where lists are used in Racket, but arrays and lists are very different. In Section 11 we construct Racket-like lists in C.
int my_array[6] = {4, 8, 15, 16, 23, 42};

To define an array we must know the length of the array in advance (we address this limitation in Section 10).

Each individual value in the array is known as an element. To access an element, its index is required.

The first element of my_array is at index 0, and it is written as my_array[0].

The second element is my_array[1] and the last is my_array[5].

In computer science we often start counting at 0.
example: accessing array elements

Each individual array element can be used in an expression as if it was a variable.

```c
int a[6] = {4, 8, 15, 16, 23, 42};

int j = a[0]; // j is 4
int *p = &a[j-1]; // p points at a[3]

a[2] = a[a[0]]; // a[2] is now 23
++a[1]; // a[1] is now 9
```
example: arrays & iteration

Arrays and iteration are a powerful combination.

```c
int a[6] = {4, 8, 15, 16, 23, 42};
int sum = 0;

for (int i = 0; i < 6; ++i) {
    printf("a[%d] = %d\n", i, a[i]);
    sum += a[i];
}
printf("sum = %d\n", sum);
```

a[0] = 4
a[1] = 8
a[2] = 15
a[3] = 16
a[4] = 23
a[5] = 42
sum = 108
Array initialization

Arrays can only be initialized with braces ({}).

```c
int a[6] = {4, 8, 15, 16, 23, 42};
```

```c
a = {0, 0, 0, 0, 0, 0}; // INVALID
a = ??? ; // INVALID
```

Once defined, the entire array cannot be assigned to at once.

Each *individual element* must be mutated.

```c
for (int i=0; i < 6; ++i) {
    a[i] = 0;
}
```
Like variables, the value of an uninitialized array depends on the scope of the array:

```c
int a[5];
```

- uninitialized *global* arrays are zero-filled.

- uninitialized *local* arrays are filled with arbitrary ("garbage") values from the stack.

If there are not enough elements in the braces, the remaining values are initialized to zero (even with local arrays).

```c
int c[5] = {0};      // c[0]...c[4] = 0
```
If an array is initialized, the length of the array can be omitted from the declaration and automatically determined from the number of elements in the initialization.

```c
int a[] = {4, 8, 15, 16, 23, 42}; // int a[6] = ... 
```

This syntax is only allowed if the array is initialized.

```c
int b[]; // INVALID
```

Similar to structures, C99 supports a partial initialization syntax.

```c
```

Omitted elements are initialized to zero.
C99 allows the length of an **uninitialized local array** to be determined *while the program is running*. The size of the stack frame is increased accordingly.

```c
int count;
printf("How many numbers? ");
scanf("%d", &count);

int a[count]; // count determined at run-time
```

This approach has many disadvantages and in the most recent version of C (C11), this feature was made optional. In Section 10 we see a better approach.
Array size

The **length** of an array is the number of elements in the array.

The **size** of an array is the number of bytes it occupies in memory.

An array of $k$ elements, each of size $s$, requires exactly $k \times s$ bytes.

In the C memory model, array elements are adjacent to each other. Each element of an array is placed in memory immediately after the previous element.

If $a$ is an integer array with six elements (int $a[6]$) the size of $a$ is:

$$ (6 \times \text{sizeof(int)}) = 6 \times 4 = 24. $$

Not everyone uses the same terminology for length and size.
**example: array in memory**

```c
int a[6] = {4, 8, 15, 16, 23, 42};
printf("&a[0] = %p ... &a[5] = %p\n", &a[0], &a[5]);
&a[0] = 0x5000 ... &a[5] = 0x5014
```

<table>
<thead>
<tr>
<th>addresses</th>
<th>contents (4 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5000 ... 0x5003</td>
<td>4</td>
</tr>
<tr>
<td>0x5004 ... 0x5007</td>
<td>8</td>
</tr>
<tr>
<td>0x5008 ... 0x500B</td>
<td>15</td>
</tr>
<tr>
<td>0x500C ... 0x500F</td>
<td>16</td>
</tr>
<tr>
<td>0x5010 ... 0x5013</td>
<td>23</td>
</tr>
<tr>
<td>0x5014 ... 0x5017</td>
<td>42</td>
</tr>
</tbody>
</table>
Array length

C does not explicitly keep track of the array length as part of the array data structure.

You must keep track of the array length separately.

Typically, the array length is stored in a separate variable.

```c
const int a_length = 6;
int a[6] = {4, 8, 15, 16, 23, 42};
```
const int a_length = 6;
int a[a_length];

The above definition is fine in Seashell, but some C environments do not allow the length of the array to be specified by a variable.

In those environments, the `#define` syntax is more often used. This is common in CP:AMA.

```
#define A_LENGTH 6
int a[A_LENGTH];
```
Theoretically, in some circumstances you could use `sizeof` to determine the length of an array.

```c
int len = sizeof(a) / sizeof(a[0]);
```

The CP:AMA textbook uses this on occasion.

However, in practice, this should be avoided, as the `sizeof` operator only properly reports the array size in very specific circumstances.
The array identifier

The value of an array (a) is the same as the address of the array (&a), which is also the address of the first element (&a[0]).

```c
int a[] = {4, 8, 15, 16, 23, 42};
printf("%p %p %p\n", a, &a, &a[0]);
printf("%d %d\n", a[0], *a);
```

```
0x5000 0x5000 0x5000
4 4
```

Dereferencing the array (*a) is equivalent to referencing the first element (a[0]).
Passing arrays to functions

When an array is passed to a function only the **address** of the array is copied into the stack frame. This is more efficient than copying the entire array to the stack.

Typically, the length of the array is unknown, and is provided as a separate parameter.
example: array parameters

```c
int sum_array(int a[], int len) {
    int sum = 0;
    for (int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}

int main(void) {
    int my_array[6] = {4, 8, 15, 16, 23, 42};
    int sum = sum_array(my_array, 6);
}
```

Note the parameter syntax: `int a[]`
and the calling syntax: `sum_array(my_array, 6)`.
As we have seen before, passing an address to a function allows the function to change (mutate) the contents at that address.

```c
void array_add1(int a[], int len) {
    for (int i = 0; i < len; ++i) {
        ++a[i];
    }
}
```

It’s good style to use the `const` keyword to both prevent mutation and communicate that no mutation occurs.

```c
int sum_array(const int a[], int len) {
    int sum = 0;
    for (int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```
Because a structure can contain an array:

```c
struct mystruct {
    int big[1000];
};
```

It is especially important to pass a pointer to such a structure, otherwise, the entire array is copied to the stack frame.

```c
int slower(struct mystruct s) {
    ...
}
```

```c
int faster(struct mystruct * s) {
    ...
}
```
**Pointer arithmetic**

We have not yet discussed any *pointer arithmetic*.

C allows an integer to be added to a pointer, but the result may not be what you expect.

If \( p \) is a pointer, the value of \( (p+1) \) **depends on the type** of the pointer \( p \).

\( (p+1) \) adds the `sizeof` whatever \( p \) points at.

According to the official C standard, pointer arithmetic is only valid **within an array** (or a structure) context. This becomes clearer later.
Pointer arithmetic rules

- When adding an integer \( i \) to a pointer \( p \), the address computed by \((p + i)\) in C is given in “normal” arithmetic by:
  \[
  p + i \times \text{sizeof}(\ast p).
  \]

- Subtracting an integer from a pointer \((p - i)\) works in the same way.

- Mutable pointers can be incremented (or decremented).
  \( ++p \) is equivalent to \( p = p + 1 \).
• You cannot add two pointers.

• You can subtract a pointer \( q \) from another pointer \( p \) if the pointers are the same type (point to the same type). The value of \( (p - q) \) in C is given in “normal” arithmetic by:

\[
(p - q)/\text{sizeof}(\ast p).
\]

In other words, if \( p = q + i \) then \( i = p - q \).

• Pointers (of the same type) can be compared with the comparison operators: <, <=, ==, !=, >=, >

\((\text{e.g., if } (p < q) \ldots)\).
Pointer arithmetic and arrays

Pointer arithmetic is useful when working with arrays.

Recall that for an array `a`, the value of `a` is the address of the first element (`&a[0]`).

Using pointer arithmetic, the address of the second element `&a[1]` is `(a + 1)`, and it can be referenced as `*(a + 1)`.

The array indexing syntax ([ ]) is an operator that performs pointer arithmetic.

`a[i]` is equivalent to `*(a + i)`. 
In *array pointer notation*, square brackets ([ ]) are not used, and all array elements are accessed through pointer arithmetic.

```c
int sum_array(const int *a, int len) {
    int sum = 0;
    for (const int *p = a; p < a + len; ++p) {
        sum += *p;
    }
    return sum;
}
```

Note that the above code behaves **identically** to the previously defined `sum_array`:

```c
int sum_array(const int a[], int len) {
    int sum = 0;
    for (int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```
another example: pointer notation

// count_match(item, a, len) counts the number of occurrences of item in the array a

int count_match(int item, const int *a, int len) {
    int count = 0;
    const int *p = a;
    while (p < a + len) {
        if (*p == item) {
            ++count;
        }
        ++p;
    }
    return count;
}
The choice of notation (pointers or \[\]) is a matter of style and context. You are expected to be comfortable with both.

C makes no distinction between the following two function declarations:

\[
\begin{align*}
\text{int array\_function(int a[], int len) \{\ldots\} // a[]} \\
\text{int array\_function(int \ast a, int len) \{\ldots\} // \ast a}
\end{align*}
\]

In most contexts, there is no practical difference between an array identifier and an immutable pointer.

The subtle differences between an array and a pointer are discussed at the end of this Section.
example: “pretty” print an array
// pretty prints an array with commas, ending with a period
// requires: len > 0
void print_array(int a[], int len) {
assert(len > 0);
for (int i=0; i < len; ++i) {
if (i) {
printf(", ");
}
printf("%d", a[i]);
}
printf(".\n");
}
int main(void) {
int a[6] = {4, 8, 15, 16, 23, 42};
print_array(a, 6);
}
4, 8, 15, 16, 23, 42.
CS 136 Fall 2017

08: Arrays & Strings

27


Array map

Aside from the awkward function pointer parameter syntax, the implementation of `array_map` is straightforward.

// effects: replaces each element a[i] with f(a[i])

```c
void array_map(int (*f)(int), int a[], int len) {
    for (int i=0; i < len; ++i) {
        a[i] = f(a[i]);
    }
}
```
#include "array_map.h"

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

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

int main(void) {
    int a[] = {4, 8, 15, 16, 23, 42};
    print_array(a, 6);
    array_map(add1, a, 6);
    print_array(a, 6);
    array_map(sqr, a, 6);
    print_array(a, 6);
}

4, 8, 15, 16, 23, 42.
5, 9, 16, 17, 24, 43.
25, 81, 256, 289, 576, 1849.
Selection sort

In *selection sort*, the smallest element is *selected* to be the first element in the new sorted sequence, and then the next smallest element is selected to be the second element, and so on.
First, we find the position of the smallest element...

\[ \begin{array}{cccccc}
8 & 6 & 7 & 5 & 3 & 0 & 9 \\
\end{array} \]

and then we *swap* the first element with the smallest.

\[ \begin{array}{cccccc}
0 & 6 & 7 & 5 & 3 & 8 & 9 \\
\end{array} \]

Then, we find the next smallest element...

\[ \begin{array}{cccccc}
0 & 6 & 7 & 5 & 3 & 8 & 9 \\
\end{array} \]

and then we *swap* that element with the second one, and so forth...

\[ \begin{array}{cccccc}
0 & 3 & 7 & 5 & 6 & 8 & 9 \\
\end{array} \]
```c
void selection_sort(int a[], int len) {
    for (int i=0; i < len - 1; ++i) {
        int pos = i;
        for (int j = i + 1; j < len; ++j) {
            if (a[j] < a[pos]) {
                pos = j;
            }
        }
        swap(&a[i], &a[pos]);  // see Section 05
    }
}
```

// Notes:
// i: loops from 0 ... len-2 and represents the
//    "next" element to be replaced
// j: loops from i+1 ... len-1 and is "searching"
//    for the next smallest element
// pos: position of the "next smallest"
Quicksort

Quicksort is an example of a “divide & conquer“ algorithm.

First, an element is selected as a “pivot” element.

The list is then partitioned (divided) into two sub-groups: elements less than (or equal to) the pivot and those greater than the pivot.

Finally, each sub-group is then sorted (conquered).

Quicksort is also known as partition-exchange sort or Hoare’s quicksort (named after the author).
We have already seen the implementation of quick sort in racket.

```
(define (quick-sort lon)
  (cond [(empty? lon) empty]
    [else (define pivot (first lon))
      (define less (filter (lambda (x)
          (<= x pivot)) (rest lon)))
      (define greater (filter (lambda (x)
          (> x pivot)) (rest lon)))
      (append (quick-sort less)
        (list pivot)
        (quick-sort greater))])
```

For simplicity, we select the first element as the “pivot”. A more in-depth discussion of pivot selection occurs in CS 240.
In our C implementation of quick sort, we:

- select the first element of the array as our “pivot”
- move all elements that are larger than the pivot to the back of the array
- move (“swap”) the pivot into the correct position
- recursively sort the “smaller than” sub-array and the “larger than” sub-array

The core quick sort function `quick_sort_range` has parameters for the range of elements (`first` and `last`) to be sorted, so a wrapper function is required.
void quick_sort_range(int a[], int first, int last) {

    if (last <= first) return;  // length is <= 1

    int pivot = a[first];      // first element is the pivot
    int pos = last;            // where to put next larger

    for (int i = last; i > first; --i) {
        if (a[i] > pivot) {
            swap(&a[pos], &a[i]);
            --pos;
        }
    }

    swap(&a[first], &a[pos]);   // put pivot in correct place
    quick_sort_range(a, first, pos-1);
    quick_sort_range(a, pos+1, last);
}

void quick_sort(int a[], int len) {
    quick_sort_range(a, 0, len-1);
}
Binary search

In Racket, the built-in function `member` can be used to determine if a list contains an element.

We can write a similar function in C that finds the index of an element in an array:

```c
// find(item, a, len) finds the index of item in a, // or returns -1 if it does not exist

int find(int item, const int a[], int len) {
    for (int i=0; i < len; ++i) {
        if (a[i] == item) {
            return i;
        }
    }
    return -1;
}
```
But what if the array was previously sorted?

We can use **binary search** to find the element faster:

```c
int find_sorted(int item, const int a[], int len) {
    int low = 0;
    int high = len-1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (a[mid] == item) {
            return mid;
        } else if (a[mid] < item) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}
```
Multi-dimensional data

All of the arrays seen so far have been one-dimensional (1D) arrays.

We can represent multi-dimensional data by “mapping” the higher dimensions down to one.

For example, consider a 2D array with 2 rows and 3 columns.

\[
\begin{array}{ccc}
1 & 2 & 3 \\
7 & 8 & 9 \\
\end{array}
\]

We can represent the data in a simple one-dimensional array.

```c
int data[6] = {1, 2, 3, 7, 8, 9};
```

To access the entry in row \( r \) and column \( c \), we simply access the element at \( \text{data}[r \times 3 + c] \).

In general, it would be \( \text{data}[\text{row} \times \text{NUMCOLS} + \text{col}] \).
C supports multiple-dimension arrays, but they are not covered in this course.

```c
int two_d_array[2][3];
int three_d_array[10][10][10];
```

When multi-dimensional arrays passed as parameters, the second (and higher) dimensions must be fixed. (e.g., `int function_2d(int a[][10], int numrows)`).

Internally, C represents a multi-dimensional array as a 1D array and performs “mapping” similar to the method described in the previous slide.

See CP:AMA sections 8.2 & 12.4 for more details.
Fixed-Length Arrays

A significant limitation of an array is that you need to know the length of the array in advance.

In Section 10 we introduce dynamic memory which can be used to circumvent this limitation, but first we explore a less sophisticated approach.

In some applications, it may be “appropriate” (or “easier”) to have a maximum length for an array.

In general, maximums should only be used when appropriate:

- They are wasteful if the maximum is excessively large.
- They are restrictive if the maximum is too small.
When working with maximum-length arrays, we need to keep track of

- the **“actual” length** of the array, and

- the **maximum possible length**.
To illustrate fixed-length arrays, we will implement an integer stack structure with a maximum length of 100 elements.

The len field will keep track of the actual length of the stack.

```c
struct stack {
    int len;
    int maxlen;
    int data[100];
};
```

We will need to provide a stack_init function to initialize the structure:

```c
void stack_init(struct stack *s) {
    assert(s);
    s->len = 0;
    s->maxlen = 100;
}
```
Ignoring the **push** operation for now, we can write the rest of the stack implementation:

```c
bool stack_is_empty(const struct stack *s) {
  assert(s);
  return s->len == 0;
}

int stack_top(const struct stack *s) {
  assert(s);
  assert(s->len > 0);
  return s->data[s->len - 1];
}

// note: stack_pop returns the element popped
int stack_pop(struct stack *s) {
  assert(s);
  assert(s->len > 0);
  s->len -= 1;
  return s->data[s->len];
}
```
What happens if we exceed the maximum length when we try to push an element?

There are a few possibilities:

- the stack is not modified and an error message is displayed
- a special return value can be used
- an assertion fails (terminating the program)
- the program explicitly terminates with an error message

Any approach may be appropriate as long as the contract properly documents the behaviour.
The `exit` function (part of `<stdlib.h>`) stops program execution. It is useful for “fatal” errors.

The argument passed to `exit` is equivalent to the `return` value of `main`.

For convenience, `<stdlib.h>` defines `EXIT_SUCCESS` (0) and `EXIT_FAILURE` (non-zero).

```c
if (something_bad) {
    printf("FATAL ERROR: Something bad happened!\n");
    exit(EXIT_FAILURE);
}
```
// stack_push(item, s) pushes item onto stack s
// requires: s is a valid stack
// effects: modifies s
// if max stack size is exceeded,
// prints a message and exits

void stack_push(int item, struct stack *s) {
    assert(s);
    if (s->len == s->maxlen) {
        printf("FATAL ERROR: max stack size (%d) exceeded\n", s->maxlen);
        exit(EXIT_FAILURE);
    }
    s->data[s->len] = item;
    s->len += 1;
}
Strings

There is no built-in C `string` type. The “convention” is that a C string is an array of characters, terminated by a null character.

```c
char my_string[4] = {'c', 'a', 't', '\0'};
```

The null character, also known as a null terminator, is a char with a value of zero. It is often written as `\0` instead of just 0 to improve communication and indicate that a null character is intended.

`\0` is equivalent to 0. That is different from '0', which is equivalent to 48 (the ASCII character for the symbol zero).
String initialization

char arrays also support a double quote (" ) initialization syntax. When combined with the automatic length declaration ([ ]), the length includes the null terminator.

The following definitions create equivalent 4-character arrays:

```c
char a[] = { 'c', 'a', 't', '\0' };
char b[] = { 'c', 'a', 't', 0 };
char c[4] = { 'c', 'a', 't' };
char d[] = { 99, 97, 116, 0 };
char e[4] = "cat";
char f[] = "cat";
```

This array initialization notation is different than the double quote notation used in expressions (e.g., in printf("string")).
Null termination

With null terminated strings, we do not need to pass the length to functions. It is determined by the location of the ‘\0’.

```c
// e_count(s) counts the # of e’s and E’s in string s

int e_count(const char s[]) {
    int count = 0;
    int i = 0;
    while (s[i]) { // not the null terminator
        if ((s[i] == 'e') || (s[i] == 'E')) {
            ++count;
        }
        ++i;
    }
    return count;
}
```

It is good style to have `const` parameters to communicate that no changes (mutation) occurs to the string.
The string library (#include <string.h>) provides many useful functions for processing strings (more on this library later).

The strlen function returns the length of the string, not necessarily the length of the array. It does not include the null character.

```c
int my_strlen(const char s[]) {
    int len = 0;
    while (s[len]) {
        ++len;
    }
    return len;
}
```
Here is an alternative implementation of `my_strlen` that uses pointer arithmetic.

```c
int my_strlen(const char * s) {
    const char * p = s;
    while (*p) {
        ++p;
    }
    return (p-s);
}
```

In practice, pointer notation is often used with strings as it is slightly faster. Using array index notation (`s[i]`) performs an extra addition in the loop.
Lexicographical order

Characters can be easily compared \((c_1 < c_2)\) as they are numbers, so the character order is determined by the ASCII table.

If we try to compare two strings \((s_1 < s_2)\), C compares their addresses (pointers), which is not helpful.

To compare strings we are typically interested in using a lexicographical order.

Strings require us to be more careful with our terminology, as “smaller than” and “greater than” are ambiguous: are we considering just the length of the string? To avoid this problem we use precedes (“before”) and follows (“after”).
To compare two strings using a lexicographical order, we first compare the first character of each string. If they are different, the string with the smaller first character precedes the other string. Otherwise (the first characters are the same), the second characters are compared, and so on.

If the end of one string is encountered, it precedes the other string. Two strings are equal (the same) if they are the same length and all of their characters are identical.

The following strings are in lexicographical order:

"" "a" "az" "c" "cab" "cabin" "cat" "catastrophe"
The `<string.h>` library function `strcmp` uses lexicographical ordering.

`strcmp(s1, s2)` returns zero if the strings are identical. If `s1` precedes `s2`, it returns a negative integer. Otherwise (`s1` follows `s2`) it returns a positive integer.

```c
int my_strcmp(const char s1[], const char s2[]) {
    int i = 0;
    while (s1[i] == s2[i]) {
        if ((s1[i] == '\0') && (s2[i] == '\0')) return 0;
        ++i;
    }
    if (s1[i] < s2[i]) return -1;
    return 1;
}
```
To compare if two strings are equal (identical), use the `strcmp` function.

The equality operator (`==`) only compares the addresses of the strings, and not the contents of the arrays.

```c
char a[] = "the same?";
char b[] = "the same?";
char *s = a;

if (a == b) ...          // False (diff. addresses)
if (strcmp(a, b) == 0) ...  // True (proper comparison)
if (a == s) ...          // True (same addresses)
```
Lexicographical orders can be used to compare (and sort) any sequence of elements (arrays, lists, ...) and not just strings.

The following Racket function lexicographically compares two lists of numbers:

\[
(\text{define} \ (\text{lon<=?} \ \text{lon1} \ \text{lon2})
\begin{cases}
\text{((empty?} \ \text{lon1}) \ #t) \\
\text{((empty?} \ \text{lon2}) \ #f) \\
\text{[(< (first} \ \text{lon1}) \ (first} \ \text{lon2)})) \ #t] \\
\text{[(< (first} \ \text{lon2}) \ (first} \ \text{lon1})) \ #f] \\
\text{else (lon<=? (rest} \ \text{lon1}) \ (rest} \ \text{lon2)))})
\end{cases}
\]

\[
(\text{lon<=?} \ '(4 \ 9 \ 1 \ 2 \ 1) \ '(4 \ 5 \ 9)) \ ; \Rightarrow \ #f
\]

\[
(\text{lon<=?} \ '(4 \ 3) \ '(4 \ 3 \ 2)) \ ; \Rightarrow \ #t
\]
String I/O

The `printf` placeholder for strings is `%s`.

```c
char a[] = "cat";
printf("the %s in the hat\n", a);
```

`printf` prints out characters until the null character is encountered.

`printf` does not print out the null character.
When using %s with `scanf`, it stops reading the string when a “white space” character is encountered (e.g., a space or \n).

`scanf("%s", ...)` is useful for reading in one “word” at a time.

```c
char name[81];
printf("What is your first name?\n");
scanf("%s", name);
```

You must be very careful to reserve enough space for the string to be read in, and **do not forget the null character**.

`scanf("%s", ...)` will automatically add the null character.
example: understanding scanf

```c
char name[10] = {0};
while (scanf("%s", name) == 1) {
    printf("Hello, %s!\n", name);
}
```

The input:

```
Samantha Bob [EOF]
```

Generates the following output:

```
Hello, Samantha!
Hello, Bob!
```

Afterwards, what is stored in the `name` array?

```
Bob\ntthaha\n```
In the following example, the `name` array is 81 characters and can accommodate first names with a length of up to 80 characters.

```c
char name[81];
printf("What is your first name?\n");
scanf("%s", name);
```

What if someone has a *really* long first name?
int main(void) {
    char command[8];
    int balance = 0;
    while (1) {
        printf("Command? ('balance', 'deposit', or 'q' to quit): ");
        scanf("%s", command);
        if (strcmp(command, "balance") == 0) {
            printf("Your balance is: %d\n", balance);
        } else if (strcmp(command, "deposit") == 0) {
            printf("Enter your deposit amount: ");
            int dep;
            scanf("%d", &dep);
            balance += dep;
        } else if (strcmp(command, "q") == 0) {
            printf("Bye!\n"); break;
        } else {
            printf("Invalid command. Please try again.\n");
        }
    }
}
In this banking example, entering a long command causes C to write characters beyond the length of the `command` array. Eventually, it overwrites the memory where `balance` is stored.

This is known as a **buffer overrun** (or **buffer overflow**). The C language is especially susceptible to **buffer overruns**, which can cause serious stability and security problems.

In this introductory course, having an array with an appropriate length and using `scanf` is “good enough”.

In practice you would **never** use this insecure method for reading in a string.
If you need to read in a string that includes whitespace until a newline (\n) is encountered, the `gets` function can be used (CP:AMA 13.3).

It is also very susceptible to overruns, but is convenient to use in this course.

```c
char name[81];
printf("What is your full name?\n");
gets(name);
```
There are C library functions that are more secure than scanf and gets.

One popular strategy to avoid overruns is to only read in one character at a time (e.g., with `scanf("%c")` or `getchar`). For an example of using `getchar` to avoid overruns, see CP:AMA 13.3.
Two additional `<string.h>` library functions that are useful, but susceptible to buffer overruns are:

```c
strcpy(char *dest, const char *src)
```

overwrites the contents of `dest` with the contents of `src`.

```c
strcat(char *dest, const char *src)
```

copies (appends or concatenates) `src` to the end of `dest`.

You should always ensure that the `dest` array is large enough (and don’t forget the null terminator).
Consider this simple implementation of `my_strcpy`:

```c
char *my_strcpy(char *dst, const char *src) {
    char *d = dst;
    while (*src) {
        *d = *src;
        ++d; ++src;
    }
    *d = '\0';
    return dst;
}
```

with the following function call:

```c
char s[] = "spam";
my_strcpy(s + 4, s);
```

The null terminator of `src` is overwritten, so it will continue to fill up memory with `spamspamspam...` until a crash occurs.
While *writing* to a buffer can cause dangerous buffer overruns, *reading* an improperly terminated string can also cause problems.

```c
char c[3] = "cat";  // NOT properly terminated!
printf("%s\n", c);
printf("The length of c is: %d\n", strlen(c));

```

The string library has “safer” versions of many of the functions that stop when a maximum number of characters is reached.

For example, `strnlen`, `strncmp`, `strncpy` and `strncat`. 
String literals

C strings in quotations (e.g., "string") that are in an expression (i.e., not part of an array initialization) are known as string literals.

```c
printf("literal\n");

printf("literal %s\n", "another literal");

if (strcmp(s, "literal") == 0) ...

strcpy(dst, "literal");

int i = strlen("literal");

scanf("%d", &i);
```
String literal storage

Where are string literals stored?

For each string literal, a null-terminated const char array is created in the read-only data section.

In the code, the occurrence of the string literal is replaced with address of the corresponding array.

The “read-only” section is also known as the “literal pool”.
**example: string literals**

```c
void foo(int i, int j) {
    printf("i = %d\n", i);
    printf("the value of j is %d\n", j);
}
```

Although no name is actually given to each literal, it is helpful to imagine that one is:

```c
const char string_literal_1[] = "i = %d\n";
const char string_literal_2[] = "the value of j is %d\n";
```  

```c
void foo(int i, int j) {
    printf(string_literal_1, i);
    printf(string_literal_2, j);
}
```

You should not try to modify a string literal. The behaviour is undefined, and it causes an error in Seashell.
Note the subtle difference between the following two definitions:

```c
int main(void) {
    char a[] = "mutable char array";
    char *p = "constant string literal";
    //...
}
```

Once again, it is helpful to think of the string literal as a separately defined `const char array`.

```c
const char string_literal_1[] = "constant string literal";
```

```c
int main(void) {
    char a[] = "mutable char array";
    char *p = string_literal_1;
    //...
}
```
Arrays vs. pointers

Earlier, we said arrays and pointers are similar but different.

Consider again two similar string definitions:

```c
void f(void) {
    char a[] = "pointers are not arrays";
    char *p = "pointers are not arrays";
    ...
}
```

- The first reserves space for an initialized 24 character array (`a`) in the stack frame (24 bytes).

- The second reserves space for a `char` pointer (`p`) in the stack frame (8 bytes), initialized to point at a string literal (`const char array`) created in the read-only data section.
example: more arrays vs. pointers

```c
char a[] = "pointers are not arrays";
char *p = "pointers are not arrays";
char d[] = "different string";
```

*a* is a *char* array. The *identifier* *a* has a constant value (the address of the array), but the elements of *a* can be changed.

```c
a = d;              // INVALID
a[0] = 'P';         // VALID
```

*p* is a *char* pointer. *p* is initialized to point at a string literal, but *p* can be changed to point at any *char*.

```c
p[0] = 'P';         // INVALID (p points at a const literal)
p = d;              // VALID
p[0] = 'D';         // NOW VALID (p points at d)
```
An array is more similar to a **constant** pointer (that cannot change what it “points at”).

```c
int a[6] = {4, 8, 15, 16, 23, 42};
int * const p = a;
```

In most practical expressions `a` and `p` would be equivalent. The only significant differences between them are:

- `a` has the same value as `&a`, while `p` and `&p` have different values
- The size of `a` is 24 bytes, while `sizeof(p)` is 8
Arrays of Strings

An array of strings can be defined as a 2D array of `chars`, but it requires that each string is allocated the same fixed number of `chars` (regardless of the actual string length).

```c
char aos[][21] = {"my", "two dimensional", "char array"};
```

This is awkward because a function would need to know the fixed length in advance.

```c
void aos_function(char aos[][21], int num_strings) { ... }
```

This method is rarely used.

It is possible to send the fixed length as a separate parameter if the array is “re-interpreted” (cast) to a 1D array.
Instead, an array of pointers is more often used.

    char *aos[] = {"my awesome array", "of string", "literals"};

In the above example, `aos` is an array of pointers, with each pointer pointing to a string literal.

    // equivalent definition
    
    const char str_lit_0[] = "my awesome array";
    const char str_lit_1[] = "of string";
    const char str_lit_2[] = "literals";

    char *aos[] = {str_lit_0, str_lit_1, str_lit_2};

This array of pointers can be passed to a function, but as with all arrays, you must still pass the array length:

    void aos_function(char *aos[], int num_strings) { ... }
    // OR
    void aos_function(char **aos, int num_strings) { ... }
Until we learn how to use dynamic memory, defining an array of *mutable* strings is a little more awkward.

```c
char s0[] = "my mutable array";
char s1[] = "of strings";
char *aos[] = {s0, s1};
```

With an array of pointers to strings, you can access any `char` as if it was a 2D array of `chars`.

For example, `aos[0][1]` is `**(aos[0])**[1]`, which is `'y'`. 
Goals of this Section

At the end of this section, you should be able to:

- define and initialize arrays and strings
- use iteration to loop through arrays
- use pointer arithmetic
- explain how arrays are represented in the memory model, and how the array index operator (\[\]) uses pointer arithmetic to access array elements in constant time
- use both array index notation (\[\]) and array pointer notation and convert between the two
• use fixed-length arrays

• describe insertion sort, quicksort and binary search on a sorted array

• represent multi-dimensional data in a single-dimensional array

• explain and demonstrate the use of the null termination convention for strings

• explain string literals and the difference between defining a string array and a string pointer

• sort a string or sequence lexicographically
• use I/O with strings and explain the consequences of buffer overruns

• use `<string.h>` library functions (when provided with a well documented interface)
Efficiency

Readings: None
Algorithms

An *algorithm* is step-by-step description of *how* to solve a “problem”.

*Algorithms* are not restricted to computing. For example, every day you might use an algorithm to select which clothes to wear.

For most of this course, the “problems” are function descriptions (*interfaces*) and we work with *implementations* of algorithms that solve those problems.

The word *algorithm* is named after Muḥammad ibn Mūsā al-Khwārizmī (≈ 800 A.D.).
There are many objective and subjective methods for comparing algorithms:

- How easy is it to understand?
- How easy is it to implement?
- How accurate is it?
- How robust is it? (Can it handle errors well?)
- How adaptable is it? (Can it be used to solve similar problems?)
- How fast (efficient) is it?

In this course, we use **efficiency** to objectively compare algorithms.
Efficiency

The most common measure of efficiency is \textit{time efficiency}, or how long it takes an algorithm to solve a problem. Unless we specify otherwise, we \textbf{always mean time efficiency}.

Another efficiency measure is \textit{space efficiency}, or how much space (memory) an algorithm requires to solve a problem. We briefly discuss space efficiency at the end of this module.

The \textit{efficiency} of an algorithm may depend on its \textit{implementation}.

To avoid any confusion, we always measure the efficiency of a \textit{specific implementation} of an algorithm.
Running time

To quantify efficiency, we are interested in measuring the *running time* of an algorithm.

What unit of measure should we use? Seconds?

“My algorithm can sort one billion integers in 9.037 seconds”.

- What *year* did you make this statement?
- What machine & model did you use? (With how much RAM?)
- What computer language & operating system did you use?
- Was that the actual CPU time, or the total time elapsed?
- How accurate is the time measurement? Is the 0.037 relevant?
Measuring *running times* in seconds can be problematic.

What are the alternatives?

Typically, we measure the number of *elementary operations* required to solve the problem.

- In C, we can count the number of operations, or in other words, the number of *operators* executed.

- In Racket, we can count the total number of (substitution) *steps* required, although that can be deceiving for built-in functions †.

† We will revisit the issue of built-in functions later.
You are not expected to count the exact the number of operations.

We only count operations in these notes for illustrative purposes.

We introduce some simplification shortcuts soon.
Input size

What is the number of operations executed for this implementation?

```c
int sum_array(const int a[], int len) {
    int sum = 0;
    int i = 0;
    while (i < len) {
        sum = sum + a[i];
        i = i + 1;
    }
    return sum;
}
```

The running time **depends on the length** of the array.

If there are \( n \) items in the array, it requires \( 7n + 3 \) operations.

We are always interested in the running time **with respect to the size of the input**.
Traditionally, the variable $n$ is used to represent the size (or length) of the input. $m$ and $k$ are also popular when there is more than one input.

Often, $n$ is obvious from the context, but if there is any ambiguity you should clearly state what $n$ represents.

For example, with lists of strings, $n$ may represent the number of strings in the list, or it may represent the length of all of the strings in the list.

The running Time of an implementation is a function of $n$ and is written as $T(n)$. 
There may also be another attribute of the input that is also important.

For example, with trees, we use $n$ to represent the number of nodes in the tree and $h$ to represent the height of the tree.

In advanced algorithm analysis, $n$ may represent the number of bits required to represent the input, or the length of the string necessary to describe the input.
Algorithm Comparison

Problem: Write a function to determine if an array of positive integers contains at least $e$ even numbers and $o$ odd numbers.

```c
// check_array(a, len, e, o) determines if array a
// contains at least e even numbers and
// at least o odd numbers
// requires: len > 0
//           elements of a > 0
//           e, o >= 0
```

Homer and Bart are debating the best algorithm (strategy) for implementing `check_array`. 
Bart just wants to count the total number of odd numbers in the entire array.

```cpp
bool bart(const int a[], int len, int e, int o) {
    int odd_count = 0;
    for (int i = 0; i < len; i = i + 1) {
        odd_count = odd_count + (a[i] % 2);
    }
    return (odd_count >= o) && (len - odd_count >= e);
}
```

If there are \( n \) elements in the array, \( T(n) = 8n + 7 \).

Remember, you are not expected to calculate this precisely.
Homer is lazy, and he doesn’t want to check all of the elements in the array if he doesn’t have to.

```c
bool homer(const int a[], int len, int e, int o) {
    // only loop while it’s still possible
    while (len > 0 && e + o <= len) {
        if (a[len - 1] % 2 == 0) { // even case:
            if (e > 0) {
                e = e - 1; // only decrement e if e > 0
            }
        } else if (o > 0) {
            o = o - 1;
        }
        if (e == 0 && o == 0) {
            return true;
        }
        len = len - 1;
    }
    return false;
}
```
The problem with analyzing Homer’s code is that it depends not just on the length of the array, but on the contents of the array and the parameters $e$ and $o$.

```c
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// these will be fast:
bool fast1 = homer(a, 10, 0, 11); // false;
bool fast2 = homer(a, 10, 1, 0); // true;

// these will be slower:
bool slow1 = homer(a, 10, 5, 5); // true;
bool slow2 = homer(a, 10, 6, 4); // false;
```
For Homer’s code, the best case is when it can return immediately, and the worst case is when all of the array elements are visited.

For Bart’s code, the best case is the same as the worst case.

\[ T(n) = 4 \] (best case)
\[ T(n) = 17n + 1 \] (worst case)
\[ T(n) = 8n + 7 \] (all cases)

Which implementation is more efficient?

Is it more “fair” to compare against the best case or the worst case?
Worst case running time

Typically, we want to be conservative (*pessimistic*) and use the *worst case*.

Unless otherwise specified, the running time of an algorithm is the **worst case running time**.

Comparing the worst case, Bart’s implementation \((8n + 7)\) is more efficient than Homer’s \((17n + 1)\).

We may also be interested in the *average* case running time, but that analysis is typically much more complicated.
Big O notation

In practice, we are not concerned with the difference between the running times \((8n + 7)\) and \((17n + 1)\).

We are interested in the \textit{order} of a running time. The order is the “\textit{dominant}” term in the running time \textit{without any constant coefficients}.

The dominant term in both \((8n + 7)\) and \((17n + 1)\) is \(n\), and so they are both “\textit{order} \(n\)’’.

To represent \textit{orders}, we use \textit{Big O notation}. Instead of “\textit{order} \(n\)”, we use \(O(n)\).

We define Big O notation more formally later.
The “dominant” term is the term that grows the largest when $n$ is very large ($n \to \infty$). The order is also known as the “growth rate”.

In this course, we encounter only a few orders (arranged from smallest to largest):

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$

**Example: orders**

- $2016 = O(1)$
- $100000 + n = O(n)$
- $n + n \log n = O(n \log n)$
- $999n + 0.01n^2 = O(n^2)$
- $\frac{n(n+1)(2n+1)}{6} = O(n^3)$
- $n^3 + 2^n = O(2^n)$
When comparing algorithms, the most efficient algorithm is the one with the lowest order.

For example, an $O(n \log n)$ algorithm is more efficient than an $O(n^2)$ algorithm.

If two algorithms have the same order, they are considered equivalent.

Both Homer’s and Bart’s implementations are $O(n)$, so they are equivalent.
Big O arithmetic

When *adding* two orders, the result is the largest of the two orders.

- \( O(\log n) + O(n) = O(n) \)
- \( O(1) + O(1) = O(1) \)

When *multiplying* two orders, the result is the product of the two orders.

- \( O(\log n) \times O(n) = O(n \log n) \)
- \( O(1) \times O(n) = O(n) \)
There is no “universally accepted” Big O notation.

In many textbooks, and in this introductory course, the notation

\[ T(n) = 1 + 2n + 3n^2 = O(1) + O(n) + O(n^2) = O(n^2) \]

is acceptable.

In other textbooks, and in other courses, this notation may be too informal.

In CS 240 and CS 341 you will study orders and Big O notation much more rigourously.
Algorithm analysis

An important skill in Computer Science is the ability to analyze a function and determine the order of the running time.

In this course, our goal is to give you experience and work toward building your intuition:

```c
int sum_array(const int a[], int len) {
    int sum = 0;
    for (int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```

“Clearly, each element is visited once, so the running time of `sum_array` is $O(n)$”.

Contract update

You should include the time (efficiency) of each function that is not \( O(1) \) and is not obviously \( O(1) \).

If there is any ambiguity as to how \( n \) is measured, it should be specified.

// sum_array(const int a[], int len) sums the elements // of array a // time: \( O(n) \), \( n \) is the len of a
Analyzing simple functions

First, consider **simple** functions (without recursion or iteration).

```c
int max(int a, int b) {
    if (a > b) return a;
    return b;
}
```

If no other functions are called, there must be a fixed number of operators. Each operator is $O(1)$, so the running time is:

$$O(1) + O(1) + \ldots + O(1) = O(1)$$

If a simple function calls other functions, its running time will depend on those functions.
Built-in functions

Consider the following two implementations.

```c
// is_len_two(s) determines if the length of s is exactly 2

bool is_len_two_a(const char *s) {
    return strlen(s) == 2;
}

bool is_len_two_b(const char *s) {
    return s[0] && s[1] && (s[2] == 0);
}
```

The running time of \texttt{a} is \( O(n) \), while the running time of \texttt{b} is \( O(1) \).

When using a function that is built-in or provided by a module (library) you should always be aware of the running time.
C running times (strings & I/O)

*string.h* functions (e.g., `strlen`, `strcpy`) are $O(n)$, where $n$ is the length of the string. For `strcmp`, $n$ is the length of the smallest string.

*stdio.h* functions `printf` and `scanf` are $O(1)$, except when working with strings ("%s"), which are $O(n)$, where $n$ is the length of the string.

Note that the string literal used with `printf` must always be constant length (i.e., `printf("literal")`).
Racket running times (numeric)

When working with small integers (i.e., valid C integers), the Racket numeric functions are $O(1)$.

However, because Racket can handle arbitrarily large numbers it is more complicated.

For example, the running time to add two large positive integers is $O(\log n)$, where $n$ is the largest number.
Racket running times (lists)

Elementary list functions are $O(1)$:

- `cons` cons?
- `empty` empty?
- `rest` first
- `second` second
- `tenth`

List functions that process the full list are typically $O(n)$:

- `length` last
- `reverse` append

Abstract list functions (e.g., `map`, `filter`) depend on the consumed function, but are $O(n)$ for straightforward $O(1)$ functions.

The exception is Racket’s `sort`, which is $O(n \log n)$. 
Racket running times (equality)

We can assume $= \text{(numeric equality)}$ is $O(1)$.

symbol=? is $O(1)$, but

string=? is $O(n)$, where $n$ is the length of the smallest string.

Racket’s generic equal? is deceiving: its running time is $O(n)$, where $n$ is the “size” of the smallest argument.

Because (member e lst) depends on equal?, its running time is $O(nm)$ where $n$ is the length of the lst and $m$ is the size of e.

† This highlights another difference between symbols & strings.
Array efficiency

One of the significant differences between arrays and lists is that any element of an array can be accessed in constant time regardless of the index or the length of the array.

To access the \( i \)-th element in an array (e.g., \( a[i] \)) is always \( O(1) \).

To access the \( i \)-th element in a list (e.g., \( \text{list-ref} \)) is \( O(i) \).

Racket has a vector data type that is very similar to arrays in C.

\[
\text{(define v (vector 4 8 15 16 23 42))}
\]

Like C’s arrays, any element of a vector can be accessed by the \( \text{vector-ref} \) function in \( O(1) \) time.
Iterative analysis uses summations.

\[
\text{for (i = 1; i <= n; ++i) }
\begin{align*}
\text{printf("*");} \\
\}
\]

\[
T(n) = \sum_{i=1}^{n} O(1) = O(1) + \ldots + O(1) = n \times O(1) = O(n)
\]

Because we are primarily interested in orders,

\[
\sum_{i=0}^{n-1} O(x), \sum_{i=1}^{10n} O(x), \text{ or } \sum_{i=1}^{n/2} O(x)
\]

are equivalent* to \[
\sum_{i=1}^{n} O(x)
\]

* unless \( x \) is exponential (e.g., \( O(2^i) \)).
Procedure for iteration

1. Work from the *innermost* loop to the *outermost*

2. Determine the number of iterations in the loop (in the worst case) in relation to the size of the input \( n \) or an outer loop counter

3. Determine the running time per iteration

4. Write the summation(s) and simplify the expression

\[
\text{sum} = 0;
\text{for } (i = 0; i < n; ++i) \{
\text{sum} += i;
\}
\]

\[
\sum_{i=1}^{n} O(1) = O(n)
\]
Common summations

\[ \log n \sum_{i=1}^{n} O(1) = O(\log n) \]

\[ n \sum_{i=1}^{n} O(1) = O(n) \]

\[ n \sum_{i=1}^{n} O(n) = O(n^2) \]

\[ n \sum_{i=1}^{n} O(i) = O(n^2) \]

\[ n \sum_{i=1}^{n} O(i^2) = O(n^3) \]
The summation index should reflect the *number of iterations* in relation to the *size of the input* and does not necessarily reflect the actual loop counter values.

```c
k = n;  // n is size of the input
while (k > 0) {
    printf("*");
    k -= 10;
}
```

There are \( n/10 \) iterations. Because we are only interested in the order, \( n/10 \) and \( n \) are equivalent.

\[
\sum_{i=1}^{n/10} O(1) = O(n)
\]
When the loop counter changes geometrically, the number of iterations is often logarithmic.

```c
k = n; // n is size of the input
while (k > 0) {
    printf("*");
    k /= 10;
}
```

There are $\log_{10} n$ iterations.

$$\log n \sum_{i=1}^{\log n} O(1) = O(\log n)$$
When working with nested loops, evaluate the innermost loop first.

```c
for (i = 0; i < n; ++i) {
    for (j = 0; j < i; ++j) {
        printf("* ");
    }
    printf("\n");
}
```

**Inner loop:**

\[ \sum_{j=0}^{i-1} O(1) = O(i) \]

**Outer loop:**

\[ \sum_{i=0}^{n-1} (O(1) + O(i)) = O(n^2) \]
Do NOT put the **strlen** function within a loop.

```c
int char_count(char c, char *s) {
    int count = 0;
    for (int i=0; i < strlen(s); ++i) { // BAD !!!!
        if (s[i] == c) ++count;
    }
    return count;
}
```

By using an \( O(n) \) function (**strlen**) inside of the loop, the function becomes \( O(n^2) \) instead of \( O(n) \).

Unfortunately, this mistake is common amongst beginners.

This will be harshly penalized on assignments & exams.
Recurrence relations

To determine the running time of a recursive function we must determine the *recurrence relation*. For example,

\[ T(n) = O(n) + T(n - 1) \]

We can then look up the recurrence relation in a table to determine the *closed-form* (non-recursive) running time.

\[ T(n) = O(n) + T(n - 1) = O(n^2) \]

In later courses, you *derive* the closed-form solutions and *prove* their correctness.
The recurrence relations we encounter in this course are:

<table>
<thead>
<tr>
<th>Equation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) = O(1) + T(n - k_1) )</td>
<td>( = O(n) )</td>
</tr>
<tr>
<td>( T(n) = O(n) + T(n - k_1) )</td>
<td>( = O(n^2) )</td>
</tr>
<tr>
<td>( T(n) = O(n^2) + T(n - k_1) )</td>
<td>( = O(n^3) )</td>
</tr>
<tr>
<td>( T(n) = O(1) + T\left(\frac{n}{k_2}\right) )</td>
<td>( = O(\log n) )</td>
</tr>
<tr>
<td>( T(n) = O(1) + k_2 \cdot T\left(\frac{n}{k_2}\right) )</td>
<td>( = O(n) )</td>
</tr>
<tr>
<td>( T(n) = O(n) + k_2 \cdot T\left(\frac{n}{k_2}\right) )</td>
<td>( = O(n \log n) )</td>
</tr>
<tr>
<td>( T(n) = O(1) + T(n - k_1) + T(n - k'_1) )</td>
<td>( = O(2^n) )</td>
</tr>
</tbody>
</table>

where \( k_1, k'_1 \geq 1 \) and \( k_2 > 1 \)

This table will be provided on exams.
Procedure for recursive functions

1. Identify the order of the function *excluding* any recursion
2. Determine the size of the input for the next recursive call(s)
3. Write the full *recurrence relation* (combine step 1 & 2)
4. Look up the closed-form solution in a table

```
(define (sum lon)
  (cond 
      [(empty? lon) 0]
      [else (+ (first lon) (sum (rest lon)))])
```

1. non-recursive functions: $O(1)$ (empty?, first, rest)
2. size of the recursion: $n - 1$ (rest lon)
3. $T(n) = O(1) + T(n - 1)$ (combine 1 & 2)
4. $T(n) = O(n)$ (table lookup)
Revisiting sorting algorithms

No introduction to efficiency is complete without a discussion of sorting algorithms.

First we will analyze two recursive sorting algorithms in Racket.

For simplicity, we only consider sorting numbers.

When sorting strings or large data structures, you must also include the time to compare each element.

When analyzing sorting algorithms, one measure of running time is the number of comparisons.
Insertion sort

Recall *insertion sort* (from CS 135), where we start with an empty (sorted) sequence, and then *insert* each element into the sorted sequence, maintaining the order after each insert.

\[
T(n) = O(1) + T(n - 1) = O(n)
\]

\[
T(n) = O(n) + T(n - 1) = O(n^2)
\]
Merge Sort

In *merge sort*, the list is split into two separate lists. After the two lists are sorted they are merged together.

This is another example of a *divide and conquer* algorithm.

The lists are divided into two smaller problems, which are then sorted (*conquered*). The results are combined to solve the original problem.

For now, we can only easily implement merge sort in Racket.
For *merge sort*, we need a function to *merge* two sorted lists.

\[
\text{(define (merge slon1 slon2)} \\
\text{(cond [(empty? slon1) slon2]} \\
\text{[(empty? slon2) slon1]} \\
\text{[(< (first slon1) (first slon2))} \\
\text{ (cons (first slon1) (merge (rest slon1) slon2))]} \\
\text{[else (cons (first slon2) \\
\text{ (merge slon1 (rest slon2)))]})]
\]

If the size of the two lists are \(m\) and \(p\), then the recursive calls are either \([ (m - 1) \text{ and } p] \) or \([m \text{ and } (p - 1)] \).

However, if we define \(n = m + p\) (the combined size of both lists), then each recursive call is of size \((n - 1)\).

\[
T(n) = O(1) + T(n - 1) = O(n)
\]
Now, we can complete \texttt{merge-sort}.

\begin{verbatim}
(define (merge-sort lon)
  (define len (length lon))
  (define mid (quotient len 2))
  (define left (drop-right lon mid)) ; \(O(n)\)
  (define right (take-right lon mid)) ; \(O(n)\)
  (cond [\((\leq\ len\ 1)\) lon]
        [else (merge (merge-sort left)
                      (merge-sort right))])

T(n) = O(n) + 2T(n/2) = O(n \log n)
\end{verbatim}

The built-in Racket function \texttt{sort} uses \texttt{merge-sort}. 
Selection sort

Recall our C implementation of selection sort:

```c
void selection_sort(int a[], int len) {
    for (int i=0; i < len - 1; ++i) {
        int pos = i;
        for (int j = i + 1; j < len; ++j) {
            if (a[j] < a[pos]) {
                pos = j;
            }
        }
        swap(&a[i], &a[pos]);
    }
}
```

\[ T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} O(1) = O(n^2) \]
Quick sort

In our C implementation of quick sort, we:

1. select the first element of the array as our “pivot”. $O(1)$

2. move all elements that are larger than the pivot to the back of the array. $O(n)$.

3. move (“swap”) the pivot into the correct position. $O(1)$.

4. recursively sort the “smaller than” sub-array and the “larger than” sub-array. $T(?)$

The analysis of step 4 is a little trickier.
When the pivot is in “the middle” it splits the sublists equally, so
\[ T(n) = O(n) + 2T\left(\frac{n}{2}\right) = O(n \log n) \]

But that is the best case. In the worst case, the “pivot” is the smallest (or largest element), so one of the sublists is empty and the other is of size \((n - 1)\).
\[ T(n) = O(n) + T(n - 1) = O(n^2) \]

Despite its worst case behaviour, quick sort is still popular and in widespread use. The average case behaviour is quite good and there are straightforward methods that can be used to improve the selection of the pivot.

It is part of the C standard library (see Section 12).
## Sorting summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>best case</th>
<th>worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>quick sort</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
Binary search

In Section 08, we implemented binary search on a sorted array.

```c
int find_sorted(int item, const int a[], int len) {
    // ...
    while (low <= high) {
        int mid = low + (high - low) / 2;
        // ...
        if (a[mid] < item) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    //...
}
```

In each iteration, the size of the search range \( n = \text{high} - \text{low} \) was halved, so the running time is:

\[
T(n) = \sum_{i=1}^{\log_2{n}} O(1) = O(\log n)
\]
Algorithm Design

In this introductory course, the algorithms we develop will be mostly straightforward.

To provide some insight into *algorithm design*, we will introduce a problem that is simple to describe, but hard to solve efficiently.

We will present four different algorithms to solve this problem, each with a different running time.
The maximum subarray problem

Problem: Given an array of integers, find the maximum sum of any contiguous sequence (subarray) of elements.

For example, for the following array:

\[
\begin{array}{ccccccccccc}
31 & -41 & 59 & 26 & -53 & 58 & 97 & -93 & -23 & 84 \\
\end{array}
\]

the maximum sum is 187:

\[
\begin{array}{ccccccccccc}
31 & -41 & \boxed{59} & 26 & -53 & 58 & \boxed{97} & -93 & -23 & 84 \\
\end{array}
\]

This problem has many applications, including pattern recognition in artificial intelligence.
Solution A: $O(n^3)$

// for every start position i and ending position j
// loop between them (k) summing elements

```c
int max_subarray(const int a[], int len) {
    int maxsofar = 0;
    for (i = 0; i < len; ++i) {
        for (j = i; j < len; ++j) {
            int sum = 0;
            for (k = i; k <= j; ++k) {
                sum += a[k];
            }
            maxsofar = max(maxsofar, sum);
        }
    }
    return maxsofar;
}
```

$T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} O(1) = O(n^3)$
Solution B: $O(n^2)$

// for every start position $i$,
// check if the sum from $i...j$ is the max

int max_subarray(const int a[], int len) {
    int maxsofar = 0;
    for (i = 0; i < len; ++i) {
        int sum = 0;
        for (j = i; j < len; ++j) {
            sum += a[j];
            maxsofar = max(maxsofar, sum);
        }
    }
    return maxsofar;
}

$T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} O(1) = O(n^2)$
Solution C: $O(n \log n)$

We will only describe this recursive divide and conquer approach.

1. Find the midpoint position $m$. $O(1)$

2. Find (a) the maximum subarray from $(0 \ldots m-1)$, and 
   (b) the maximum subarray from $(m+1 \ldots \text{len}-1)$. $2T(n/2)$

3. Find (c) the maximum subarray that includes $m$. $O(n)$

4. Find the maximum of (a), (b) and (c). $O(1)$

$T(n) = O(n) + 2T(n/2) = O(n \log n)$
Solution D: $O(n)$

// for each position i, keep track of
// the maximum subarray ending at i

int max_subarray(const int a[], int len) {
    int maxsofar = 0;
    int maxendhere = 0;
    for (i = 0; i < len; ++i) {
        maxendhere = max(maxendhere + a[i], 0);
        maxsofar = max(maxsofar, maxendhere);
    }
    return maxsofar;
}
Space complexity

The *space complexity* of an algorithm is the amount of *additional memory* that the algorithm requires to solve the problem.

While we are mostly interested in *time complexity*, there are circumstances where space is more important.

If two algorithms have the same time complexity but different space complexity, it is likely that the one with the lower space complexity is faster.
Consider the following two Racket implementations of a function to sum a list of numbers.

\[
\begin{align*}
\text{(define (sum lst)}
& \text{(cond ((empty? lst) 0)]} \\
& \text{[else (+ (first lst) (sum (rest lst)))])))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (asum lst)}
& \text{(define (asum/acc lst sofar)}
\text{(cond ((empty? lst) sofar]}
\text{[else (asum/acc (rest lst)
\text{(+ (first lst) sofar)]})])})
\end{align*}
\]

(asm/acc lst 0))

Both functions produce the same result and both functions have a time complexity \( T(n) = O(n) \).

The significant difference is that \textbf{asum} uses \textit{accumulative} recursion.
If we examine the substitution steps of \texttt{sum} and \texttt{asum}, we get some insight into their differences.

\begin{verbatim}
(sum '(1 1 1))
=> (+ 1 (sum '(1 1)))
=> (+ 1 (+ 1 (sum '(1))))
=> (+ 1 (+ 1 (+ 1 (sum empty))))
=> (+ 1 (+ 1 (+ 1 0)))
=> (+ 1 (+ 1 1))
=> (+ 1 2)
=> 3
\end{verbatim}

\begin{verbatim}
(asum '(1 1 1))
=> (asum/acc '(1 1 1) 0)
=> (asum/acc '(1 1) 1)
=> (asum/acc '(1) 2)
=> (asum/acc empty 3)
=> 3
\end{verbatim}

The \texttt{sum} expression “grows” to $O(n)$ +’s, but the \texttt{asum} expression does not use any additional space.
The measured run-time of `asum` is *significantly* faster than `sum` (in an experiment with a list of one million 1’s, over 40 times faster).

`sum` uses $O(n)$ space, whereas `asum` uses $O(1)$ space.

But both functions make the same number of recursive calls, how is this explained?

The difference is that `asum` uses tail recursion.
A function is *tail recursive* if the recursive call is always the last expression to be evaluated (the “tail”).

Typically, this is achieved by using accumulative recursion and providing a partial result as one of the parameters.

With tail recursion, the previous stack frame can be reused for the next recursion (or the previous frame can be discarded before the new stack frame is created).

Tail recursion is more space efficient and avoids stack overflow.

Many modern C compilers detect and take advantage of tail recursion.
Big O revisited

We now revisit *Big O notation* and define it more formally.

\[ O(g(n)) \] is the **set** of all functions whose “order” is **less than or equal** to \( g(n) \).

\[
\begin{align*}
n^2 & \in O(n^{100}) \\
n^3 & \in O(2^n)
\end{align*}
\]

While you can say that \( n^2 \) is in the set \( O(n^{100}) \), it’s not very useful information.

In this course, we always want the **most appropriate** order, or in other words, the **smallest** correct order.
Big O describes the *asymptotic* behaviour of a function.

This is **different** than describing the **worst case** behaviour of a algorithm.

Many confuse these two topics but they are completely **separate concepts**. You can asymptotically define the best case and the worst case behaviour of an algorithm.

For example, the best case insertion sort is $O(n)$, while the worst case is $O(n^2)$. 
A slightly more formal definition of Big O is

\[ f(n) \in O(g(n)) \iff f(n) \leq c \cdot g(n) \]

for large \( n \) and some positive number \( c \).

This definition makes it clear why we “ignore” constant coefficients.

For example,

\[ 9n \in O(n) \quad \text{for } c = 10, \quad 9n \leq 10n, \text{ and} \]

\[ 0.01n^3 + 1000n^2 \in O(n^3) \]

for \( c = 1001, \quad 0.01n^3 + 1000n^2 \leq 1001n^3 \)
The full definition of Big O is

\[ f(n) \in O(g(n)) \iff \exists c, n_0 > 0, \forall n \geq n_0, f(n) \leq c \cdot g(n) \]

\( f(n) \) is in \( O(g(n)) \) if there exists a positive \( c \) and \( n_0 \) such that for any value of \( n \geq n_0 \), \( f(n) \leq c \cdot g(n) \).
In later CS courses, you will use the formal definition of Big O to prove algorithm behaviour more rigorously.

There are other asymptotic functions in addition to Big O.

(for each of the following, \( \exists n_0 > 0, \forall n \geq n_0 \ldots \))

\[
\begin{align*}
f(n) \in \omega(g(n)) & \iff \forall c > 0, \, c \cdot g(n) \leq f(n) \\
f(n) \in \Omega(g(n)) & \iff \exists c > 0, \, c \cdot g(n) \leq f(n) \\
f(n) \in \Theta(g(n)) & \iff \exists c_1, c_2 > 0, \, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \\
f(n) \in O(g(n)) & \iff \exists c > 0, \, f(n) \leq c \cdot g(n) \\
f(n) \in o(g(n)) & \iff \forall c > 0, \, f(n) \leq c \cdot g(n)
\end{align*}
\]

\( O(g(n)) \) is often used when \( \Theta(g(n)) \) is more appropriate.
Goals of this Section

At the end of this section, you should be able to:

• use the new terminology introduced (e.g., algorithm, time efficiency, running time, order)

• compute the order of an expression

• explain and demonstrate the use of Big O notation and how \( n \) is used to represent the size of the input

• determine the “worst case” running time for a given implementation
• deduce the running time for many built-in functions

• avoid common design mistakes with expensive operations such as `strlen`

• analyze a recursive function, determine its recurrence relation and look up its closed-form running time in a provided lookup table

• analyze an iterative function and determine its running time

• explain and demonstrate the use of the four sorting algorithms presented
• analyze your own code to ensure it achieves a desired running time

• describe the formal definition of Big O notation and its asymptotic behaviour

• explain space complexity, and how it relates to tail recursion

• use running times in your contracts
Dynamic Memory & ADTs in C

Readings: CP:AMA 17.1, 17.2, 17.3, 17.4
The heap

The heap is the final section in the C memory model. It can be thought of a big “pile” (or “pool”) of memory that is available to your program.

Memory is dynamically “borrowed” from the heap. We call this allocation.

When the borrowed memory is no longer needed, it can be “returned” and possibly reused. We call this deallocation.

If too much memory has already been allocated, attempts to borrow additional memory fail.
Unfortunately, there is also a *data structure* known as a heap, and the two are unrelated.

To avoid confusion, prominent computer scientist Donald Knuth campaigned to use the name “free store” or the “memory pool”, but the name “heap” has stuck.

A similar problem arises with “the stack” region of memory because there is also a Stack ADT. However, their behaviour is very similar so it is far less confusing.
malloc

The **malloc** (memory allocation) function obtains memory from the heap *dynamically*. It is provided in `<stdlib.h>`.

```c
// malloc(s) requests s bytes of memory from the heap
// and returns a pointer to a block of s bytes, or
// NULL if not enough memory is available
// time: O(1) [close enough for this course]
```

For example, if you want enough space for an array of 100 *ints*:

```c
int *my_array = malloc(100 * sizeof(int));
```

or an array of *n* `struct posns`:

```c
struct posn *my_posn_array = malloc(n * sizeof(struct posn));
```
You should always use `sizeof` with `malloc` to improve portability and to improve communication.

Seashell will allow

```c
int *my_array = malloc(400);
```

instead of

```c
int *my_array = malloc(100 * sizeof(int));
```

but the latter is much better style and is more portable.
Strictly speaking, the type of the `malloc` parameter is `size_t`, which is a special type produced by the `sizeof` operator. `size_t` and `int` are different types of integers.

Seashell is mostly forgiving, but in other C environments using an `int` when C expects a `size_t` may generate a warning.

The proper `printf` placeholder to print a `size_t` is `%zd`. 
The declaration for the `malloc` function is:

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

The return type is a `(void *)` (*void pointer*), a special pointer that can point at *any* type.

```c
int *pi = malloc(sizeof(int));
struct posn *pp = malloc(sizeof(struct posn));
```
int main(void) {
    int *arr1 = malloc(10 * sizeof(int));
    int *arr2 = malloc(5 * sizeof(int));
    //...
}

example: visualizing the heap
An unsuccessful call to `malloc` returns `NULL`.

In practice it’s good style to check every `malloc` return value and gracefully handle a `NULL` instead of crashing.

```c
int *my_array = malloc(n * sizeof(int));
if (my_array == NULL) {
    printf("Sorry dude, I’m out of memory! I’m exiting....\n");
    exit(EXIT_FAILURE);
}
```

In the “real world” you should always perform this check, but in this course, you do **not** have to check for a `NULL` return value unless instructed otherwise.

In these notes, we omit this check to save space.
The heap memory provided by malloc is uninitialized.

```c
int *p = malloc(sizeof(int));
printf("the mystery value is: %d\n", *p);
```

Although malloc is very complicated, for the purposes of this course, you can assume that malloc is $O(1)$.

There is also a calloc function which essentially calls malloc and then “initializes” the memory by filling it with zeros. calloc is $O(n)$, where $n$ is the size of the block.
free

For every block of memory obtained through `malloc`, you must eventually **free** the memory (when the memory is no longer in use).

```c
// free(p) returns memory at p back to the heap
// requires: p must be from a previous malloc
// effects: the memory at p is invalid
// time: O(1)
```

In the Seashell environment, you **must** `free` every block.

```c
int *my_array = malloc(n * sizeof(int));
// ...
// ...
// ...
free(my_array);
```
Invalid after free

Once a block of memory is freed, reading from or writing to that memory is invalid and may cause errors (or unpredictable results).

Similarly, it is invalid to free memory that was not returned by a malloc or that has already been freed.

```c
int *p = malloc(sizeof(int));
free(p);
int k = *p; // INVALID
*p = 42;    // INVALID
free(p);    // INVALID
p = NULL;   // GOOD STYLE
```

Pointer variables may still contain the address of the memory that was freed, so it is often good style to assign NULL to a freed pointer variable.
Memory leaks

A memory leak occurs when allocated memory is not eventually freed.

Programs that leak memory may suffer degraded performance or eventually crash.

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

In this example, the address from the original `malloc` has been overwritten.

That memory is now “lost” (or leaked) and so it can never be freed.
Garbage collection

Many modern languages (including Racket) have a *garbage collector*.

A garbage collector *detscts* when memory is no longer in use and *automatically* frees memory and returns it to the heap.

One disadvantage of a garbage collector is that it can be slow and affect performance, which is a concern in high performance computing.
Merge sort

In Section 09 we saw a Racket implementation of the *divide and conquer* algorithm **merge sort** that is $O(n \log n)$.

In merge sort, the data is split into two smaller groups. After each smaller group is sorted, they are **merged** together.

To simplify our C implementation, we will use a **merge** helper function.
// merge(dest, src1, len1, src2, len2) modifies dest to contain
// the elements from both src1 and src2 in sorted order
// requires: length of dest is at least (len1 + len2)
// src1 and src2 are sorted
// effects: modifies dest
// time: O(n), where n is len1 + len2

void merge(int dest[], const int src1[], int len1,
            const int src2[], int len2) {
    int pos1 = 0;
    int pos2 = 0;
    for (int i=0; i < len1 + len2; ++i) {
        if (pos1 == len1 || (pos2 < len2 && src2[pos2] < src1[pos1])) {
            dest[i] = src2[pos2];
            ++pos2;
        } else {
            dest[i] = src1[pos1];
            ++pos1;
        }
    }
}
void merge_sort(int a[], int len) {
    if (len <= 1) return;
    int llen = len / 2;
    int rlen = len - llen;

    int *left = malloc(llen * sizeof(int));
    int *right = malloc(rlen * sizeof(int));

    for (int i=0; i < llen; ++i) left[i] = a[i];
    for (int i=0; i < rlen; ++i) right[i] = a[i + llen];

    merge_sort(left, llen);
    merge_sort(right, rlen);

    merge(a, left, llen, right, rlen);

    free(left);
    free(right);
}

This implementation of merge sort is also $O(n \log n)$. 
Using dynamic (heap) memory, a function can obtain memory that persists after the function has returned.

```c
int *build_array(int len) {
    assert(len > 0);
    int *a = malloc(len * sizeof(int));
    for (int i=0; i < len; ++i) {
        a[i] = i;
    }
    return a; // array exists beyond function return
}
```

The caller (client) is responsible for freeing the memory (the contract should communicate this).
The `<string.h>` function `strdup` makes a duplicate of a string.

```c
char *my_strdup(const char *s) {
    char *newstr = malloc((strlen(s) + 1) * sizeof(char));
    strcpy(newstr, s);
    return newstr;
}
```

Recall that the `strcpy(dest, src)` copies the characters from `src` to `dest`, and that the `dest` array must be large enough.

When allocating memory for strings, don’t forget to the include space for the null terminator.

`strdup` is not officially part of the C standard, but common.
Resizing arrays

Because `malloc` requires the size of the block of memory to be allocated, it does not seem to solve the problem:

“What if we do not know the length of an array in advance?”

To solve this problem, we can `resize` an array by:

- creating a new array
- copying the items from the old to the new array
- `freeing` the old array
example: resizing an array

As we will see shortly, this is not how it is done in practice, but this is an illustrative example.

```c
// my_array has a length of 100
int *my_array = malloc(100 * sizeof(int));

// stuff happens...

// oops, my_array now needs to have a length of 101
int *old = my_array;
my_array = malloc(101 * sizeof(int));
for (int i=0; i < 100; ++i) {
    my_array[i] = old[i];
}
free(old);
```
realloc

To make resizing arrays easier, there is a `realloc` function.

```c
// realloc(p, newsize) resizes the memory block at p
//    to be newsize and returns a pointer to the
//    new location, or NULL if unsuccessful
// requires: p must be from a previous malloc/realloc
// effects: the memory at p is invalid (freed)
// time: O(n), where n is newsize
```

Similar to our previous example, `realloc` preserves the contents from the old array location.

```c
int *my_array = malloc(100 * sizeof(int));
// stuff happens...
my_array = realloc(my_array, 101 * sizeof(int));
```
The pointer returned by `realloc` may actually be the *original* pointer, depending on the circumstances.

Regardless, after `realloc` only the new returned pointer can be **used**. You should assume that the parameter of `realloc` was **freed** and is now **invalid**.

Typically, `realloc` is used to request a larger size and the additional memory is *uninitialized*.

If the size is smaller, the extraneous memory is discarded.

```
realloc(NULL, s) behaves the same as malloc(s).
realloc(ptr, 0) behaves the same as free(ptr).
```
Although rare, in practice,

```c
my_array = realloc(my_array, newsize);
```
could possibly cause a memory leak if an “out of memory” condition occurs.

In C99, an unsuccessful `realloc` returns `NULL` and the original memory block is not freed.

```c
// safer use of realloc
int *tmp = realloc(my_array, newsize);
if (tmp) {
    my_array = tmp;
} else {
    // handle out of memory condition
}
```
String I/O: strings of unknown size

In Section 08 we saw how reading in strings can be susceptible to buffer overruns.

```c
char str[81];
int retval = scanf("%s", str);
```

The target array is often oversized to ensure there is capacity to store the string. Unfortunately, regardless of the length of the array, a buffer overrun may occur.

To solve this problem we can continuously resize (`realloc`) an array while reading in only one character at a time.
// readstr() reads in a non-whitespace string from I/O
// or returns NULL if unsuccessful
// effects: allocates memory (caller must free)

char *readstr(void) {
    char c;
    if (scanf(" %c", &c) != 1) return NULL; // ignore initial WS
    int len = 1;
    char *str = malloc(len * sizeof(char));
    str[0] = c;
    while (1) {
        if (scanf("%c", &c) != 1) break;
        if (c == ' ' || c == '\n') break;
        ++len;
        str = realloc(str, len * sizeof(char));
        str[len - 1] = c;
    }
    str = realloc(str, (len + 1) * sizeof(char));
    str[len] = '\0';
    return str;
}
Amortized analysis

Unfortunately, the running time of `readstr` is $O(n^2)$, where $n$ is the length of the string.

This is because `realloc` is $O(n)$ and occurs inside of the loop.

A better approach might be to allocate more memory than necessary and only call `realloc` when the array is “full”.

A popular strategy is to double the size of the array when it is full.

Similar to working with maximum-length arrays, we need to keep track of the “actual” length in addition to the allocated length.
char *readstr(void) {
    char c;
    if (scanf(" %c", &c) != 1) return NULL; // ignore initial WS
    int maxlen = 1;
    int len = 1;
    char *str = malloc(maxlen * sizeof(char));
    str[0] = c;
    while (1) {
        if (scanf("%c", &c) != 1) break;
        if (c == ' ' || c == '
') break;
        if (len == maxlen) {
            maxlen *= 2;
            str = realloc(str, maxlen * sizeof(char));
        }
        ++len;
        str[len - 1] = c;
    }
    str = realloc(str, (len + 1) * sizeof(char));
    str[len] = '\0';
    return str;
}
With our “doubling” strategy, most iterations will be $O(1)$, unless it is necessary to resize (realloc) the array.

The resizing time for the first 32 iterations would be:

\[ 2, 4, 0, 8, 0, 0, 0, 16, 0, 0, 0, 0, 0, 0, 0, 32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 64 \]

For $n$ iterations, the total resizing time is at most:

\[ 2 + 4 + 8 + \ldots + \frac{n}{4} + \frac{n}{2} + n + 2n = 4n - 2 = O(n). \]

By using this doubling strategy, the total run time for readstr is now only $O(n)$.

In other words, the amortized (“average”) time for each iteration is:

\[ O(n)/n = O(1). \]
ADTs in C

With dynamic memory, we now have the ability to implement an Abstract Data Type (ADT) in C.

In Section 02, the first ADT we saw was a simple account ADT, which stored a username and a password. It demonstrated information hiding, which provides both security and flexibility.

We will also need to use opaque structures (incomplete declarations without fields), as introduced in Section 06.
example: account ADT

In the **interface**, we only provide an *incomplete declaration*. In addition to the normal operations, we provide functions to **create** and **destroy** instances of the ADT.

```c
// account.h -- a simple account ADT module

struct account; // incomplete

// create_account(username, password) creates an account
// with the given username and password
// effects: allocates memory (client must call destroy_account)
struct account *create_account(const char *username,
                               const char *password);

// destroy_account(acc) removes all memory for acc
// effects: memory at acc is free'd and invalid
void destroy_account(struct account *acc);
```
Because the interface only provides an incomplete declaration, the **client** does not know the fields of the **account** structure.

The client can only define a **pointer** to the structure, which is returned by **create_account**.

```c
// client.c
#include "account.h"

c char username[9];
char password[41];
// ...

struct account *my_account = create_account(username, password);
// ...

destroy_account(my_account);
```
The *complete* structure declaration only appears in the implementation.

```c
// account.c

struct account {
    char *uname;
    char *pword;
};
```
create_account returns a pointer to a new account.

```c
struct account *create_account(const char *username,
                               const char *password) {

    struct account *a = malloc(sizeof(struct account));
    a->uname = malloc((strlen(username) + 1) * sizeof(char));
    strcpy(a->uname, username);
    a->pword = malloc((strlen(password) + 1) * sizeof(char));
    strcpy(a->pword, password);

    return a;
}
```

it makes duplicates of the username and password strings provided by the client.
In C, our ADT also requires a `destroy_account` to free the memory created (both the fields and the structure itself).

```c
void destroy_account(struct account * a) {
    free(a->username);
    free(a->password);
    free(a);
}
```

The remaining operations are straightforward.

```c
const char * get_username(const struct account * acc) {
    return acc->uname;
}
```

```c
bool is_correct_password(const struct account * acc, const char * word) {
    return (strcmp(acc->pword, word) == 0);
}
```
Implementing a Stack ADT

As discussed in Section 02, the account ADT illustrates the principles of an ADT, but it is not a typical ADT.

The **Stack ADT** (one of the *Collection ADTs*) is more representative.

The interface is nearly identical to the stack implementation from Section 08 that demonstrated *maximum-length arrays*.

The only differences are: it uses an opaque structure, it provides *create* and *destroy* functions, and there is no maximum: it can store an arbitrary number of integers.
// stack.h (INTERFACE)

struct stack;

struct stack *create_stack(void);

bool stack_is_empty(const struct stack *s);

int stack_top(const struct stack *s);

int stack_pop(struct stack *s);

void stack_push(int item, struct stack *s);

void stack_destroy(const struct stack *s);
The Stack ADT uses the “doubling” strategy.

// stack.c (IMPLEMENTATION)

struct stack {
    int len;
    int maxlen;
    int *data;
};

struct stack *create_stack(void) {
    struct stack *s = malloc(sizeof(struct stack));
    s->len = 0;
    s->maxlen = 1;
    s->data = malloc(s->maxlen * sizeof(int));
    return s;
}
The doubling is implemented in \texttt{push}.

\texttt{destroy} must \texttt{free} the field and the structure itself.

\begin{verbatim}
// Time: O(1) [amortized]

void stack_push(int item, struct stack *s) {
    assert(s);
    if (s->len == s->maxlen) {
        s->maxlen *= 2;
        s->data = realloc(s->data, s->maxlen * sizeof(int));
    }
    s->data[s->len] = item;
    s->len += 1;
}

void stack_destroy(struct stack *s) {
    free(s->data);
    free(s);
}
\end{verbatim}
The remaining operations are identical to the maximum-length implementation.

```c
bool stack_is_empty(const struct stack *s) {
    assert(s);
    return s->len == 0;
}

int stack_top(const struct stack *s) {
    assert(s);
    assert(s->len);
    return s->data[s->len - 1];
}

int stack_pop(struct stack *s) {
    assert(s);
    assert(s->len);
    s->len -= 1;
    return s->data[s->len];
}
```
As discussed earlier, the *amortized* run-time for *push* is $O(1)$.

You will use *amortized* analysis in CS 240 and in CS 341.

In this implementation, we never “shrink” the array when items are popped.

A popular strategy is to reduce the size when the length reaches $\frac{1}{4}$ of the maximum capacity. Although more complicated, this also has an *amortized* run-time of $O(1)$ for an arbitrary sequence of *pushes* and *pops*.

Languages that have a built-in resizable array (e.g., C++’s vector) often use a similar “doubling” strategy.
Goals of this Section

At the end of this section, you should be able to:

• describe the heap

• use the functions malloc, realloc and free to interact with the heap

• explain that the heap is finite, and demonstrate how to use check malloc for success

• describe memory leaks, how they occur, and how to prevent them
• describe the doubling strategy, and how it can be used to manage dynamic arrays to achieve an amortized $O(1)$ run-time for additions

• create dynamic resizable arrays in the heap

• write functions that create and return a new struct

• document dynamic memory side-effects in contracts
Linked Data Structures

Readings: CP:AMA 17.5
Linked lists

Racket’s list type is more commonly known as a *linked list*. Each *node* contains an *item* and a *link* (*pointer*) to the *next* node in the list.

The *link* in the *last node* is a *sentinel value*.

In Racket we use *empty*, and in C we use a *NULL* pointer.
Linked lists are usually represented as a link (pointer) to the \textit{front}.

Unlike arrays, linked list nodes are \textbf{not} arranged sequentially in memory. There is no fast and convenient way to “jump” to the $i$-th element. The list must be \textbf{traversed} from the \textit{front}. Traversing a linked list is $O(n)$. 
A significant advantage of a linked list is that its length can easily change, and the length does not need to be known in advance.

The memory for each node is allocated dynamically (i.e., using dynamic memory).
Functional vs. Imperative approach

In Section 04, we discussed some of the differences between the functional and imperative programming paradigms.

The core concept of a linked list data structure is independent of any single paradigm.

However, the approach used to implement linked list functions are often very different.

Programming with linked lists further illustrates the differences between the two paradigms.
Dynamic memory in Racket

Dynamic memory in Racket is mostly “hidden”.

The cons function dynamically creates a new linked list node.

In other words, inside of every cons is a hidden malloc.

Structure constructors also use dynamic memory (e.g., make-posn or simply posn in full Racket).

list and quote list notation '(1 2 3) implicitly use cons.
In the functional programming paradigm, functions always produce **new** values rather than changing existing ones.

Consider a function that “squares” a list of numbers.

- In the *functional* paradigm, the function **must** produce a new list, because there is no *mutation*.

- in the *imperative* paradigm, the function is more likely to **mutate** an existing list instead of producing a new list.
sqr-list uses cons to construct a new list:

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

(define a '(10 3 5 7))
(define b (sqr-list a))
Of course, in an imperative language (e.g., C) it is also possible to write a “square list” function that follows the functional paradigm and generates a new list.

This is another example of why clear communication (purposes and contracts) is so important.

In practice, most imperative list functions perform mutation. If the caller wants a new list (instead of mutating an existing one), they can first make a copy of the original list and then mutate the new copy.
Mixing paradigms

Problems may arise if we naïvely use the functional paradigm in an imperative environment without considering the consequences.

This is especially important in C, where there is no garbage collector.

<table>
<thead>
<tr>
<th>Functional (Racket)</th>
<th>Imperative (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no mutation</td>
<td>mutation</td>
</tr>
<tr>
<td>garbage collector</td>
<td>no garbage collector</td>
</tr>
<tr>
<td>hidden pointers</td>
<td>explicit pointers</td>
</tr>
</tbody>
</table>

The following example highlights the potential problems.
Consider an `insert` function (used in *insertion sort*).

```scheme
;; (insert n slon) inserts n into a sorted list of numbers
(define (insert n slon)
  (cond [\(\text{empty? slon}\) (cons n empty)]
        [(\(\leq n \text{ (first slon)}\)) (cons n slon)]
        [else (cons (first slon) (insert n (rest slon)))]
))

(define a '(10 20 50 100))
(define b (insert 30 a))

What will the memory diagram look like for `a` and `b`?

The lists *share the last two nodes*. 
example: more node sharing in Racket

(define a '(4 5 6))
(define b (cons 3 a))
(define c (append '(1 2) b))
In Racket, lists can share nodes with no negative consequences. It is *transparent* because there is *no mutation*, and there is a garbage collector.

In an imperative language like C, this configuration is problematic.

- If we apply a mutative function such as “square list” on a, then some of the elements of b will unexpectedly change.
- If we explicitly `free` all of the memory for list a, then list b will become invalid.
To avoid mixing paradigms, we will use the following guidelines when implementing linked lists in C:

- lists will not share nodes
- new nodes will only be created (malloc’d) when necessary (inserting nodes and creating new lists).
In Racket, lists generated with \texttt{cons} are immutable.

There is a special \texttt{mcons} function to generate a mutable list.

This is one of the significant differences between the Racket language and the Scheme language.

In the Scheme language, lists generated with \texttt{cons} are mutable.
Linked lists in C

We declare a *linked list node* (llnode) that stores an “item” and a link (pointer) to the next node. For the *last node*, next is **NULL**.

```c
struct llnode {
    int item;
    struct llnode *next;
};
```

A C structure can contain a *pointer* to its own structure type. This is the first *recursive data structure* we have seen in C.

There is no “official” way of implementing a linked list in C. The CP:AMA textbook and other sources use slightly different conventions.
In this Section we use a **wrapper strategy**, where we wrap the link to the first node of a list inside of another structure (`llist`).

```c
struct llist {
    struct llnode *front;
};
```

This wrapper strategy makes some of the following code more straightforward.

It also makes it easier to avoid having lists share nodes (mixing paradigms).

In Appendix A.8, we present examples that do not use a wrapper and briefly discuss the advantages and disadvantages.
To dynamically create a list, we define a `list_create` function.

```c
struct llist *list_create(void) {
    struct llist *lst = malloc(sizeof(struct llist));
    lst->front = NULL;
    return lst;
}
```

```c
int main(void) {
    struct llist *lst = list_create();
    // ...
}
```
We need to add items to our linked list.

The following code creates a **new** node, and inserts it at the **front** of the list.

```c
void add_front(int i, struct llist *lst) {
    struct llnode *node = malloc(sizeof(struct llnode));
    node->item = i;
    node->next = lst->front;
    lst->front = node;
}
```
example: building a linked list

The following code builds a linked list by reading in integers from the input.

```c
int main(void) {
    struct llist *lst = list_create();

    while(1) {
        int i;
        if (scanf("%d", &i) != 1) break;
        add_front(i, lst);
    }
    // ...
}
```
Traversing a list

We can traverse a list iteratively or recursively.

When iterating through a list, we typically use a (llnode) pointer to keep track of the “current” node.

```c
int length(const struct llist *lst) {
    int len = 0;
    struct llnode *node = lst->front;
    while (node) {
        ++len;
        node = node->next;
    }
    return len;
}
```

Remember (node) will be false at the end of the list (NULL).
When using recursion, remember to recurse on a node (llnode) not the wrapper list itself (llist).

```c
int length_nodes(struct llnode *node) {
    if (node == NULL) return 0;
    return 1 + length_nodes(node->next);
}
```

You can write a corresponding wrapper function:

```c
int list_length(struct llist *lst) {
    return length_nodes(lst->front);
}
```
Destroying a list

In C, we don’t have a garbage collector, so we must be able to free our linked list. We need to free every node and the list wrapper.

When using an iterative approach, we are going to need two node pointers to ensure that the nodes are freed in a safe way.

```c
void list_destroy(struct llist *lst) {
    struct llnode *curnode = lst->front;
    while (curnode) {
        struct llnode *nextnode = curnode->next;
        free(curnode);
        curnode = nextnode;
    }
    free(lst);
}
```
For more advanced list traversal functions, the technique of maintaining more than one node pointer is often necessary.

It may take some practice and diagrams to master this technique.

For extra practice, consider this slightly different implementation:

```c
void list_destroy(struct llist *lst) {
    struct llnode *curnode = lst->front;
    while (curnode) {
        struct llnode *backup = curnode;
        curnode = curnode->next;
        free(backup);
    }
    free(lst);
}
```
With a recursive approach, it is more convenient to free the rest of the list before we free the first node.

```c
void free_nodes(struct llnode *node) {
    if (node) {
        free_nodes(node->next);
        free(node);
    }
}

void list_destroy(struct llist *lst) {
    free_nodes(lst->front);
    free(lst);
}
```
Duplicating a list

Previously, we used the “square list” function to illustrate the differences between the functional and imperative paradigms.

```c
// list_sqr(lst) squares each item in lst
// effects: modifies lst

void list_sqr(struct llist *lst) {
    struct llnode *node = lst->front;
    while (node) {
        node->item *= node->item;
        node = node->next;
    }
}
```

But what if we do want a new list that is squared instead of mutating an existing one?
One solution is to provide a `list_dup` function, that makes a 
duplicate of an existing list.

The recursive function is the most straightforward.

```c
struct llnode *dup_nodes(struct llnode *oldnode) {
    struct llnode *newnode = NULL;
    if (oldnode) {
        newnode = malloc(sizeof(struct llnode));
        newnode->item = oldnode->item;
        newnode->next = dup_nodes(oldnode->next);
    }
    return newnode;
}

struct llist *list_dup(struct llist *oldlist) {
    struct llist *newlist = list_create();
    newlist->front = dup_nodes(oldlist->front);
    return newlist;
}
```
The iterative solution is more complicated:

```c
struct llist *list_dup(struct llist *oldlist) {
    struct llist *newlist = list_create();
    struct llnode *oldnode = oldlist->front;
    struct llnode *prevnode = NULL;
    while (oldnode) {
        struct llnode *newnode = malloc(sizeof(struct llnode));
        newnode->item = oldnode->item;
        newnode->next = NULL;
        if (prevnode) {
            prevnode->next = newnode;
        } else {
            newlist->front = newnode;
        }
        prevnode = newnode;
        oldnode = oldnode->next;
    }
    return newlist;
}
```
Imperative insert

Earlier, we saw how the Racket (functional) implementation of `insert` (into a sorted list) would be problematic in C.

For an `insert` function in C, we expect the following behaviour:

```c
insert( 5, a);
insert(30, a);
```

![Diagram showing the insertion of 5 and 30 into a sorted list][1]

---

[1]: https://example.com/diagram.png
// insert(i, slst) inserts i into sorted list slst
// effects: modifies slst
// time: $O(n)$, where $n$ is the length of slst

void insert(int i, struct llist * slst) {
    if (slst->front == NULL || i < slst->front->item) {
        add_front(i, slst);
    } else {
        struct llnode * prevnode = slst->front;
        while (prevnode->next && i > prevnode->next->item) {
            prevnode = prevnode->next;
        }
        struct llnode * newnode = malloc(sizeof(struct llnode));
        newnode->item = i;
        newnode->next = prevnode->next;
        prevnode->next = newnode;
    }
}
Removing nodes

In Racket, the rest function does not actually remove the first element, instead it provides a pointer to the next node.

In C, we can implement a function that removes the first node.

```c
void remove_front(struct llist *lst) {
    assert(lst->front);
    struct llnode *backup = lst->front;
    lst->front = lst->front->next;
    free(backup);
}
```
Removing a node from an arbitrary list position is more complicated.

// remove_item(i, lst) removes the first occurrence of i in lst
// returns true if item is successfully removed

bool remove_item(int i, struct llist *lst) {
    if (lst->front == NULL) return false;
    if (lst->front->item == i) {
        remove_front(lst);
        return true;
    }
    struct llnode *prevnode = lst->front;
    while (prevnode->next && i != prevnode->next->item) {
        prevnode = prevnode->next;
    }
    if (prevnode->next == NULL) return false;
    struct llnode *backup = prevnode->next;
    prevnode->next = prevnode->next->next;
    free(backup);
    return true;
}
Revisiting the wrapper approach

Throughout these slides we have used a wrapper strategy, where we wrap the link to the first node inside of another structure (llist).

Some of the advantages of this strategy are:

- cleaner function interfaces
- reduced need for double pointers
- reinforces the imperative paradigm
- less susceptible to misuse and list corruption
The disadvantages of the wrapper approach include:

- slightly more awkward recursive implementations
- extra “special case” code around the first item

However, there is one more significant advantage of the wrapper approach: additional information can be stored in the list structure.

See Appendix A.8 for more details.
Consider that we are writing an application where the length of a linked list will be queried often.

Typically, finding the length of a linked list is $O(n)$.

However, we can store (or “cache”) the length in the wrapper structure, so the length can be retrieved in $O(1)$ time.

```c
struct llist {
    struct llnode *front;
    int length;
};
```
Naturally, other list functions would have to update the length as necessary:

- `list_create` would initialize length to zero
- `add_front` would increment length
- `remove_front` would decrement length
- `etc.`
Data integrity

The introduction of the length field to the linked list may seem like a great idea to improve efficiency.

However, it introduces new ways that the structure can be corrupted. What if the length field does not accurately reflect the true length?

For example, imagine that someone implements the remove_item function, but forgets to update the length field?

Or a naïve coder may think that the following statement removes all of the nodes from the list.

```c
lst->length = 0;
```
Whenever the same information is stored in more than one way, it is susceptible to integrity (consistency) issues.

Advanced testing methods can often find these types of errors, but you must exercise caution.

If data integrity is an issue, it is often better to repackage the data structure as a separate ADT module and only provide interface functions to the client.

This is an example of security (protecting the client from themselves).
Queue ADT

A queue is like a “lineup”, where new items go to the “back” of the line, and the items are removed from the “front” of the line. While a stack is LIFO, a queue is FIFO (first in, first out).

Typical queue ADT operations:

- **add_back**: adds an item to the end of the queue
- **remove_front**: removes the item at the front of the queue
- **front**: returns the item at the front
- **is_empty**: determines if the queue is empty
A Stack ADT can be easily implemented using a dynamic array (as we did in Section 10) or with a linked list.

While it is possible to implement a Queue ADT with a dynamic array, the implementation is a bit tricky. Queues are typically implemented with linked lists.

The only concern is that an add_back operation is normally $O(n)$. However, if we maintain a pointer to the back (last element) of the list, in addition to a pointer to the front of the list, we can implement add_back in $O(1)$.

Maintaining a back pointer is a popular modification to a traditional linked list, and another reason to use a wrapper.
// queue.h

// all operations are O(1) (except destroy)

struct queue;

struct queue *queue_create(void);

void queue_add_back(int i, struct queue *q);

int queue_remove_front(struct queue *q);

int queue_front(struct queue *q);

bool queue_is_empty(struct queue *q);

void queue_destroy(struct queue *q);
// queue.c (IMPLEMENTATION)

struct llnode {
    int item;
    struct llnode *next;
};

struct queue {
    struct llnode *front;
    struct llnode *back;  // <--- NEW
};

struct queue *queue_create(void) {
    struct queue *q = malloc(sizeof(struct queue));
    q->front = NULL;
    q->back = NULL;
    return q;
}

CS 136 Fall 2017 11: Linked Data Structures
void queue_add_back(int i, struct queue *q) {
    struct llnode *node = malloc(sizeof(struct llnode));
    node->item = i;
    node->next = NULL;
    if (q->front == NULL) {
        q->front = node;
    } else {
        q->back->next = node;
    }
    q->back = node;
}

int queue_remove_front(struct queue *q) {
    assert(q->front);
    int retval = q->front->item;
    struct llnode *backup = q->front;
    q->front = q->front->next;
    free(backup);
    if (q->front == NULL) q->back = NULL;
    return retval;
}
The remainder of the Queue ADT is straightforward.

```c
int queue_front(struct queue *q) {
    assert(q->front);
    return q->front->item;
}

bool queue_is_empty(struct queue *q) {
    return q->front == NULL;
}

void queue_destroy(struct queue *q) {
    while (!queue_is_empty(q)) {
        queue_remove_front(q);
    }
    free(q);
}
```
Node augmentation strategy

In an **node augmentation strategy**, each *node* is *augmented* to include additional information about the node or the structure.

For example, a **dictionary** node can contain both a *key* (item) and a corresponding *value*.

Or for a **priority queue**, each node can additionally store the priority of the item.
The most common node augmentation for a linked list is to create a **doubly linked list**, where each node also contains a pointer to the *previous* node. When combined with a *back* pointer in a wrapper, a doubly linked list can add or remove from the front and back in \( O(1) \) time.

Many programming environments provide a Double-Ended Queue (dequeue or deque) ADT, which can be used as a Stack or a Queue ADT.
Trees

At the implementation level, *trees* are very similar to linked lists.

Each node can *link* to more than one node.
Tree terminology

- The **root node** has no **parent**, all others have exactly one.
- Nodes can have multiple **children**.
- In a **binary tree**, each node has at most two children.
- A **leaf node** has no children.
- The **height** of a tree is the maximum possible number of nodes from the root to a leaf (inclusive).
- The height of an empty tree is zero.
- The **size** of a tree is the number of nodes.
Binary Search Trees (BSTs)

Binary Search Tree (BSTs) enforce the **ordering property**: for every node with an item $i$, all items in the left child subtree are less than $i$, and all items in the right child subtree are greater than $i$. 

```
  40
 /   \
20    50
 / \
10 30
   \
   60
```
Mixing paradigms

As with linked lists, we have to be careful not to mix functional and imperative paradigms, especially when adding nodes. The following example visualizes what Racket produces when a node (45) is added to the BST illustrated earlier.
Our *BST node* (*bstnode*) is very similar to our linked list node definition.

```c
struct bstnode {
    int item;
    struct bstnode *left;
    struct bstnode *right;
};
```

```c
struct bst {
    struct bstnode *root;
};
```

In CS 135, BSTs were used as *dictionaries*, with each node storing both a key and a value. Traditionally, a BST only stores a single item, and additional values can be added as *node augmentations* if required.
As with linked lists, we will need a function to create a new BST.

```c
// bst_create() creates a new BST
// effects: allocates memory: call bst_destroy

struct bst *bst_create(void) {
    struct bst *t = malloc(sizeof(struct bst));
    t->root = NULL;
    return t;
}
```
Before writing code to *insert* a new node, first we write a helper to create a new *leaf* node.

```c
struct bstnode *new_leaf(int i) {
    struct bstnode *leaf = malloc(sizeof(struct bstnode));
    leaf->item = i;
    leaf->left = NULL;
    leaf->right = NULL;
    return leaf;
}
```
As with lists, we can write tree functions *recursively* or *iteratively*. We need to **recurse** on *nodes*. This code emulates a functional approach, but is careful to only allocate one new (leaf) node.

```c
struct bstnode *insert_bstnode(int i, struct bstnode *node) {
    if (node == NULL) {
        node = new_leaf(i);
    } else if (i < node->item) {
        node->left = insert_bstnode(i, node->left);
    } else if (i > node->item) {
        node->right = insert_bstnode(i, node->right);
    } // else do nothing, as item already exists
    return node;
}

void bst_insert(int i, struct bst *t) {
    t->root = insert_bstnode(i, t->root);
}
```
The iterative version is similar to the linked list approach.

```c
void bst_insert(int i, struct bst *t) {
    struct bstnode *curnode = t->root;
    struct bstnode *prevnode = NULL;
    while (curnode) {
        if (curnode->item == i) return;
        prevnode = curnode;
        if (i < curnode->item) {
            curnode = curnode->left;
        } else {
            curnode = curnode->right;
        }
    }
    if (prevnode == NULL) { // tree was empty
        t->root = new_leaf(i);
    } else if (i < prevnode->item) {
        prevnode->left = new_leaf(i);
    } else {
        prevnode->right = new_leaf(i);
    }
}
```
example: building a BST

The following code builds a BST by reading in integers from the input.

```c
int main(void) {

    struct bst *t = bst_create();

    while(1) {
        int i;
        if (scanf("%d", &i) != 1) break;
        bst_insert(i, t);
    }
    // ...
}
```
Trees and efficiency

What is the efficiency of \texttt{bst\_insert}?

The \textit{worst case} is when the tree is \textit{unbalanced}, and every node in the tree must be visited.

In this example, the running time of \texttt{bst\_insert} is $O(n)$, where $n$ is the number of nodes in the tree.
The running time of \texttt{bst\_insert} is $O(h)$: it depends more on the height of the tree ($h$) than the size of the tree ($n$).

The definition of a \textit{balanced tree} is a tree where the height ($h$) is $O(\log n)$.

Conversely, an \textit{unbalanced} tree is a tree with a height that is \textbf{not} \textit{O}($\log n$). The height of an unbalanced tree is $O(n)$.

Using the \texttt{bst\_insert} function we provided, inserting the nodes in \textit{sorted order} creates an \textit{unbalanced} tree.
With a **balanced** tree, the running time of standard tree functions (e.g., insert, remove, search) are all $O(\log n)$.

With an **unbalanced** tree, the running time of each function is $O(h)$.

A **self-balancing tree** "re-arranges" the nodes to ensure that tree is always balanced.

With a good self-balancing implementation, all standard tree functions *preserve the balance of the tree* and have an $O(\log n)$ running time.

In CS 240 and CS 341 you will see **self-balancing trees**.

Self-balancing trees often use node augmentations to store extra information to aid in re-balancing.
Size node augmentation

A popular tree node augmentation is to store in each node the size of its subtree.

```c
struct bstnode {
    int item;
    struct bstnode * left;
    struct bstnode * right;
    int size;        // *****NEW
};
```

This augmentation allows us to retrieve the size of the tree in $O(1)$ time.

It also allows us to implement a select function in $O(h)$ time. `select(k)` finds the $k$-th smallest item in the tree.
example: size node augmentation

![Diagram of a tree with node sizes]

- Node 40 has a size of 6
- Node 20 has a size of 3
- Node 50 has a size of 2
- Node 10 has a size of 1
- Node 30 has a size of 1
- Node 60 has a size of 1
The following code illustrates how to select the $k$-th item in a BST with a *size* node augmentation.

```c
int select_node(int k, struct bstnode *node) {
    assert(node && 0 <= k && k < node->size);
    int left_size = 0;
    if (node->left) left_size = node->left->size;
    if (k < left_size) return select_node(k, node->left);
    if (k == left_size) return node->item;
    return select_node(k - left_size - 1, node->right);
}

int bst_select(int k, struct bst *t) {
    return select_node(k, t->root);
}
```

`select(0, t)` finds the smallest item in the tree.
Array-based trees

For some types of trees, it is possible to use an array to store a tree.

- the root is stored at \( a[0] \)
- for the node at \( a[i] \), its left is stored at \( a[2i+1] \)
- its right is stored at \( a[2i+2] \)
- its parent is stored at \( a[(i-1)/2] \)
- a special sentinel value can be used to indicate an empty node
- a tree of height \( h \) requires an array of size \( 2^h - 1 \)
  (a dynamic array can be realloc’d as the tree height grows)
example: array-based tree representation

left: 2i+1
	right: 2i+2

40  20  50  10  30  -  60
Array-based trees are often used to implement “complete trees”, where there are no *empty* nodes, and every level of the tree is filled (except the bottom).

The *heap* data structure (not the section of memory) is often implemented as a complete tree in an array.

For *self-balancing* trees, the self-balancing (e.g., rotations) is often more awkward in the array notation. However, arrays work well with *lazy* rebalancing, where a rebalancing occurs infrequently (i.e., when a large inbalance is detected). The tree can be rebalanced in $O(n)$ time, typically achieving *amortized* $O(\log n)$ operations.
Dictionary ADT (revisited)

The dictionary ADT (also called a map, associative array, or symbol table), is a collection of pairs of keys and values. Each key is unique and has a corresponding value, but more than one key may have the same value.

Typical dictionary ADT operations:

- **lookup**: for a given key, retrieve the corresponding value or “not found”
- **insert**: adds a new key/value pair (or replaces the value of an existing key)
- **remove**: deletes a key and its value
In the following example, we implement a Dictionary ADT using a BST data structure.

As in CS 135, we will use int keys and string values.

```c
// dictionary.h

struct dictionary;

struct dictionary *dict_create(void);

void dict_insert(int key, const char *val, struct dictionary *d);

const char *dict_lookup(int key, struct dictionary *d);

void dict_remove(int key, struct dictionary *d);

void dict_destroy(struct dictionary *d);
```
Using the same `bstnode` structure, we *augment* each node by adding an additional `value` field.

```c
struct bstnode {
    int item; // key
    char *value; // additional value (augmentation)
    struct bstnode *left;
    struct bstnode *right;
};

struct dictionary {
    struct bstnode *root;
};

struct dictionary *dict_create(void) {
    struct dictionary *d = malloc(sizeof(struct dictionary));
    d->root = NULL;
    return d;
}
```
When inserting key/value pairs to the dictionary, we make a *copy* of the string passed by the client. When removing nodes, we also *free* the value.

If the client tries to insert a duplicate key, we replace the old value with the new value.

The following *recursive* implementation of the *insert* operation is nearly identical to our previous *bst_insert*. The differences are noted with comments.
struct bstnode *insert_bstnode(int key, const char *val, struct bstnode *node) {
    if (node == NULL) {
        node = malloc(sizeof(struct bstnode));
        node->item = key;
        node->value = my_strdup(val); // make copy
        node->left = NULL;
        node->right = NULL;
    } else if (key < node->item) {
        node->left = insert_bstnode(key, val, node->left);
    } else if (key > node->item) {
        node->right = insert_bstnode(key, val, node->right);
    } else { // key == node->item: must replace the old value
        free(node->value);
        node->value = my_strdup(val);
    }
    return node;
}

void dict_insert(int key, const char *val, struct dictionary *d) {
    d->root = insert_bstnode(key, val, d->root);
}
This implementation of the lookup operation will return NULL if unsuccessful.

```c
const char *dict_lookup(int key, struct dictionary *d) {
    struct bstnode *curnode = d->root;
    while (curnode) {
        if (curnode->item == key) {
            return curnode->value;
        }
        if (key < curnode->item) {
            curnode = curnode->left;
        } else {
            curnode = curnode->right;
        }
    }
    return NULL;
}
```
There are several different ways of removing a node from a BST.

We implement \textit{remove} with the following strategy:

- If the node with the key ("key node") is a leaf, we remove it.
- If one child of the key node is empty (\texttt{NULL}), the other child is "promoted" to replace the key node.
- Otherwise, we find the node with the \textit{next largest} key ("next node") in the tree (\textit{i.e.}, the smallest key in the right subtree). We replace the key/value of the key node with the key/value of the next node, and then remove the next node from the right subtree.
void dict_remove(int key, struct dictionary *d) {
    d->root = remove_bstnode(key, d->root);
}

struct bstnode *remove_bstnode(int key, struct bstnode *node) {
    // key did not exist:
    if (node == NULL) return NULL;
    // search for the node that contains the key
    if (key < node->item) {
        node->left = remove_bstnode(key, node->left);
    } else if (key > node->item) {
        node->right = remove_bstnode(key, node->right);
    } else if // continued on next page ...
        // (we have now found the key node)
If either child is **NULL**, the node is removed (**free’d**) and the other child is promoted.

```c
} else if (node->left == NULL) {
    struct bstnode *new_root = node->right;
    free(node->value);
    free(node);
    return new_root;
} else if (node->right == NULL) {
    struct bstnode *new_root = node->left;
    free(node->value);
    free(node);
    return new_root;
} else // continued...
    // (neither child is NULL)
```
Otherwise, we replace the key/value at this node with next largest key/value, and then remove the next key from the right subtree.

```c
} else {
    // find the next largest key
    struct bstnode *next = node->right;
    while (next->left) {
        next = next->left;
    }
    // remove the old value
    free(node->value);
    // replace the key/value of this node
    node->item = next->item;
    node->value = my_strdup(next->value);
    // remove the next largest key
    node->right = remove_bstnode(next->item, node->right);
}
return node;
}```
Finally, the recursive `destroy` operation frees the children and the (string) value before itself.

```c
void free_bstnode(struct bstnode *node) {
    if (node) {
        free_bstnode(node->left);
        free_bstnode(node->right);
        free(node->value);
        free(node);
    }
}

void dict_destroy(struct dictionary *d) {
    free_bstnode(d->root);
    free(d);
}
```
Graphs

Linked lists and trees can be thought of as “special cases” of a **graph** data structure. Graphs are the only core data structure we are not working with in this course.

**Graphs** link **nodes** with **edges**. Graphs may be undirected (i) or directed (ii), allow cycles (ii) or be acyclic (iii), and have labeled edges (iv) or unlabeled edges (iii).
Goals of this Section

At the end of this section, you should be able to:

- use the new linked list and tree terminology introduced
- use linked lists and trees with a recursive or iterative approach
- use wrapper structures and node augmentations to improve efficiency
- explain why an unbalanced tree can affect the efficiency of tree functions
Abstract Data Types (ADTs) & Design

Readings: CP:AMA 19.5, 17.7 (qsort)
Selecting a data structure

In Computer Science, every data structure is some combination of the following “core” data structures.

- primitives (e.g., an `int`)
- structures (i.e., `struct`)
- arrays
- linked lists
- trees
- graphs
Selecting an appropriate data structure is important in program design. Consider a situation where you are choosing between an array, a linked list, and a BST. Some design considerations are:

- How frequently will you add items? remove items?
- How frequently will you search for items?
- Do you need to access an item at a specific position?
- Do you need to preserve the “original sequence” of the data, or can it be re-arranged?
- Can you have duplicate items?

Knowing the answers to these questions and the efficiency of each data structure function will help you make design decisions.
Sequenced data

Consider the following strings to be stored in a data structure.

"Bob" "Alice" "Charlie"

Is the original sequencing important?

- If it’s the result of a competition, yes: "Bob" is in first place. We call this type of data sequenced.

- If it’s a list of friends to invite to a party, it is not important. We call this type of data unsequenced or "rearrangeable".

If the data is sequenced, then a data structure that sorts the data (e.g., a BST) is likely not an appropriate choice. Arrays and linked lists are better suited for sequenced data.
### Data structure comparison: sequenced data

<table>
<thead>
<tr>
<th>Function</th>
<th>Dynamic Array</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>item at</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>search</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>insert at</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>insert_front</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert_back</td>
<td>$O(1)^*$</td>
<td>$O(n)^\dagger$</td>
</tr>
<tr>
<td>remove at</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>remove_front</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove_back</td>
<td>$O(1)$</td>
<td>$O(n)^\diamond$</td>
</tr>
</tbody>
</table>

* amortized

† $O(1)$ with a wrapper strategy and a back pointer

◊ $O(1)$ with a *doubly* linked list and a back pointer.
# Data structure comparison: unsequenced (sorted) data

<table>
<thead>
<tr>
<th>Function</th>
<th>Sorted Dynamic</th>
<th>Sorted Linked List</th>
<th>Regular BST</th>
<th>Self-Balancing BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)^*$</td>
<td>$O(n)^{†}$</td>
</tr>
<tr>
<td>search</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

* $O(h)$ with a size augmentation.

† $O(\log n)$ with a size augmentation.

select($k$) finds the $k$-th smallest item in the data structure.

For example, select(0) finds the smallest element.
example: design decisions

• An array is a good choice if you frequently access elements at specific positions (random access).

• A linked list is a good choice for sequenced data if you frequently add and remove elements at the start.

• A self-balancing BST is a good choice for unsequenced data if you frequently search for, add and remove items.

• A sorted array is a good choice if you rarely add/remove elements, but frequently search for elements and select the data in sorted order.
Implementing collection ADTs

A significant benefit of a collection ADT is that a client can use it “abstractly” without worrying about how it is implemented.

In practice, ADT modules are usually well-written, optimized and have a well documented interface.

In this course, we are interested in how to implement ADTs.
Typically, the collection ADTs are implemented as follows.

- **Stack**: linked lists or dynamic arrays
- **Queue**: linked lists
- **Sequence**: linked lists or dynamic arrays.
  Some libraries provide two different ADTs (e.g., a list and a vector) that provide the same interface but have different operation run-times.
- **Dictionary** (and **Sets**): self-balanced BSTs or hash tables*

* A hash table is typically an array of linked lists (more on hash tables in CS 240).
Beyond integers

In Section 10, we presented an implementation of a Stack ADT that only supported a stack of integers.

What if we want to have a stack of a different type?

There are three common strategies to solve this “type” problem in C:

• create a separate implementation for each possible item type,
• use a typedef to define the item type, or
• use a void pointer type (void *).

The first option is unwieldy and unsustainable. We first discuss the typedef strategy, and then the void * strategy.
We don’t have this problem in Racket because of dynamic typing.

This is one reason why Racket and other dynamic typing languages are so popular.

Some statically typed languages have a *template* feature to avoid this problem. For example, in C++ a stack of integers is defined as:

```
stack<int> my_int_stack;
```

The stack ADT (called a stack “container”) is built-in to the C++ STL (standard template library).
typedef

The C `typedef` keyword allows you to create your own “type” from previously existing types. This is typically done to improve the readability of the code, or to hide the type (for security or flexibility).

```c
typedef int Integer;
typedef int *IntPtr;

Integer i;
IntPtr p = &i;
```

It is common to use a different coding style (we use CamelCase) when defining a new “type” with `typedef`. 
typedef is often used to simplify complex declarations (e.g., function pointer types).

typedef int (* MapFn)(int);

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

void array_map(MapFn f, int a[], int len) { // <- cleaner!
    for (int i=0; i < len; ++i) {
        a[i] = f(a[i]);
    }
}

int main(void) {
    int arr[6] = {4, 8, 15, 16, 23, 42};
    array_map(add1, arr, 6);
    MapFn f = add1;
    array_map(f, arr, 6);
    //...
}

Stack ADT: cleaner interface

struct stack;

// use [Stack] instead of [struct stack *]
typedef struct stack *Stack;

// operations:

Stack stack_create(void);

bool stack_is_empty(Stack s);

int stack_top(Stack s);

int stack_pop(Stack s);

void stack_push(int item, Stack s);

void stack_destroy(Stack s);
Some programmers consider it poor style to use `typedef` to “abstract” that a type is a `pointer`, as it may accidentally lead to memory leaks.

A compromise is to use a type name that reflects that the type is a pointer (e.g., `StackPtr`).

The Linux kernel programming style guide recommends avoiding `typedefs` altogether.
The “typedef” strategy is to define the type of each item (ItemType) in a separate header file ("item.h") that can be provided by the client.

    // item.h
typedef int ItemType; // for stacks of ints

or...

    // item.h
typedef struct posn ItemType; // for stacks of posns

The ADT module would then be implemented with this ItemType.

    #include "item.h"

    void stack_push(Stack S, ItemType i);

    ItemType stack_top(Stack s);
Having a client-defined `ItemType` is a popular approach for small applications, but it does not support having two different stack types in the same application.

The `typedef` approach can also be problematic if `ItemType` is a pointer type and it is used with dynamic memory. In this case, calling `destroyStack` may create a memory leak.

Memory management issues are even more of a concern with the third approach (`void *`).
void pointers

The **void** pointer (**void** * *) is the closest C has to a “generic” type, which makes it suitable for ADT implementations.

**void** pointers can point to “any” type, and are essentially just memory addresses. They can be converted to any other type of pointer, but **they cannot be directly dereferenced**.

```c
int i = 42;
void *vp = &i;
int j = *vp;    // INVALID
int *ip = vp;
int k = *ip;    // VALID
```
While some C conversions are implicit (e.g., char to int), there is a C language feature known as casting, which explicitly “forces” a type conversion.

To cast an expression, place the destination type in parentheses to the left of the expression. This example casts a “void *” to an “int *”, which can then be dereferenced:

```c
int i = 42;
void *vp = &i;
int j = *(int *)vp;
```

A useful application of casting is to avoid integer division when working with floats (see CP:AMA 7.4):

```c
float one_half = ((float) 1) / 2;
```
Implementing ADTs with void pointers

There are two complications that arise from implementing ADTs with void pointers:

- **Memory management** is a problem because a protocol must be established to determine if the client or the ADT is responsible for freeing item data.

- **Comparisons** are a problem because some ADTs must be able to compare items when searching and sorting.

Both problems also arise in the `typedef` approach.
The solution to the **memory management** problem is to make the **ADT interface explicitly clear** whose responsibility it is to **free** any item data: the client or the ADT. Both choices present problems.

For example, when it is the **client’s responsibility** to **free** items, care must be taken to retrieve and **free** every item before a **destroy** operation, otherwise **destroy** could cause memory leaks. A precondition to the **destroy** operation could be that the ADT is empty (all items have been removed).
When it is the **ADT’s responsibility**, problems arise if the items contain additional dynamic memory.

For example, consider if we desire a sequence of accounts, where each account is an instance of the account ADT we implemented earlier. If the sequence `remove_at` operation simply calls `free` on the item, it creates a memory leak as the username and password are not freed.

To solve this problem, the client can provide a customized `free` function for the ADT to call (e.g., `destroy_account`).
example: stack interface with void pointers

// (partial interface) CLIENT’S RESPONSIBILITY TO FREE ITEMS

// stack_push(s, i) puts item i on top of the stack
//   NOTE: The caller should not free the item until it is popped
void stack_push(Stack s, void *i);

// stack_top(s) returns the top but does not pop it
//   NOTE: The caller should not free the item until it is popped
const void *stack_top(Stack s);

// stack_pop(s) removes the top item and returns it
//   NOTE: The caller is responsible for freeing the item
void *stack_pop(Stack s);

// stack_destroy(s) destroys the stack
// requires: The stack must be empty (all items popped)
void stack_destroy(Stack s);
#include "stack.h"

// this program reverses the characters typed
int main(void) {
    Stack s = create_Stack();
    while(1) {
        char c;
        if (scanf("%c", &c) != 1) break;
        char *newc = malloc(sizeof(char));
        *newc = c;
        push(s, newc);
    }
    while(!is_empty(s)) {
        char *oldc = pop(s);
        printf("%c", *oldc);
        free(oldc);
    }
    destroy_Stack(s);
}
Comparison functions

The dictionary and set ADTs often sort and compare their items, which is a problem if the item types are void pointers.

To solve this problem, we can provide the ADT with a comparison function (pointer) when the ADT is created.

The ADT would then just call the comparison function whenever a comparison is necessary.
Comparison functions follow the `strcmp(a,b)` convention where return values of $-1$, $0$ and $1$ correspond to $(a < b)$, $(a == b)$, and $(a > b)$ respectively.

```c
// a comparison function for integers
int compare_ints(const void * a, const void * b) {
    const int *ia = a;
    const int *ib = b;
    if (*ia < *ib) { return -1; }
    if (*ia > *ib) { return 1; }
    return 0;
}
```

A `typedef` can be used to make declarations less complicated.

```c
typedef int (*CompFuncPtr) (const void *, const void *);
```
example: dictionary

// dictionary.h (partial interface)

struct dictionary;
typedef struct dictionary *Dictionary;

typedef int (*DictKeyCompare) (const void *, const void *);

// create a dictionary that uses key comparison function f
Dictionary dict_create(DictKeyCompare f);

// lookup key k in Dictionary d
const void *dict_lookup(Dictionary d, void *k);
// dictionary.c (partial implementation)

struct bstnode {
    void * item; // key
    void * value; // additional value (augmentation)
    struct bstnode * left;
    struct bstnode * right;
};

struct dictionary {
    struct bstnode * root;
    DictKeyCompare key_compare; // function pointer
};

Dictionary dict_create(DictKeyCompare f) {
    Dictionary d = malloc(sizeof(struct dictionary));
    d->root = NULL;
    d->key_compare = f;
    return d;
}
This implementation of `dict_lookup` illustrates how the comparison function would work.

```c
const void *dict_lookup(void *key, Dictionary d) {
    struct bstnode *curnode = d->root;
    while (curnode) {
        int result = d->key_compare(key, curnode->item);
        if (result == 0) {
            return curnode->value;
        } else if (result < 0) {
            curnode = curnode->left;
        } else {
            curnode = curnode->right;
        }
    }
    return NULL;
}
```
C generic algorithms

Now that we are comfortable with `void` pointers, we can use C’s built-in `qsort` function.

`qsort` is part of `<stdlib.h>` and can sort an array of any type. This is known as a “generic” algorithm.

`qsort` requires a comparison function (pointer) that is used identically to the comparison approach we described for ADTs.

```c
void qsort(void * arr, int len, size_t size, 
       CompFuncPtr f);
```

The other parameters of `qsort` are an array of any type, the length of the array (number of elements), and the `sizeof` each element.
example: qsort

// see previous definition
int compare_ints (const void *a, const void *b);

int main(void) {

    int a[7] = {8, 6, 7, 5, 3, 0, 9};

    qsort(a, 7, sizeof(int), compare_ints);

    //...
}

C also provides a generic binary search (\texttt{bsearch}) function that searches any sorted array for a key, and either return a pointer to the element if found, or \texttt{NULL} if not found.

\begin{verbatim}
void *bsearch(void *key,
              void *arr,
              int len,
              size_t size,
              CompFuncPtr f);
\end{verbatim}
Goals of this Section

At the end of this section, you should be able to:

• determine an appropriate data structure or ADT for a given design problem

• describe the memory management issues related to using void pointers in ADTs and how void pointer comparison functions can be used with generic ADTs and generic algorithms
Beyond this course

Readings: CP:AMA 2.1, 15.4
Machine code

In Section 05 we briefly discussed compiling: converting source code into machine code so it can be “run” or executed.

Each processor has its own unique machine code language, although some processors are designed to be compatible (e.g., Intel and AMD).

The C language was designed to be easily converted into machine code. This is one reason for C’s popularity.
As an example, the following source code:

```c
int sum_first(int n) {
    int sum = 0;
    for (int i=1; i <= n; ++i) {
        sum += i;
    }
    return sum;
}
```

generates the following machine code (shown as bytes) when it is compiled on an Intel machine.

```
55 89 E5 83 EC 10 C7 45 F8 00 00 00 00 C7 45 FC 01 00 00 00 EB 0A 8B 45 FC 01 45 F8 83 45 FC 01 8B 45 FC 3B 45 08 7E EE 8B 45 F8 C9 C3.
```

How to compile code is covered in CS 241.
When source code is compiled, the identifiers (names) disappear. In the machine code, only *addresses* are used.

The machine code generated for this function

```c
int sum_first(int n) {
    int sum = 0;
    for (int i=1; i <= n; ++i) sum += i;
    return sum;
}
```

is identical to the machine code generated for this function

```c
int fghjkl(int qwerty) {
    int zxcv = 0;
    for (int asdf=1; asdf <= qwerty; ++asdf) zxcv += asdf;
    return zxcv;
}
```
One of the most significant differences between C and Racket is that C is compiled, while Racket is typically interpreted.

An interpreter reads source code and “translates” it into machine code while the program is running. JavaScript and Python are popular languages that are typically interpreted.

Another approach that Racket supports is to compile source code into an intermediate language ("bytecode") that is not machine specific. A virtual machine “translates” the bytecode into machine code while the program is running. Java and C# use this approach, which is faster than interpreting source code.
Compilation

There are three separate steps required to *compile* a C program.

- preprocessing
- compilation
- linking

In modern environments the steps are often *merged* together and simply referred to as “compiling”.

CS 136 Fall 2017 13: Beyond
Preprocessing

In the preprocessing step the preprocessing directives are carried out (Section 03).

For example, the \texttt{#include} directive “cut and pastes” the contents of one file into another file.

The C preprocessor is not \textit{strictly} part of the C language. Other languages can also use C preprocessor and support the \# directives.
Compiling

In the compiling stage, each source code (.c) file is analyzed, checked for errors and then converted into an object code (.o) file. *Object code* is almost complete machine code, except that many of the global identifiers (variable and function names) remain in the code as “placeholders”, as their final addresses are still unknown.

An object file (module.o) includes:

- object code for all functions in module.c
- a list of all identifiers “provided” by module.c
- a list of all identifiers “required” by module.c
Linking

In the linking stage, all of the object files are combined and each global identifier is assigned an address. The final result is a single **executable file**.

The *executable file* contains the *code* section as well as the contents of the *global data* and *read-only data* sections.

The *linker* also ensures that:

- all of the “required” identifiers are “provided” by a module
- there are no duplicate identifiers
- there is an entry point (*i.e.*, a *main* function)
The simplified view of scope (local/module/program) presented in this course is really a combination of:

- **scope**: block scope (local) or file scope (global)

- **storage**: static storage (e.g., global or read-only memory) or automatic storage (stack section)

- **linkage**: internal linkage (when static is used for module scope) or external linkage (the default for a global is program scope) or no linkage (local variables)

See AP:AMA 18.2 for more details.
Command-line (shell) interface

To see compilation at work, we will first explore how to interact with an Operating System (OS) via the command-line.

To start, launch a “Terminal” or similarly named application on your computer. A text-only window will appear with a “prompt” (e.g., $).

You can launch programs directly from the command line.

For example, type date and press return (enter).

We will be providing examples in Linux, but Windows and Mac also have similar command line interfaces. There are numerous online guides available to help you.
Directory navigation

You are most likely familiar with file systems that contain directories (folders) and files organized in a “tree” structure.

At the command line, you are always “working” in one directory. This is also known as your “current” directory or the directory you are “in”.

`pwd` (print working directory) displays your current directory.

```
$ pwd
/u1/username
```

The full directory name is the `path` through the tree starting from the `root` (`/`) followed by each “sub-directory”, separated by `/’s.
When you start the command-line, your current directory is likely your “home directory”.

**cd** (change directory) will return you to your home directory.

```
$ pwd
/somewhere/else
$ cd /u1/username
```

Just like functions, programs can have *parameters* (although they are often *optional*). **cd dirname** will change your current directory.

```
$ pwd
/u1/username
$ cd /somewhere/else
$ pwd
/somewhere/else
```
The argument passed to `cd` can be a full *(absolute)* path (starting with the root `/`) or it can be a path *(relative)* to the current directory. There are also three “special” directory names:

- the current directory
- . the current directory’s parent in the tree (“one level up”)
- ~ your home directory

```
$ cd ~
$ pwd
/u1/username
$ cd ..
$ pwd
/u1
$ cd username <-- relative path
$ pwd
/u1/username
```
The following commands are useful for working with files and navigating at the command-line.

- `ls`: list the contents of the current directory
- `mkdir d`: make a new directory `d`
- `rmdir d`: remove an empty directory `d`
- `cp a b`: make a copy of the file `a` and call it `b`
- `mv a b`: move (rename) file `a` and call it `b`
- `rm a`: delete (remove) the file `a`
- `cat a`: display the contents of the file `a`

A file name may also include the *path* to the file, which can be absolute (from the root) or relative to the current directory.
SSH

SSH (Secure SHell) allows you to use a command-line interface on a remote computer.

For example, to connect to your user account at Waterloo:

$ ssh username@linux.student.cs.uwaterloo.ca

In Windows, a popular (and free) SSH tool is known as PuTTY.
Text Editor

It is often useful to edit a text file in your terminal (or SSH) window, especially when you are connecting to a remote computer.

Emacs and vi (vim) are popular text editors and there is a long-standing friendly rivalry between users over which is better.

One of the easiest text editors for beginners is nano. To start using nano, you only need to remember two commands. To save (output) your file, press (Ctrl-O), and to exit the editor, press (Ctrl-X).
Create hello.c

1) Create a new folder and a new file:

   $ mkdir cs136
   $ cd cs136
   $ nano hello.c

2) Type in the following program:

   #include <stdio.h>

   int main(void) {
       printf("Hello, World!\n");
   }

3) (Ctrl-O) to save (press enter to confirm the file name) and (Ctrl-X) to exit.

   $ ls
   hello.c
We are now ready to compile and execute our program. The most popular C compiler is known as gcc.

$ gcc hello.c
$ ls
  a.out hello.c

gcc’s default executable file name is a.out.

To execute it, we need to specify its path (the current folder .):

$ ./a.out
  Hello, World!

In the Seashell environment we use clang, which is similar to gcc.
To specify the executable file name (instead of `a.out`), a *pair* of parameters is required. The first is `-o` (output) followed by the name.

```
$ gcc hello.c -o hello
$ ./hello
Hello, World!
```

Optional program parameters often start with a hyphen (`-`) and are known as options or “switches”. Options can modify the behaviour of the program (*e.g.*, the option `-v` makes `gcc` verbose and display additional information). Options like `gcc`’s `-o` (output) often require a second parameter.

The `--help` option often displays all of the options available.
gcc can generate object (.o) files by compiling (-c) and not linking.

$ gcc -c module1.c
$ ls
module1.c module1.o

This is really useful when distributing your modules to clients. The client can be provided with just the interface (.h) and the object (.o) file. The implementation details and source file (.c) can remain hidden from the client.

The default behaviour of gcc is to link (or combine) multiple module files (.c and .o) together.

$ gcc module1.o module2.c main.c -o program
Command-line arguments

We have seen how programs can have parameters, but we have not seen how to create a program that accepts parameters.

In Section 03 we described how the \texttt{main} function does not have any parameters, but that is not exactly true. They are optional.

\begin{verbatim}
int main(int argc, char *argv[]) {
    //...
}
\end{verbatim}

\texttt{argv} is an array of strings, and \texttt{argc} is the length of the array.

The length of the array is always at least one, because \texttt{argv[0]} contains the name of the executable program itself. The number of parameters is \((\texttt{argv} - 1)\).
int main(int argc, char *argv[]) {
    int num_param = argc - 1;
    if (num_param == 0) {
        printf("Hello, Stranger!\n");
    } else if (num_param == 1) {
        printf("Hello, %s!\n", argv[1]);
    } else {
        printf("Sorry, too many names.\n");
    }
}

$ gcc hello.c -o hello
$ ./hello
Hello, Stranger!
$ ./hello Alice
Hello, Alice!
$ ./hello Bob
Hello, Bob!
$ ./hello Bob Smith
Sorry, too many names.
Streams

In Section 07 we discussed how programs can interact with the “real world” through input (e.g., `scanf`) and output (e.g., `printf`).

A popular programming abstraction is to represent I/O data as a *stream* of data that moves (or “flows”) from a *source* to a *destination*.

A program can be both a destination (receives input) and a source (produces output).

The source/destination of a stream could be a device, a file, another program or another computer. The stream programming *interface* is the same, regardless of what the source/destination is.
Some programs connect to specific streams, but many programs use the “standard” input & output streams known as stdin & stdout. `scanf` reads from stdin and `printf` outputs to the stdout stream.

The default source for stdin is the keyboard, and the default destination for stdout is the “output window”.

However, we can *redirect* (change) the standard streams to come from any source or go to any destination.
To test I/O, we will create a program that reads characters from stdin and then prints the reverse-case letters to stdout.

```
// swapcase.c
#include <stdio.h>

int main(void) {
    char c;
    while(1) {
        if (scanf("%c", &c) != 1) break;
        if ((c >= 'a')&&(c <= 'z')) {
            c = c - 'a' + 'A';
        } else if ((c >= 'A')&&(c <= 'Z')) {
            c = c - 'A' + 'a';
        }
        printf("%c", c);
    }
}
```
Redirection

To *redirect* output to a file, the `>` symbol is used (*i.e.*, `>` *filename*).

```
$ ./hello > message.txt
$ cat message.txt
Hello, Stranger!
```

Above, the output is stored in a file named `message.txt` instead of displaying the output in the window.

To redirect input *from* a file, use the `<` symbol (*i.e.*, `< filename`).

```
$ ./swapcase < message.txt
hELLO, sTRANGER!
```

You can redirect input and output at the same time.

$ ./swapcase < message.txt > swapped.txt
$ cat swapped.txt
hELLO, sTRANGER!

To redirect directly to or from another program, it is known as piping, and the pipe (|) symbol is used.

$ ./hello Bob | ./swapcase
hELLO, bOB!

$ ./hello DoubleSwap | ./swapcase | ./swapcase
Hello, DoubleSwap!
The Seashell environment

We can now understand all of the tasks that Seashell performs.

- scan the “run” file for `#includes` to determine the required modules, then compile and link all of the modules together
- if “running”: execute while reading `stdio` from seashell
- if “testing”: for each `.in` file, execute the program redirecting from the `.in` file to an output file:
  ```
  $ ./program < mytest.in > mytest.out
  ```

Next, use a comparison program to compare the output files to the `.expect` files and display the differences

  ```
  $ diff mytest.out mytest.expect
  ```
Full C language

We have skipped many C language features, including:

- unions and enumerations
- integer and machine-specific types
- switch
- multi-dimensional arrays
- #define macros and other directives
- bit-wise operators and bit-fields
- advanced file I/O
- several C libraries (e.g., math.h)
CS 246

The successor to this course is:

CS 246: Object-Oriented Software Development

- the C++ language
- object-oriented design and patterns
- tools (bash, svn, gdb, make)
- introduction to software engineering
Feedback welcome

Please send any corrections, feedback or suggestions to improve these course notes to:

Dave Tompkins
dtompkins@uwaterloo.ca

Good Luck on your final exams!
Appendix A

This appendix contains additional content, examples and language syntax details that may not be covered in the lectures.

Some of this content will be covered in tutorials.

You are still responsible for this content, even if it is not presented in the lectures.
A.1: CS 135 Review

In this Section we review some of the CS 135 content revisited in CS 136.

To be concise, we just refer to CS 135 instead of “CS 135 or CS 145 or CS 115 and/or CS 116”.
Functions

Racket *functions* are defined with the `define` special form, which *binds* the function body to the name (identifier).

Racket uses *prefix* notation (instead of *infix* notation).

\[
f(x, y) = (x + y)^2
\]

\[
(\text{define} \ (f \ x \ y)
  (\text{sqr} \ (+ \ x \ y)))
\]
For the above example, we first define a function \( f \), which has two parameters \((x, y)\).

We then apply the function \( f \), which consumes two arguments \((3, 4)\) and produces a single value \(49\).

In this course, we use different terminology: functions are called by passing argument values, and a value is returned.
Constants

*Constants* make your code easier to read and help you to avoid using “magic numbers” in your code.

Constants also give you *flexibility* to make changes in the future.

```scheme
(define ontario-hst .13) ; effective July 1, 2010

(define (add-tax price)
  (* price (add1 ontario-hst)))
```

In this course, constants are often referred to as “variables”. 
Running in Racket

A Racket program is a sequence of definitions and top-level expressions (expressions that are not inside of a definition).

When a program is “run” it starts at the top of the file, binding each definition and evaluating each expression. Racket also “outputs” the final value of each top-level expression.

```
(define (f x) (sqr x)) ; function definition
(define c (f 3)) ; variable definition

; top-level expressions:
(+ 2 3) ; => 5
(f (+ 1 1)) ; => 4
(f c) ; => 81
```
The `cond` special form produces the first “answer” for which the “question” is true. The questions are evaluated in order until a true question or `else` is encountered.
Elementary data types

In addition to numbers, Racket also supports the elementary data types 'symbols and "strings".

- 'symbols are atomic and useful when a small, fixed number of labels are needed. The only practical symbol function is comparison (symbol=?).
- "strings" are compound data and useful when the values are indeterminate or when computation on the contents of the string is required (e.g., sorting).

Strings are composed of characters (e.g., #\c #\h #\a #\r).
Structures

The `define-struct` special form defines a new compound structure with named *fields*.

```
(define-struct my-posn (x y))
; defines posn, posn?, posn-x & posn-y

(define p (make-posn 3 4))
(posn-y p) ; => 4
```
Lists

(define a1 (cons 1 (cons 2 (cons 3 empty))))
(define a2 (list 1 2 3))
(define a3 '(1 2 3))

You should be familiar with list functions introduced in CS 135, including: cons, list, empty, first, rest, list-ref, length, append, and reverse.
Binary search trees

In CS 135 we saw the Binary Search Tree (BST), where each node stores a key and a value

;; A Binary Search Tree (BST) is one of:
;; * empty
;; * a Node

(define-struct node (key val left right))
;; A Node is a (make-node Num Str BST BST)
;; requires: key > every key in left BST
;; key < every key in right BST

We used the list function empty to represent an empty tree, but any sentinel value is fine. A popular alternative is false.
There can be several possible BSTs holding the same set of keys:
(we often only show the keys in a BST diagram)

(define bst1 (make-node 5 "" (make-node 1 "" empty empty)
             (make-node 6 "" empty (make-node 14 "" empty empty))))

(define bst2 (make-node 5 "" (make-node 1 "" empty empty)
             (make-node 14 "" (make-node 6 "" empty empty) empty)))
Remember that the left and right subtrees must also be BSTs and maintain the ordering property.

This is **not** a BST:
BST review

You should be comfortable inserting key/value pairs into a BST.

You should also be comfortable searching for a key in a BST.

You are *not* expected to know how to delete items from a BST.

You are *not* expected to know how to re-balance a BST.
Abstract list functions

In Racket, functions are also first-class values and can be provided as arguments to functions.

The built-in abstract list functions accept functions as parameters. You should be familiar with the abstract list functions \texttt{filter}, \texttt{map}, \texttt{foldr}, \texttt{foldl}, and \texttt{build-list}.

\begin{verbatim}
(define lst '(1 2 3 4 5))

(filter odd? lst) ; => '(1 3 5)
(map sqr lst) ; => '(1 4 9 16 25)
(foldr + 5 lst) ; => 20
\end{verbatim}
Lambda

In Racket, \texttt{lambda} can be used to generate an anonymous function when needed.

\begin{verbatim}
(define lst '(1 2 3 4 5))

(filter (lambda (x) (> x 2)) lst) ; => '(3 4 5)

(build-list 7 (lambda (x) (sqr x)))
; => '(0 1 4 9 16 25 36)
\end{verbatim}
A.2 Full Racket

In this course, we continue to use Racket, but we use the “full Racket” language (#lang racket), not one of the Racket “teaching languages”.

There are some minor differences, which we highlight.

The first line of your Racket (.rkt) files must be:
#lang racket.

In DrRacket, you should also set your language to:
“Determine language from source”.

To save space, we often omit #lang racket in these notes.
Even though you now have the full \#lang racket available to you, you should not “go crazy” and start using every advanced function and language feature available to you.

For your assignments and exams, stick to the language features discussed in class.

If you find a Racket function that’s “too good to be true”, consult the course staff to see if you are allowed to use it on your assignments.

The objective of the assignments is not for you to go “hunting” for obscure built-in functions to do your work for you.
Functions without parameters

Functions can be defined without parameters. In contracts, use `Void` to indicate there are no parameters.

```scheme
(define magic-variable 7)

; magic-function: Void -> Int
(define (magic-function) 42)

(define (use-magic x)
  (* x magic-variable (magic-function)))
```

Parameter-less functions might seem awkward now, but later we see how they can be quite useful.
Booleans

In full Racket, the values #t and #f are used to represent true and false. true and false are constants defined with those values.

Full Racket uses a wider interpretation of “true”:

Any value that is not #f is considered true.

Many computer languages consider zero (0) to be false and any non-zero value is considered true. This is how C behaves.

Because Racket uses #f, it is one of the few languages where zero is considered true.
Logical operators and & or

The special forms `and` and `or` behave a little differently in full Racket:

`and` produces `#f` if any of the arguments are `#f`, `#t` if there are no arguments, otherwise the last argument:

```
(and 5 6 7) ; => 7
```

`or` produces either `#f` or the first non-false argument:

```
(or #f #f 5 6 7) ; => 5
```
Conditionals

\[
\text{(cond} \\
\quad [q1 \ a1] \\
\quad [q2 \ a2] \\
\quad [\text{else} \ a3])
\]

In full Racket, the questions do not have to be Boolean values because any value that is not \texttt{#f} is considered true.

In full Racket, \texttt{cond} does not produce an error if all questions are false: it produces \texttt{#<void>}, which we discuss later.
The `if` special form can be used if there are only two possible answers:

```
(cond
  [q1 a1]
  [else a2])
```

is equivalent to:

```
(if q1 a1 a2)
```

`cond` is preferred over `if` because it is more flexible and easier to follow. We only demonstrate `if` because there is a C equivalent (the `?:` operator).
Structures

Full Racket provides a more compact `struct` syntax for convenience:

- `struct` can be used instead of `define-struct`
- The make-prefix can be omitted (`posn` instead of `make-posn`)

```scheme
(struct posn (x y)) ; defines posn, posn?, posn-x & posn-y
(define p (posn 3 4))
(posn-y p) ; => 4
```

For now, you should include `#:transparent` in your `struct` definitions (this is discussed in Section 02).

```scheme
(struct posn (x y) #:transparent)
```
Full Racket does not enforce that the second argument of `cons` is a list, so it allows you to `cons` any two values (e.g., `(cons 1 2)`), but it’s not a valid list, so don’t do it!
member

In full Racket there is no member? function and member is not a predicate.

(member v lst) produces #f if v does not exist in lst. If v does exist in lst, it produces the tail of lst, starting with the first occurrence v.

(member 2 (list 1 2 3 4)) ; => '(2 3 4)

Recall, that ’(2 3 4) is “true” (not false), so it still behaves the same in most contexts.

You can define your own member? predicate function:

(define (member? v lst) (not (false? (member v lst))))
Implicit local

The `local` special form creates a new local `scope`, so identifiers defined within the local are only available within the `local` body.

In full Racket, you do not need to explicitly use `local`, as there is an *implicit* (“built-in”) `local` in every function body.

```racket
(define (t-area a b c)
  (local
    [(define s (/ (+ a b c) 2))
     (sqrt (* s (- s a) (- s b) (- s c))))])
```

Is equivalent to:

```racket
(define (t-area a b c)
  (define s (/ (+ a b c) 2))
  (sqrt (* s (- s a) (- s b) (- s c))))
```

The variable `s` is implicitly `local`. 
check-expect

The check-expect special form should not be used in full Racket.

In Section 07 we introduce more advanced testing methods.

For now, you can simply use equal? instead of check-expect.

\begin{verbatim}
(define (my-add x y) (+ x y))

;; instead of
;; (check-expect (my-add 1 1) 2)
;; (check-expect (my-add 1 -1) 0)

(equal? (my-add 1 1) 2)
(equal? (my-add 1 -1) 0)
\end{verbatim}
A.3 Racket Modules

There are two Racket special forms that allow us to work with modules: provide and require.

provide is used in a module to specify the identifiers or “bindings” (e.g., function names) that are available to clients.

require is used to identify the module (a file name) that the current file depends upon.

There is also a module special form in Racket and many other module support functions that we do not discuss in this course.
Creating a module

In full Racket, adding a `provide` special form automatically makes a `.rkt` file a module.

Conceptually, the `provide` special form can be seen as the “opposite” of the `local` special form: `local` makes definitions “invisible” to the outside, whereas `provide` makes definitions “visible” to the outside.

Any private functions you wish to hide should not be provided.
In this example, the function hidden-helper is private and not visible outside of the module.
The require special form

When the `require` special form is evaluated, it “runs” all of the code in the required module and makes the `provided` identifiers or “bindings” available.

`require` also “outputs” the final value of any of the top-level expressions in the module.

Modules should only have definitions, not any top-level expressions.
A.4 C Modules

In this appendix we further explore modularization in C modules. In particular, we focus on global scoping and the behaviour of `#include`.

To better understand C’s modularization behaviour, it helps to understand what happens when Seashell “runs” your program.
Running a program

In Racket, `require` temporarily suspends the execution of the main (client) program to run ("process") the module implementation.

C has no mechanism to automatically process module implementations.

To `build` a C program **you must specify all of the files** you wish to **package together**. For example:

- `main.c`
- `module1.c`
- `module2.c`
In the “real world” (outside of Seashell), you must manually specify all of the files you wish to package together to make the program.

Seashell automatically scans your “run file” to determine which module implementations should be packaged together to build your program.

We are presenting a simplified view of linking in C.
For this example, Seashell automatically detects that main.c also requires module1.c and module2.c and packages them all together:

```c
// main.c: set as the "run file"

#include "module1.h"
#include "module2.h"

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

Seashell can also use binary (.o) implementation files instead of source code (.c) implementation files.
In other words, when you
#include "module1.h"

it is **Seashell** that adds the implementation **module1.c** to the program, not C.

Remember, the **include preprocessor directive** simply “inserts” the contents of one file into another and does nothing else.
So while it may appear that placing a function declaration in an *interface* (.h file) “provides” the function, it is not actually the case. The function is provided when Seashell packages the two files together.

This works because, by default, all functions have *program scope*, and are available to all other files once they are packaged together.
If two different modules both have a function (or a global variable) with the same name, C will encounter an error when trying to package them together. This is known as a “collision”.

There are two key strategies to help avoid collisions:

- For *provided* functions (and global variables), use meaningful identifiers that reflect the name of the module. This will reduce the chance of a collision.

- For functions (and global variables) that are *not provided*, use the `static` keyword to give them *module scope*.
A.5 Memory

In this section we briefly discuss number representations and computer memory.
Decimal notation

In *decimal* representation (also known as *base 10*) there are ten distinct digits (0123456789).

When we write the number 7305 in base 10, we interpret it as

\[ 7305 = 7 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 5 \times 10^0. \]

4 decimal digits can represent \(10^4\) (10,000) different possible values (*i.e.*, 0 ... 9999).

The reason base 10 is popular is because we have 10 fingers.
Binary notation

In *binary* representation (also known as *base 2*) there are two distinct digits (01).

When we write the number 1011010 in binary, we interpret it as

\[1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0\]

\[= 64 + 16 + 8 + 2 = 90\] (in base 10).

4 binary digits, can represent \(2^4\) (16) different possible values (0 ... 1111) or (0 ... 15) in base 10.

In CS 251 / CS 230 you will learn more about binary notation.
Hexadecimal notation

In \textit{hexadecimal (hex)} representation (also known as \textit{base 16}) there are \textbf{sixteen} distinct digits (0123456789ABCDEF).

When we write the number 2A9F in hex, we interpret it as

\[2 \times 16^3 + 10 \times 16^2 + 9 \times 16^1 + 15 \times 16^0\]

\[= 8192 + 2560 + 144 + 15 = 10911\] (in base 10).

The reason \textit{hex} is so popular is because it is easy to switch between binary and hex representation. A single hex digit corresponds to exactly 4 bits.
## Conversion table

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</th>
<th>Hex</th>
<th>Dec</th>
<th>Bin</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>
Binary/Hex conversion

To distinguish between 1234 (decimal) and 1234 (hex), we prefix hex numbers with “0x” (0x1234).

Here are some simple hex / binary conversions:

\[
\begin{align*}
0x1234 & = 0001 0010 0011 0100 \\
0x2A9F & = 0010 1010 1001 1111 \\
0xFACE & = 1111 1010 1100 1110
\end{align*}
\]
In C, a number with the prefix 0x is interpreted as a hex number:

```c
int num = 0x2A9F; //same as num = 10911
```

A number prefixed with a zero 0 is interpreted as an octal numbers (base 8).

```c
int num = 025237; //same as num = 10911
```

This has been the source of some confusion.
Memory Capacity

To have a better understanding of the C memory model, we provide a brief introduction to working with bits and bytes.

You are probably aware that internally, computers work with bits.

A bit of storage (in the memory of a computer) is in one of two states: either 0 or 1.

A traditional light switch can be thought of as a bit of storage.
Early in computing it became obvious that working with individual bits was tedious and inefficient. It was decided to work with 8 bits of storage at a time, and a group of 8 bits became known as a byte.

Each byte in memory is in one of 256 possible states.

With today’s computers, we can have large memory capacities:

- 1 KB = 1 kilobyte = 1024 \( (2^{10}) \) bytes
- 1 MB = 1 megabyte = 1024 KB = 1,048,576 \( (2^{20}) \) bytes
- 1 GB = 1 gigabyte = 1024 MB = 1,073,741,824 \( (2^{30}) \) bytes
* The size of a kilobyte can be 1000 bytes or 1024 bytes, depending on the context. Similarly, a megabyte can be $10^6$ or $2^{20}$ bytes, etc.

Manufacturers often use the measurement that makes their product appear better. For example, a terabyte (TB) drive is almost always $10^{12}$ bytes instead of $2^{40}$.

To avoid confusion in scientific use, a standard was established to use KB for 1000 bytes and KiB for 1024 bytes, etc.

In general use, KB is still commonly used to represent both.
Primary Memory

Modern computers have primary memory in addition to secondary storage (hard drives, solid state drives, flash drives, DVDs, etc.).

The characteristics of primary memory and secondary storage devices vary, but in general:

**Primary Memory:**

- very fast (nanoseconds)
- medium capacity (≈ GB)
- high cost ($) per byte
- harder to remove
- erased on power down

**Secondary Storage:**

- (≈20x–1000x) slower
- large capacity (≈ TB)
- low cost ($) per byte
- removable or portable
- persistent after power down
In practice, programs can only “run” in primary memory.

When you “launch” a program, it is copied from secondary storage to primary memory before it is “run”.

In this course, we are always referring to primary memory, which is also known as *Random Access Memory (RAM)*. With RAM you can access any individual byte directly and you can access the memory in any order you desire (randomly).
Traditional secondary storage devices (hard drives) are faster if you access data *sequentially* (not randomly).

Primary memory became known as **RAM** to distinguish it from sequential access devices.

The term “RAM” is becoming outdated, as solid state drives and flash drives use random access. Also, modern RAM can be faster when accessed sequentially (in “bursts”).

Regardless, when you encounter the term “RAM”, you should interpret it as “primary memory”.

A.6 I/O & Testing

In this section we briefly discuss additional I/O and testing methods
Advanced Tracing

You can use different **tracing levels** to indicate how much detail you want in your tracing output. Once you have debugged your code, you can simply set the level to zero and turn off all tracing.

```c
int TRACELEVEL = 2;

int main(void) {
    if (TRACELEVEL >= 3) printf("starting main\n");
    // ...

    for (int i=0; i < 10; ++i) {
        sum += i;
        if (TRACELEVEL >= 2) printf("i = %d, sum = %d\n", i, sum);
    }
    //...
}
```
Input

The Racket read function attempts to read (or “get”) a value from the keyboard. If there is no value available, read pauses the program and waits until there is.

(define my-value (read))

read may produce a special value (#<eof>) to indicate that the End Of File (EOF) has been reached.

EOF is a special value to indicate that there is no more input.
The `read` function is quite complicated, so we present a simplified overview that is sufficient for our needs.

`read` interprets the input as if a single quote `’` has been inserted before each “value” (again, not really but close enough).

If your value begins with an open parenthesis `(`, Racket reads until a corresponding closing parenthesis `)` is reached, interpreting the input as one value (a list).

Text is interpreted as symbols, not a string (unless it starts with a double-quote `"`). The `symbol->string` function is often quite handy when working with `read`. 
example: read

(define (read-to-eof)
  (define r (read))
  (printf "~v\n" r)
  (cond [(not (eof-object? r)) (read-to-eof)]))

1
two
"three"
(1 two "three")
Ctrl-D

1
'two
"three"
'(1 two "three")
#<eof>
User interaction

With the combination of input & output, we can make interactive programs that change their behaviour based on the input.

(define (get-name)
    (printf "Please enter your first name:\n")
    (define name (read))
    (printf "Welcome, to our program, ~a!\n" name))
(define (madlib)
  (printf "Let’s play Mad Libs! Enter 4 words :\n")
  (printf "a Verb, Noun, Adverb & Adjective :\n")
  (define verb (read))
  (define noun (read))
  (define adverb (read))
  (define adj (read))
  (printf "The two ~as were too ~a to ~a ~a.\n"
          noun adj verb adverb))

(madlib)
Interactive testing

In DrRacket, the *interactions window* was quite a useful tool for debugging our programs.

In Seashell, we can create **interactive I/O testing clients**.

Consider an example with a simple arithmetic module.

```c
// addsqr.h

// sqr(x) returns x*x
int sqr(int x);

// add(x,y) returns x+y
int add(int x, int y);
```
example: interactive I/O testing in C

#include "addsqr.h"

int main(void) {
    char func;
    int x, y;
    while (1) {
        if (scanf(" %c", &func) != 1) break;
        if (func == 'x') break;
        if (func == 'a') {
            scanf("%d", &x); // you should check scanf return values
            scanf("%d", &y);
            printf("add %d %d = %d\n", x, y, add(x,y));
        } else if (func == 's') {
            scanf("%d", &x);
            printf("sqr %d = %d\n", x, sqr(x));
        }
    }
}
With this *interactive* i/o testing module, tests are entered via the keyboard.

**Input:**

- a 3 4
- a -1 0
- a 999 -1000
- s 5
- s -5
- s 0
- x

**Output:**

- add 3 4 = 7
- add -1 0 = -1
- add 999 -1000 = -1
- sqr 5 = 25
- sqr -5 = 25
- sqr 0 = 0
One big advantage of this i/o interactive testing approach is that we can experiment with our module without having to program (code) each possible test.

It’s also possible that someone could test the code without even knowing how to program.

A disadvantage of this approach is that it can become quite tedious to rely on human input at the keyboard.

Fortunately, the Seashell environment has support to *automate* interactive testing with `.in` and `.expect` files.
A.7 Efficiency

In this section we provide more efficiency examples.
Examples: recurrence relations

For simplicity and convenience (and to avoid any `equal?` issues) we use lists of integers in these examples.

\[
\begin{align*}
T(n) &= O(1) + T(n - 1) = O(n) \\
T(n) &= O(n) + T(n - 1) = O(n^2)
\end{align*}
\]

```
(define (member1 e lon)
  (cond [(empty? lon) #f]
    [(= e (first lon)) #t]
    [else (member1 e (rest lon))]))
```

```
(define (member2 e lon)
  (cond [(zero? (length lon)) #f]
    [(= e (first lon)) #t]
    [else (member2 e (rest lon))]))
```
(define (has-duplicates? lon)
  (cond [(empty? lon) #f]
        [(member (first lon) (rest lon)) #t]
        [else (has-duplicates? (rest lon))])))

\[T(n) = O(n) + T(n - 1) = O(n^2)\]

(define (has-same-adjacent? lon) ;; O(n)
  (cond [(or (empty? lon) (empty? (rest lon))) #f]
        [(= (first lon) (second lon)) #t]
        [else (has-same-adjacent? (rest lon))])))

(define (faster-has-duplicates? lon)
  (has-same-adjacent? (sort lon <)))

\[T(n) = O(n \log n) + O(n) = O(n \log n)\]
(define (find-max lon)
  (cond
    [(empty? (rest lon)) (first lon)]
    [(> (first lon) (find-max (rest lon))) (first lon)]
    [else (find-max (rest lon))])))

\[ T(n) = O(1) + 2T(n - 1) = O(2^n) \]

(define (fast-max lon)
  (cond
    [(empty? (rest lon)) (first lon)]
    [else (max (first lon) (fast-max (rest lon)))]))

\[ T(n) = O(1) + T(n - 1) = O(n) \]

(define (clean-max lon)
  (apply max lon))

\[ T(n) = O(n) \]
A.8 Linked lists

In Section 11 we present a linked list with a *wrapper strategy*. We also intentionally avoid following a very “functional” approach to help prevent mixing paradigms.

In this Section we present additional examples, still using the same `llnode` structure:

```c
struct llnode {
    int item;
    struct llnode *next;
};
```
Functional approach

We can create helper functions to create a “functional” atmosphere:

```c
int first(struct llnode *lst) {
    assert (lst != NULL);
    return lst->item;
}

struct llnode *rest(struct llnode *lst) {
    assert (lst != NULL);
    return lst->next;
}

struct llnode *empty(void) {
    return NULL;
}

bool is_empty(struct llnode *lst) {
    return lst == empty();
}
```
At the heart of Racket’s functional list approach is the cons function.

This C cons returns a new node that links to the rest.

```c
struct llnode *cons(int f, struct llnode *r) {
    struct llnode *new = malloc(sizeof(struct llnode));
    new->item = f;
    new->next = r;
    return new;
}
```

This is very similar to how Racket’s cons is implemented.
We can use our new C `cons` function the same way we use the Racket `cons` function.

```c
struct llnode * my_list =
    cons(10, cons(3, cons(5, cons(7, empty()))));
```

![Diagram](10 -> 3 -> 5 -> 7)

We can also use `cons` in different ways (e.g., with mutation).

```c
struct llnode * my_list = empty();
my_list = cons(7, my_list);
my_list = cons(5, my_list);
my_list = cons(3, my_list);
my_list = cons(10, my_list);
```
Using this approach, the code to make a duplicate of a list is very straightforward.

```c
struct llnode *list_dup(struct llnode *lst) {
    if (is_empty(lst)) return empty();
    return cons(first(lst), list_dup(rest(lst)));
}
```

In Section 11 we present a `list_dup` function that is more awkward.
A C function written with `cons` will return a **new list** (a “functional” approach).

Consider the following “square list” function:

```c
struct llnode *sqr_list(struct llnode *lst) {
    if (is_empty(lst)) return empty();
    return cons(first(lst) * first(lst),
               sqr_list(rest(lst)));
}
```
To correctly use the `sqr_list` function, the result should be stored in a separate variable.

```c
struct llnode *a = cons(10, cons(3, cons(5, cons(7, empty()))));
struct llnode *b = sqr_list(a);
```

Unfortunately, if the function is misunderstood or used **incorrectly**, it can create a **memory leak**.
The following two statements each create a memory leak.

\[
a = \text{sqr\_list}(a); \quad // \text{original list is lost}
\]

```
\textbf{a:}
\begin{array}{cccccc}
10 & | & 3 & | & 5 & | & 7 \\
\_ & | & \_ & | & \_ & | & \_ \\
100 & | & 9 & | & 25 & | & 49 \\
\_ & | & \_ & | & \_ & | & \_
\end{array}
```

\[
\text{sqr\_list}(a); \quad // \text{new list is lost}
\]

```
\textbf{a:}
\begin{array}{cccccc}
10 & | & 3 & | & 5 & | & 7 \\
\_ & | & \_ & | & \_ & | & \_ \\
100 & | & 9 & | & 25 & | & 49 \\
\_ & | & \_ & | & \_ & | & \_
\end{array}
```

The best strategy to avoid a memory leak is to provide a clear contract.
The `cons` function can cause a similar problem:

```c
struct llnode *a = cons(7, empty());
cons(5, a); // memory leak
```

This is not a concern in Racket because it uses garbage collection and automatically frees memory.
We also have to worry about **node sharing**.

```c
struct llnode *a = cons(4, cons(5, cons(6, empty())));
struct llnode *b = cons(3, a);
struct llnode *c = cons(1, cons(2, b));
```
To properly use `cons` in an imperative environment, it helps to imagine that the second parameter (the “rest”) is to be considered “inaccessible” or an “invalid” list.

Only the returned value should be used as a list.

```c
// cons(f, r) produces a new list with f added to the front of the list r
// effects: r is no longer a valid list pointer

struct llnode * cons(int f, struct llnode * r);
```
Working without a wrapper

In Section 11 we wrote an `add_front` function that worked with the wrapper structure.

Without the wrapper, we can simply use `cons` to add to the front of the list.

However, we have to ensure that the caller uses `cons` properly (storing the result and considering the second parameter invalid).
Instead, we want to write an `add_front` function where we pass the list as a parameter.

For example, we would like to have the following code sequence.

```c
struct llnode *lst = NULL;

add_front(7, lst);  // won’t work
add_front(5, lst);  // won’t work
```

Remember, to have a function change a variable (i.e., `lst`), we need to pass a `pointer` to the variable.

```c
add_front(7, &lst);
add_front(5, &lst);
```

Since `lst` is already a pointer, we need to pass a `pointer to a pointer`. 
void add_front(int n, struct llnode **ptr_front) {
    struct llnode *new = malloc(sizeof(struct llnode));
    new->item = n;
    new->next = *ptr_front;
    *ptr_front = new;
}

Example:

struct llnode *lst = NULL;
add_front(7, &lst);
add_front(5, &lst);
add_front(3, &lst);
add_front(10, &lst);

```
10 3 5 7
```
This *pointer-to-a-pointer* approach is also necessary if the first item may be removed.

```c
int remove_from_front(struct llnode **ptr_front) {
    struct llnode *front = *ptr_front;
    int retval = front->item;
    *ptr_front = front->next;
    free(front);
    return retval;
}
```

Instead of *returning* nothing (*void*), it is more useful to *return* the value of the item being removed.

Avoiding this *pointer-to-a-pointer* is one of the advantages of using a wrapper strategy.
An alternative “destructive” approach uses a Racket-like programming interface (functions produce new lists), but each list passed to a function may be destroyed (freed).

```c
struct llnode *insert(int n, struct llnode *slst) {
    if (is_empty(slst)) {
        return cons(n, empty());
    } else if (n <= first(slst)) {
        return cons(n, slst);
    } else {
        int f_backup = first(slst);
        struct llnode *r_backup = rest(slst);
        free(slst);
        return cons(f_backup, insert(n, r_backup));
    }
}
```

This approach has been taught in previous offerings of CS 136.