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 (Spring 2015)

Instructors: Ahmed Hajyasien, Dan Holtby

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!

Course Materials

Textbooks:

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

Each assignment may have different instructions and requirements.

Make sure you **read the instructions carefully**.

All work must be submitted to the Marmoset submission system:

`http://marmoset.student.cs.uwaterloo.ca/`

- For correctness marks, marmoset takes the best result from all of your submissions (there is never any harm in resubmitting)
- Marmoset provides simple feedback to ensure your program is “runnable” and **should not be used to test your code**

A0 is due this Friday: it does not count toward your grade, **but must be completed on time**.

While other courses may allow or encourage working together in groups, CS 136 assignments must be done **individually**.

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

An assignment may have “warm-up” questions that can be openly discussed.

Assignments: second chances

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

- Second chances are granted automatically by a fixed decision rule 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)

Getting Help

In addition to *office hours*, *tutorials* and *lab hours* we use an online discussion forum (**piazza**).

Instructors, ISAs and your fellow students monitor piazza and answer questions.

Post *clarification questions* to help understand assignments, but **do not** discuss assignment strategies, and **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.

Course announcements are made on piazza and are considered **mandatory reading**.

Languages

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

While significant time is required to learn 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.

Software

We use our own customized “`Seashell`” environment, which has a browser interface.

`Seashell` works with both C and Racket, provides verbose error messages, and helps to facilitate your testing.

Our testing environment (Marmoset) uses the same environment as `Seashell` to run your program. (If it does not work with `Seashell`, it will not work with our tests).

See the website for how to use `Seashell`.

Design Recipe

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

There are two main goals with the design recipe:

- 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 and 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, examples, or tests.

Throughout this course, we will extend contracts to include *effects* and *time* (efficiency).

You are expected to test your own code. Advanced testing strategies are discussed in Section 07.

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	$\Rightarrow$	CS 136	$\Rightarrow$	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

Modularization

Readings: CP:AMA 19.1

Modularization

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

For large 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**.

In computer science, a ***module*** is collection of functions that share a common aspect or purpose.

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 a “AA battery”.

Motivation

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

Re-usability: A good module can be re-used by many programs. 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 changed. We can even replace an entire module with a more efficient or more robust implementation.

Abstraction: To use a module, we need to understand **what** functionality it provides, but we do not need to understand **how** it is implemented. In other words, we only need 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)*.

example: tax module

Consider a program that allows Waterloo students to order their school supplies online and have them delivered.

Such a large program might be broken into modules, such as:

- looking up course requirements,
- keeping track of inventories,
- creating a virtual shopping cart,
- processing payments such as credit cards, etc...

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

Consider an “Ontario tax module” that can identify the proper tax rate for each item (*e.g.,* textbooks are 5%, but empty workbooks are 13%). The advantages of creating this module are:

- **re-usability:** This module could be easily reused for other Ontario-based shopping applications.
- **maintainability:** When the tax rates and rules change, we only have to modify one module.
- **abstraction:** To use this module, you do not need to understand how the taxes are calculated. *It does not matter* because the details have been *abstracted* away.

example: fun numbers

Consider that some integers are more "fun" than others, and we want to create a Racket module that "provides" a `fun?` function.

```
;; fun.rkt

(provide fun?)  ; <---- this is new!

(define lofn '(-3 7 42 136 1337 4010 8675309))

;; (fun? n) determines if n is a fun integer
;; fun?: Int -> Bool
(define (fun? n)
  (not (false? (member n lofn))))
```

The `provide` special form above makes the `fun?` function available to other programs.

The `require` special form allows us to use the `fun?` function in a different program.

```
;; different-program.rkt

(require "fun.rkt")  ; <---- this is new!

(fun? 7)     ; => #t
(fun? -7)    ; => #f
```

This programmer only requires an *"abstract"* understanding of fun integers, and does not need to understand what integers are fun, or how they are determined.

When new numbers become fun (or become less fun), this program does not need to change. Only the fun module is changed (this increases maintainability).

Modules in Racket

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 in the module.

`require` is used to identify a module that the current module (program) 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, each `.rkt` file we create is automatically a module. However, none of the functions are accessible outside of the module unless they are `provide`d.

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.

example: provide

```
(provide function-a function-b)

(define (function-a p) ...)

(define (function-b p1 p2) ...)

(define (hidden-helper n) ...)  ; not provided
```

In this example, the function `hidden-helper` is private and not visible outside of the module.

Scope

`provide` extends the **scope** of a global identifier beyond the module, giving us three “levels” of scope.

- **local:** only visible inside of the “block” or `local` region
- **module:** only visible inside of the module it is defined in
- **program:** visible outside of the module (*i.e.*, `provide`d)

We continue to use the term **“global”** to describe top-level identifiers that have either **program *or* module** scope.

Because `local`s can be “nested” there can be multiple “levels” of local scope, but in this slide we are generalizing.

example: scope

```
(provide p)

(define p 1)              ; program scope (provided)

(define m 2)              ; module scope (not provided)

(define (f x)
  (define b 3)            ; local scope (within a function)
  ...)
```

In addition, the function `f` has module scope, and its parameter `x` has local scope.

Other courses may use different scope terminologies.

The require special form

When the `require` special form is evaluated, it "runs" all of the code in the required module and makes the `provide`d identifiers or "bindings" available.

`require` also "outputs" the final value of any of the top-level expressions in the module.

In a module you are providing to clients, you should only have definitions, not any top-level expressions.

Module interface

The module ***interface*** is the list of the functions that the module provides, including the contract and purpose for each function. Interfaces may also include structure definitions and variables.

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

It is helpful to think of a *“client”* who is using a module.

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

Interface documentation

The module interface includes **documentation for the client**: it should provide the client with all of the information necessary to use your module. The client does not need to know how your module is implemented.

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.

Racket module interfaces

For Racket modules, the interface should appear at the top of the file above the implementation (function definitions).

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

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

sum.rkt

```
;; A module for summing numbers [description of module]

(provide sum-first sum-squares)  ;; [list of functions]

;; (sum-first n) sums the integers 1..n
;; sum-first: Int -> Int
;; requires: n >= 1

;; (sum-squares n) ...

;;;;;;;;;; IMPLEMENTATION ;;;;;;;;;;

;; see interface above  [no further info required]
(define (sum-first n) ...)

;; see interface above  [no further info required]
(define (sum-squares n) ...)

;; [purpose & contract for private helper function]
(define (private-helper p) ...)
```

Testing modules

Without `check-expect` or top-level expressions, how can we **test** a module?

For each module you design, you should also create a **testing module** that ensures the `provide`d functions are correct.

test-sum.rkt

```
;; this is a simple testing module for sum.rkt

(require "sum.rkt")

; Each of the following should produce #t
(equal? (sum-first 1) 1)
(equal? (sum-first 2) 3)
(equal? (sum-first 3) 6)
(equal? (sum-first 10) 55)
(equal? (sum-first 99) 4950)
```

In Section 07 we discuss more advanced testing strategies.

There may be “white box” tests that cannot be tested from outside of your module. These may include implementation-specific tests or tests of private functions.

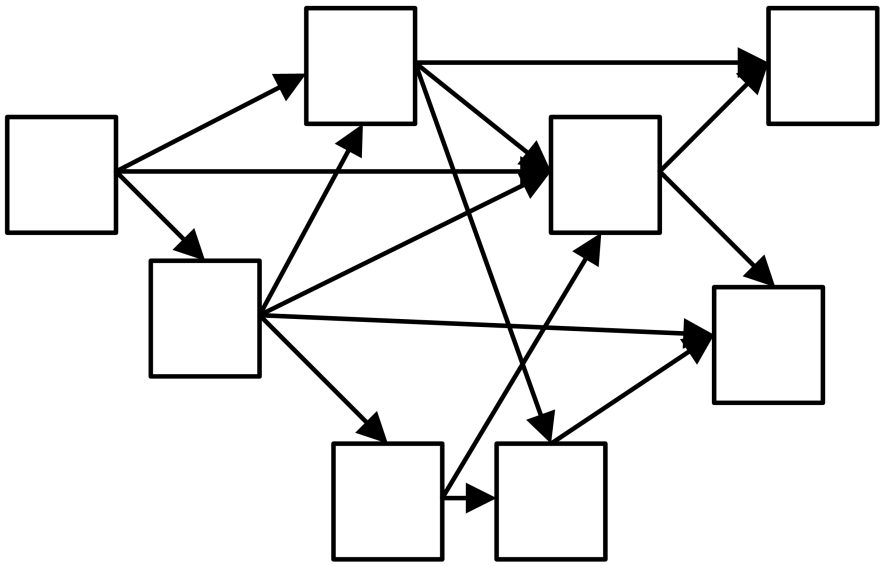
In these circumstances, you can `provide` a `test-module-name` predicate function that produces `#t` if the tests are successful.

Cohesion and coupling

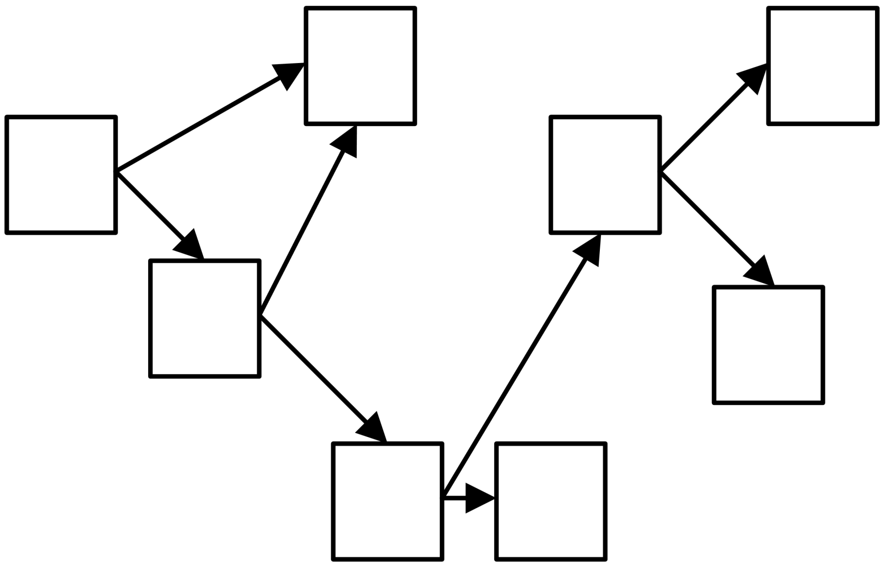
When designing module interfaces, we want to achieve ***high cohesion*** and ***low coupling***.

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. If module A depends on the interface for module B and B also depends on the interface for A, that is high coupling and poor design.



High coupling



Low coupling

Interface design is especially important when working with large projects. Identifying the appropriate modules and interfaces can be quite challenging.

In this course there is very little interface design.

example: tax module

When designing our Ontario tax module, we should consider:

- **high cohesion:** All of the interface functions should be related to determining tax rates. Including a function for calculating the area of a triangle would be *low* cohesion.

- **low coupling:** The interface of a module should not depend on the "parent" program. Having the tax module call functions from the inventory maintenance module would be *high* coupling.

Interface vs. implementation

Earlier, we made 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: information hiding (account)

Consider the following module for storing a username and a password in an “account”. The only interface functions are for creating the account and verifying the password.

```
(provide create-account correct-password?)

(struct account (username password))

;create-account: Str Str -> Account
(define (create-account u p)
  (account u p))

;correct-password?: Account Str -> Bool
(define (correct-password? a p)
  (equal? (account-password a) p))
```

The definition `(struct account (username password))` automatically generates the functions `account`, `account?`, `account-username` and `account-password`.

However, the account module **did not** **`provide`** any of those functions. As a result, the client is unable to “peek” inside of an account to view the password. The client is also unable to “forge” an account and can only create one with the provided `create-account` interface function.

This is an example of achieving **security** through information hiding.

To improve security, full Racket structures are ***opaque*** by default.

Adding `#:transparent` to a `struct` definition makes the structure "not opaque" or *transparent*.

```
(struct account (username password) #:transparent)
```

The only significant difference is that the contents of a transparent structure can be viewed in output (*e.g.,* with `printf`). This is useful while debugging and testing, but not very secure.

Both opaque and transparent structures require field selector functions (*e.g.,* `account-username`) to obtain field values.

To make the username of an account visible to the client, we could `provide` the `account-username` selector function.

However, this exposes the field name "`username`" to the client, which is an implementation detail. Including that detail in the interface is poor information hiding and restricts **flexibility**.

Instead, a "wrapper" function can be provided.

```
;get-username: Account -> String
(define (get-username a)
  (account-username a))
```

For this small Racket example, the change is very subtle. In other languages this difference is more significant.

To illustrate flexibility, we can change the **implementation** without changing the **interface**. There is no difference to the client.

```
(struct account (lst)) ; account now only has 1 field (a list):
                       ;   (list username password)

;; create-account: Str Str -> Account
(define (create-account u p)
  (account (list u p)))

;; correct-password?: Account Str -> Bool
(define (correct-password? a p)
  (equal? (second (account-lst a)) p))

;; get-username: Account -> Str
(define (get-username a)
  (first (account-lst a)))
```

If `account` was `provide`d instead of the `create-account` wrapper, this change would not have been possible.

Data structures & abstract data types

In the previous `account` example, we demonstrated two **implementations** with two different ***data structures***:

- a `struct` with two fields
- a *list* with two elements (wrapped within another `struct`)

When working with a *data structure* we know precisely how the data is “structured” or “arranged”.

However, the client doesn’t need to know how the data is structured. The client only requires an *abstract* understanding that an `account` stores data (a username and a password).

Formally, an ***Abstract Data Type (ADT)*** is a mathematical model for storing and accessing data through ***operations***. As mathematical models they transcend any specific computer language or implementation.

However, in practice (and in this course) ADTs are **implemented** as data storage **modules** that only allow access to the data through interface functions (ADT *operations*). The underlying data structure and implementation of an ADT is **hidden** from the client (which provides *flexibility* and *security*).

The `account` module is an implementation of an “account ADT” because it stores data that can only be accessed through the module interface functions (operations).

Account is not a “typical” ADT because it stores a fixed amount of data and it has limited use.

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)

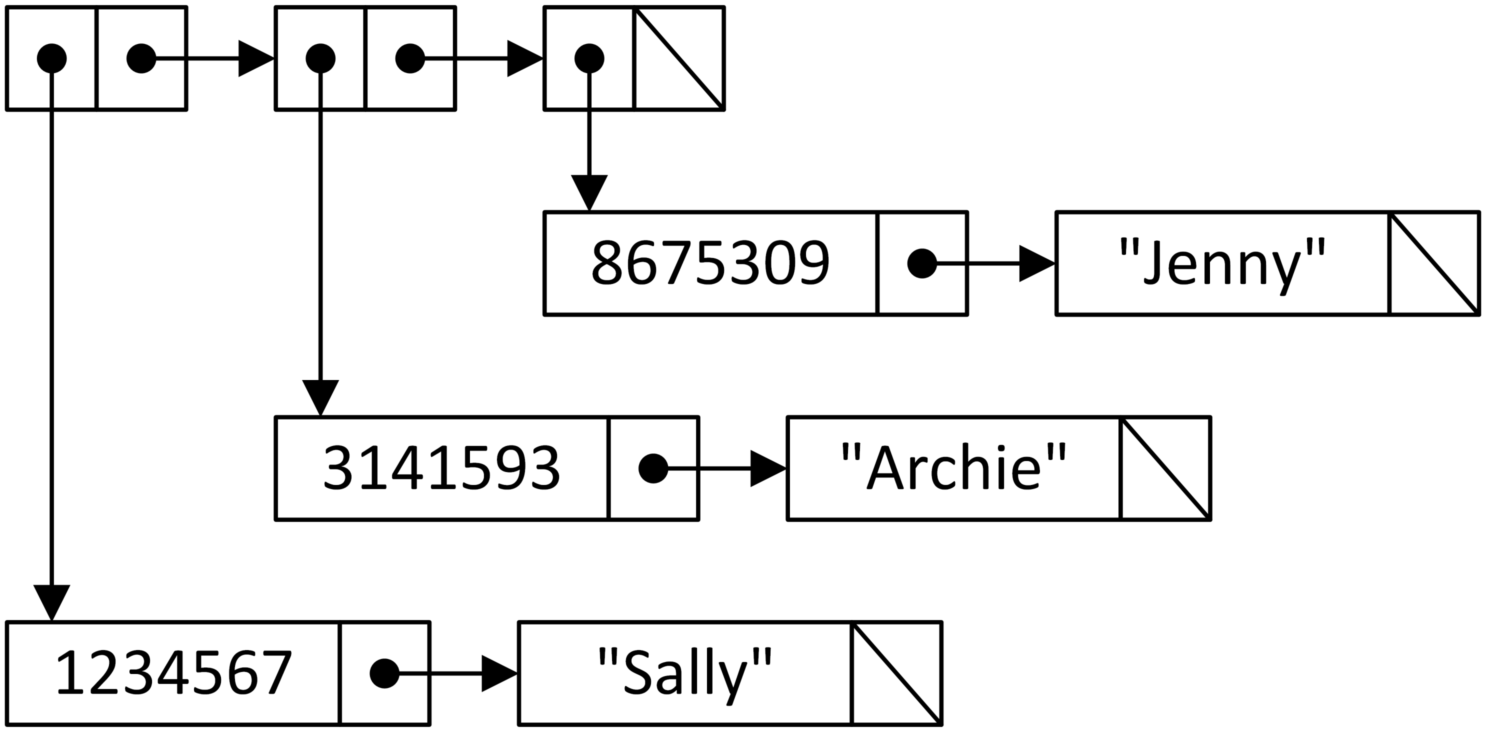
Recall that a ***Dictionary*** stores ***key*** and ***value*** pairs. For a given key, we can `lookup` the corresponding value in the dictionary (lookup is an *operation*).

example: student numbers

key (student number)	**value (student name)**
`1234567`	`"Sally"`
`3141593`	`"Archie"`
`8675309`	`"Jenny"`

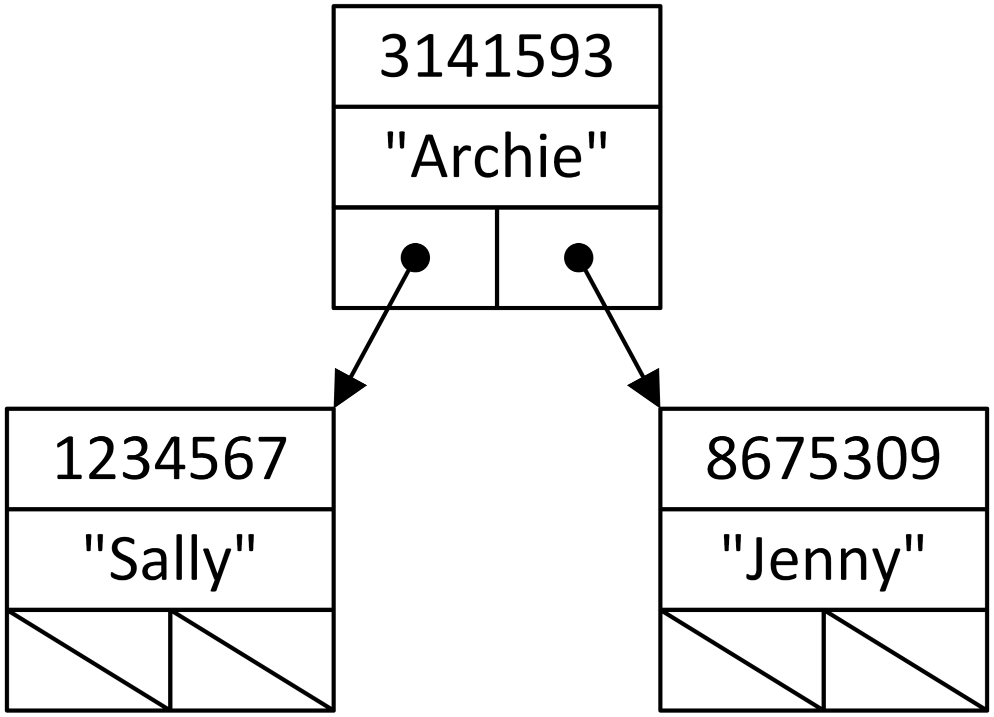
We can implement a dictionary with an ***association list*** *data structure* (a list of key/value pairs with each pair stored as a list of two elements).

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



Alternatively, we can implement a dictionary with a ***Binary Search Tree (BST)*** *data structure*.

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



BSTs are briefly reviewed in Section A.

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 a design decision.

We can **use** a dictionary ADT as a client without knowing *how* it is implemented. We are only interested in performing the dictionary operations: `lookup`, `insert` and `remove`.

Recall that hiding the data structure from the client provides us:

- **flexibility:** we can change data structures without the client knowing (perhaps with a more efficient implementation)
- **security:** the client cannot “accidentally” corrupt the data (for example, violating the ordering property of a BST)

Not all modules are designed to store data, and not all data storage modules are ADTs.

While we have demonstrated the advantages of information hiding, it is not always necessary. Transparency may be more important than security or flexibility.

An important aspect of interface design is deciding upon the level of information hiding, and determining if it is appropriate to expose any implementation details in the interface.

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.

- use `provide` and `require` to implement modules in Racket
- design an ADT module in Racket
- produce good interface documentation, including the new documentation changes introduced

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, 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 OSX, 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 (`const`ant).

The `int` keyword **declares** that `my_number` is an `int`eger.

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 `const`ants.

C identifiers ("names") are more limited than in Racket. They must start with a letter, and can only have 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 an underscore (`_name`) but their use is restricted. It's best to avoid them.

Typing

Racket and C handle types very differently.

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

```
; 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.

```
// stype is always a const int
const int stype = 42;
```

C Types

For now, we **only work with `int`egers** in C.

More types are introduced as the course progresses.

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:
(define six (+ 3 3))
(define seven (+ 1 (* 3 2)))
(define eight (* (+ 1 3) 2))
```

```
// C Expressions:
const int six = 3 + 3;
const int seven = 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 * 5 / 2` becomes `(4 * 5) / 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) any intermediate values, and behaves the same as the Racket `quotient` function.

```
const int a = (4 * 5) / 2;  // 10
const int b = 4 * (5 / 2);  // 8 !!
const int c = -5 / 2;       // -2 !!
```

Remember, use parentheses to clarify the order of operations.

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 (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
; Racket function:
; my-sqr: Int -> Int
(define (my-sqr x)
  (* x x))
```

```c
// C function:
int my_sqr(const int x) {
  return x * x;
}
```

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 a `const int` parameter `x`.

In Racket, the contract types are only documentation. It is possible to violate the contract, which may cause a “type” runtime error.

```
;; 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:
my_sqr("hello")    // => will not run !!!
```

In the previous example, the parameter was a `const int`, but the *return type* was just an `int`.

```
int my_sqr(const int x) {
  return x * x;
}
```

Most versions of C will allow a `const int` return type:

```
const int my_sqr(const int x) { ... }
```

However, the `const` in the return type is **ignored**, so we will not use it.

We will discuss using `const` parameters and return types in more detail later.

```
int my_add(const int x, const 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.

Function documentation

```
; (my-sqr x) squares x
; my-sqr: Int -> Int

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

```
// my_sqr(x) squares x

int my_sqr(const int x) {
  return x * x;
}
```

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

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

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

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 `!=`.

The value of `(3 == 3)` is `1` (“true”).

The value of `(2 == 3)` is `0` (“false”).

The value of `(2 != 3)` is `1` (“true”).

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 `||`.

The value of `!(3 == 3)` is `0`.

The value of `(3 == 3) && (2 == 3)` is `0`.

The value of `(3 == 3) && !(2 == 3)` is `1`.

The value of `(3 == 3) || (2 == 3)` is `1`.

The value of `(2 && 3 || 0)` is `1`.

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.

The value of `(2 < 3)` is `1`.

The value of `(2 >= 3)` is `0`.

The value of `!(a < b)` is equivalent to `(a >= 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` is equivalent to `1 == (3 > 0)`, but it could easily confuse some readers.

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* and an `int` return type to indicate if the program is successful (zero) or not (non-zero).

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

`main` is "special" because the return value is *optional*. If there is no `return` in `main`, zero is automatically returned.

In this course, `main` is also special because it requires no documentation (*e.g.*, a purpose statement).

* `main` has *optional* parameters discussed in Section 13.

Hello, World

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

```
// My first C program

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

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

`Seashell` may be kind and require it for us, but it will give a warning message.

`printf` is part of the C `stdio` (**st**an**d**ard **i**/**o**) module. We will introduce C modules later, but for now we will just use the C equivalent of “require” without any discussion:

```
// My second C program

#include <stdio.h>   // <-- require stdio module

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

```
Hello, World
```

In the slides, we typically omit the `#include` for 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\nfun!");
```

```
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", 1 + 1, 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 "**d**ecimal format").

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

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

```
I am 100 % sure you should watch yourspacing!
```

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

```
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 sufficent.

example: simple testing

```
#include <stdio.h>

// my_sqr(x) squares x
int my_sqr(const int x) {
  return x * x;
}

// as we will explain later,
// we are defining main "below" my_sqr

int main(void) {
  printf("Testing: my_sqr(%d) = %d\n", 3, my_sqr(3));
  printf("Testing: my_sqr(%d) = %d\n", -3, my_sqr(-3));
}
```

```
Testing: my_sqr(3) = 9
Testing: my_sqr(-3) = 9
```

Top-level expressions

In C, you cannot have any *top-level* **expressions**.

You can have a top-level (global) *definition* that contains an 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
```

In addition, in C99 you cannot define a local function within another function.

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.

```
int my_abs(const 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;
  }
}
```

example: collatz

```racket
(define (collatz n)
  (cond
    [(= n 1)
      1]
    [(even? n)
      (collatz (/ n 2))]
    [else
      (collatz (+ 1 (* 3 n)))]))
```

```c
int collatz(const int n) {

  if (n == 1) {
    return 1;
  } else if (n % 2 == 0) {
    return collatz(n / 2);
  } else {
    return collatz(3*n + 1);
  }
}
```

Recursion in C behaves the same as in Racket.

`cond` produces a value and can be used inside of an expression:

```
(+ y (cond [(< x 0) -x]
           [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 of how "statements" differ from expressions. For now, only use `if` as we have demonstrated:

```
if (q1) {
  return a1;
} else if (q2) {
  return a2;
} else {
  return a3;
}
```

The C `?:` operator does produce a value and behaves the same as Racket's `if` special form.

```
;; 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))
```

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

```
// C's ?: operator
const int c = q ? a : b;
const int abs_v = (v >= 0) ? v : -v;
const int max_ab = (a > b) ? a : b;
```

Recursion

As we have seen, recursion in C behaves the same as in Racket.

```
;; (sum k) produces the sum of 0...k
;; sum: Nat -> Nat
(define (sum k)
  (cond [(zero? k) 0]
        [else (+ k (sum (- k 1)))]))
```

```
// sum(k) produces the sum of 0...k
// requires: k >= 0
int sum(const int k) {
  if (k == 0) {
    return 0;
  } else {
    return k + sum(k-1);
  }
}
```

example: accumulative recursion

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

int sum(const int k) {
  return accsum(k, 0);
}
```

This example illustrates the importance of ordering C functions.

Because `sum` calls `accsum`, we placed `accsum` *before* (or *“above”*) `sum` in the code.

In C, a function must be defined (or more precisely, *declared*) **before** it appears in the code.

A ***forward declaration*** declares an identifier (*e.g.,* a function) *before* its definition.

The forward declaration for a function is a header without a body (code block).

It *declares* the function return type and the parameter types.

Forward declarations of *variables* are uncommon.

example: forward declaration (function)

```
int accsum(const int k, const int acc);   // forward declaration

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

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

For *mutually recursive functions*, at least one of the functions **must** have a declaration (above the other function).

C ignores the parameter names in a function declaration.

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

```
  int accsum(const int, const int ignored);
```

It is good practice to include the correct parameter names in the declaration to aid communication.

Declaration vs. definition

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

- A declaration only specifies the *type* of an identifier.
- A definition instructs C to *"create"* (or "define") an identifier.
  A definition includes the type of the identifier and provides the information necessary to create it.

A definition also includes a declaration.

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

In these notes, we use *declaration* when describing behaviour that applies to both definitions and declarations.

Many C programmers use the term “declaration” when “definition” is more appropriate, but because a definition includes a declaration, it is not a serious offense.

Scope

```
(define g 2)

(define (f p)
  (define l (+ g p))
  (sqr l))
```

```
const int g = 2;

int f(const int p) {
  const int l = g + p;
  return l * l;
}
```

Ignoring module scope, the scope behaviour in C is consistent with the behaviour in Racket. Above, `g` is a **global** variable (accessible within all functions). The parameter `p` and the variable `l` are **local** to the function they appear in.

In C, local scope is also known as "block scope" to reflect the use of code blocks (`{}`).

```racket
(define a 2)
(define b 3)

(define (f b)
  (define a 4)
  (local
    [(define a 5)]
    (+ a b)))
```

```c
const int a = 2;
const int b = 3;

int f(const int b) {
  const int a = 4;
  {
    const int a = 5;
    return a + b;
  }
}
```

While you should generally *avoid* re-using the same identifiers (names), most languages (including Racket and C) use the *innermost* (or most local) context.

Each C block (`{}`) creates a new local environment (similar to `local` in Racket).

In the above example, `f(10)` returns `15`.

Program scope

By default, all C functions and global variables have **program** scope and *are available in other modules* (`.c` files).

A client module can simply *declare* the identifiers it requires.

```
// sum.c
// define p & sum

const int p = 2;

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

```
// client.c
// declare p & sum

extern const int p;
int sum(const int k);

int double_sum(const int n) {
  return p * sum(n);
}
```

```
// declare sum
int sum(const int k);
```

To declare that the *function* `sum` appears in another module, we use the same *forward declaration* syntax.

```
// declare p
extern const int p;
```

To declare that the *variable* `p` appears in another module, we use the `extern` keyword and do not initialize the variable.

`extern` indicates the variable is *defined* `extern`ally or "somewhere else".

Module scope

The `static` keyword gives a function or variable **module scope** so it is **only available** within the *current module* (`.c` file).

```
int program_scope_function (const int n) { ... }
static int module_scope_function (const int n) {...}

const int program_scope_variable = 42;
static const int module_scope_variable = 23;
```

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

The C keyword `static` is **not** related to *static typing*.

The `static` keyword is also useful to avoid “name collision”. If two modules have global functions (or variables) with the same name, C generates an error. To avoid this problem, use `static` to give the functions (or variables) *module scope* whenever appropriate.

Name collision is less likely in Racket because identifiers have module scope by default and and must be `provide`d to have program scope.

The scope terminology we are using (local, module, program) is a *simplified* version of scope and ***linkage*** in C (CP:AMA 18.1,18.2).

Modularization: C interface files

In C, we can create a module ***interface file*** (`.h` file) that is separate from the module *implementation* (`.c` file).

The *interface file* contains the interface documentation and the **declarations** for the interface functions and variables.

example: sum module

```
// sum.c [IMPLEMENTATION]

const int p = 2;

// see sum.h for details
int sum(const int k) {
  if (k == 0) {
    return 0;
  } else {
    return k + sum(k-1);
  }
}
```

```
// sum.h [INTERFACE]

extern const int p;

// sum(k) finds the sum of all
//      integers from 0...k
// requires: k >= 0
int sum(const int k);
```

To use another module, a client needs to *“copy”* the declarations from the module’s interface file (`.h`) and *“paste”* them into the client file, but that would be time consuming and would remove one of the advantages of modularization (*maintainability*).

```
// client.c

///// declarations copied and pasted from sum.h
extern const int p;
int sum(const int k);
/////

int double_sum(const int n) {
  return p * sum(n);
}
```

#include

Fortunately, C has a built-in *"copy & paste"* feature:

```
#include "module.h"
```

"pastes" the contents of the file `module.h` into the current file.

```
// client.c -- ORIGINAL                 // client.c -- AFTER PASTE

#include "sum.h"                ⇒       extern const int p;

                                        // sum(k) finds the sum of all
                                        //     integers from 0...k
                                        // requires: k >= 0
                                        int sum(const int k);

int double_sum(const int n) {           int double_sum(const int n) {
  return p * sum(n);                      return p * sum(n);
}                                       }
```

Each implementation (`module.c`) should *also* `#include` its own interface file (`module.h`). This helps to detect any discrepancies between the interface and the implementation. It also avoids additional forward declarations in the implementation.

Later, we see additional reasons to include the interface file in the implementation.

Seashell environment

If a client "requires" a module, `#include`-ing the interface file (`module.h`) makes the interface functions (and variables) available.

Conceptually, `#include` is very similar to Racket's `require`, but the behaviour is very different: `require` "runs" the *implementation*, whereas `#include` "copy & pastes" the *interface*.

The `Seashell` environment *automatically* "bundles" into your program the *implementation* of each module you `#include`.

This is one of the many tasks that `Seashell` manages for you when you "run" your program.

Preprocessor directives

`#include` is a ***preprocessor directive***.

When you "run" a C program, there are several "stages" that occur. The very first stage is the *preprocessor*, which scans your program looking for **directives**.

Directives start with # and are "special instructions" that can modify your program before continuing to the next stage.

You typically never "see" the effects of the preprocessor (*e.g.*, you will not see the `#include` directive "paste" into your file).

`#include` is the only directive we will use in this course.

The CP:AMA textbook frequently uses the `#define` directive.

In its simplest form it performs a *search & replace*.

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

int my_add(const 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.

C standard modules

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

Fortunately, C provides several ***standard modules*** (called “libraries”) with many convenient functions. For example, `printf` is part of the `stdio` module.

When you `#include` a standard module, the syntax is a little different (<> instead of `""`).

```
// For standard C modules:
#include <stdio.h>

// For your own modules:
#include "mymodule.h"
```

In these notes we often omit `#include`s to save space.

assert

A useful standard module is `assert`, which provides a testing mechanism similar to Racket's `check-expect`.

`assert(e)` *stops* the program and display a message if the expression `e` is false (nothing happens if the expression is true).

`assert` is especially useful for verifying requirements. In this course, you are expected to `assert` your requirements when feasible.

```
#include <assert.h>

// requires: k >= 0
int sum(const int k) {
  assert(k >= 0);
  //...
```

example: testing module

When designing a module to provide to clients (or for your assignments), you should also create a corresponding testing module. The testing module is where you should put your `main` function.

```c
// test-sum.c: testing module for sum.h

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

int main(void) {
  assert(sum(0) == 0);
  assert(sum(1) == 1);
  assert(sum(2) == 3);
  assert(sum(3) == 6);
  assert(sum(10) == 55);
}
```

Boolean 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`).

```
#include <stdbool.h>

const bool is_cool = true;

bool is_even(const 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)              const 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 `enum`s 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.

Structures

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

```
(struct posn (x y)              struct posn {
  #:transparent)                  int x;
                                  int y;
                                };

(define p (posn 3 4))           const struct posn p = {3,4};

(define a (posn-x p))           const int a = p.x;
(define b (posn-y p))           const int b = p.y;
```

No *functions* are generated when you define a structure in C.

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

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

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

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

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.

The textbook refers to the structures we have seen so far as **declarations** (not definitions).

To avoid unnecessary confusion, and to be more consistent with other terminology we use, we continue to refer to structure declarations as *definitions*.

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

(define a (posn-x p))          const int a = p.x;
(define b (posn-y p))          const int b = p.y;
```

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:

```
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:

```
const struct posn p = {.x = 3}; //.y = 0
```

The *equality* operator (==) **does not work with structures**. You have to define your own equality function.

```
bool posn_equal (const struct posn a, const 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:

```
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.

```
struct posn scale(const 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. We have two suggestions.

First, you can define separate constants as required and `return` a constant.

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

Alternatively, you can define a simple “build” function for a structure (similar to a “make” function in Racket).

```
struct posn build_posn(const int a, const int b) {
  const struct posn r = {a, b};
  return r;
}

struct posn scale(const struct posn p, const int f) {
  return build_posn(p.x * f, p.y * f);
}
```

We are using “build” terminology (`build_posn`) instead of “make” terminology (`make_posn`) to avoid confusion when we introduce memory management later.

To make a structure accessible (and “transparent”) to clients, place the definition in the *interface file* (`.h`).

```c
// posn.c

#include "posn.h"

int f(const struct posn p) {
  //...
}
```

```c
// posn.h

struct posn {
  int x;
  int y;
};

int f(const struct posn p);
```

This is another reason why every implementation file (`module.c`) should include its own interface file (`module.h`).

We discuss opaque C structures in Section 12.

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 differences between local, module and program scope and demonstrate how `static` and `extern` are used

- write modules in C with implementation and interface files
- explain the significance of the 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

Introduction to Imperative C

Readings: CP:AMA 2.1, 4.2–4.5, 5.2, 6, 9.4

- 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".

example: functional programming paradigm

```
(define n 5)
(add1 n)        ; => 6
n               ; => 5
```

With functional programming, `(add1 n)` produces a **new** value, but it does not actually *change* `n`. Once `n` is `define`d, it is a **constant** and always has the same value.

```
(define lon '(15 23 4 42 8 16))
(sort lon <)      ; => '(4 8 15 16 23 42)
lon               ; => '(15 23 4 42 8 16)
```

Similarly, `(sort lon)` produces a **new** list that is sorted, but the original list `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.

begin

the `begin` special form evaluates a *sequence* of expressions.

`begin` evaluates each expression but “ignores” all of the values except the last one. `begin` **produces the value** of the **last** expression.

```
(define (mystery)
  (begin
    "four"
    'four
    (+ 2 2)))
```

```
(mystery)
4
```

There are two reasons you rarely see `begin` used in practice...

The first reason is a bit sneaky: In full Racket there is an *implicit* `begin` (similar to implicit `local`).

```
(define (mystery)
  (begin
    "four"
    'four
    (+ 2 2)))
```

```
;; this is equivalent
(define (mystery)
  "four"
  'four
  (+ 2 2))
```

The more obvious reason is that it’s rarely “useful” to evaluate an expression and “ignore” the result.

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.

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

The existence 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.

Racket also has a `printf` function very similar to C's `printf`.

In Racket's `printf` the `~a` placeholder works for **a**ll types.

```
(printf "There are ~a lights!\n" "four")
(printf "There are ~a lights!\n" 'four)
(printf "There are ~a ~a!\n" (+ 2 2) "lights")
```

```
There are four lights!
There are four lights!
There are 4 lights!
```

With side effects, `begin` makes more sense.

In fact, the combination of `begin` and `printf` can be quite useful to trace or debug your code.

```
(define (noisy-add x y)
  (printf "HEY! I'M ADDING ~a PLUS ~a!\n" x y)
  (+ x y))


(define (collatz n)
  (printf "~a " n)
  (cond [(= n 1)   1]
        [(even? n) (collatz (/ n 2))]
        [else      (collatz (+ 1 (* 3 n)))]))
```

Documentation: side effects

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

```
;; (take-headache-pills qty) simulates taking qty pills
;; take-headache-pills: Nat -> Str
;; requires: qty > 0
;; effects: may display a message

(define (take-headache-pills qty)
  (cond [(> qty 3) (printf "Nausea\n")])
  "Headache gone!")
```

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.

Some functions are designed to *only* cause a side effect, and not produce a value.

For example, Racket's `printf` produces `#<void>`, which is a special Racket value to represent "nothing" (in contracts, use `-> Void`).

```
;; (say-hello) displays a friendly message
;; say-hello: Void -> Void
;; effects: displays a message

(define (say-hello)
  (printf "Hello\n"))
```

In C, we used `void` to declare that a function has no parameters.

It is also used to declare that a function returns “nothing”.

```
// say_hello() 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***.

A C block (`{}`) is similar to Racket's `begin`: statements are evaluated **in sequence**.

Unlike `begin`, a block (`{}`) does not "produce" a value. This is one reason why `return` is needed.

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

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 ends the execution of 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)

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

int add1(const int x);          // function declaration

int add1(const int x) {         // function definition
                                // 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 it's *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").

example: changing state

```
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 `n` is **not** a `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 *mutable*.

```
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*** (=).

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

The assignment operator can be used on `struct`s.

```
struct posn p = {1,2};
struct posn q = {3,4};
p = q;
p.x = 23;
```

The = in an *initialization* is **not** the assignment operator.

Some initialization syntaxes are **invalid** with the assignment operator.

```
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.

```
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.

```
int n = 10;

int addn(const 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.

```
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 why accidentally using a single = instead of double == for equality is so dangerous!

```
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

```
x = x + 1;
y = y + z;
```

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

```
x += 1;        // equivalent to x = x + 1;
y += z;        // equivalent to y = y + z;
```

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`.

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

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
- 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 1, 7.1, 7.2, 7.3, 7.6, 11.1, 11.2, Appendix E

Course Notes: Appendix A.3

- 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.3.

Accessing memory

The smallest accessible unit of memory is a byte.

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

For example, if you have 1MB of memory (RAM), the *address* of the first byte is 0 and the *address* of the last byte is 1048575 ($2^{20} - 1$).

Note: Memory addresses are usually represented in *hex*, so with 1MB of memory, the address of the first byte is 0x0, and the address of the last byte is 0xFFFFF.

You can visualize the computer memory as a collection of “labeled mailboxes” where each mailbox stores a byte.

address (1 MB of storage)	contents (one byte per address)
`0x00000`	`00101001`
`0x00001`	`11001101`
`...`	`...`
`0xFFFFE`	`00010111`
`0xFFFFF`	`01110011`

In the above table, the contents are arbitrary examples.

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

```
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, the name is *substituted* for 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 that variable – but **how much space?**

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.

```
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 our `Seashell` environment, each `int`eger 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

```
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.

identifier	type	# bytes	starting address
`n`	`int`	`4`	`0x5000`

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

address	`0x5000`	`0x5001`	`0x5002`	`0x5003`
contents	`00000000`	`00000000`	`00000000`	`00000000`

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 . . . 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

```
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, `char`s 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:

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

Always use **single** quotes with characters:

`"a"` is **not** the same as `'a'`.

example: C characters

The `printf` placeholder to display a *character* is `"%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.

```
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

When a structure **type** is *defined*, no memory is reserved:

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

Memory is only reserved when a **variable** is defined.

```
struct posn p1 = {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.

```
struct mystruct {
  int x;
  char c;
  int y;
};

struct mystruct s = {3, 'a', 4};

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

```
sizeof(struct mystruct) = 12
sizeof(s) =               12
```

You **must** use the `sizeof` operator to determine the size of a structure.

The size may depend on the *order* of the fields:

```
struct s1 {                               struct s2 {
  char c;                                   char c;
  int i;                                    char d;
  char d;                                   int i;
};                                        };
```

```
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.

Floating point numbers in C

The C `float` (floating point) type can represent real (non-integer) values and has a much larger range than integers.

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

Unfortunately, `float`s 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):

```
(sqrt 2)            ; => #i1.4142135623730951
(sqr (sqrt 2))      ; => #i2.0000000000000004
```

example 1: inexact floats

```
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) {
  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

```
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 used `check-within` for inexact numbers in Racket, use a similar technique for testing in 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.

Just as we might represent a number in decimal as 6.022×10^{23}, a `float` uses a similar strategy.

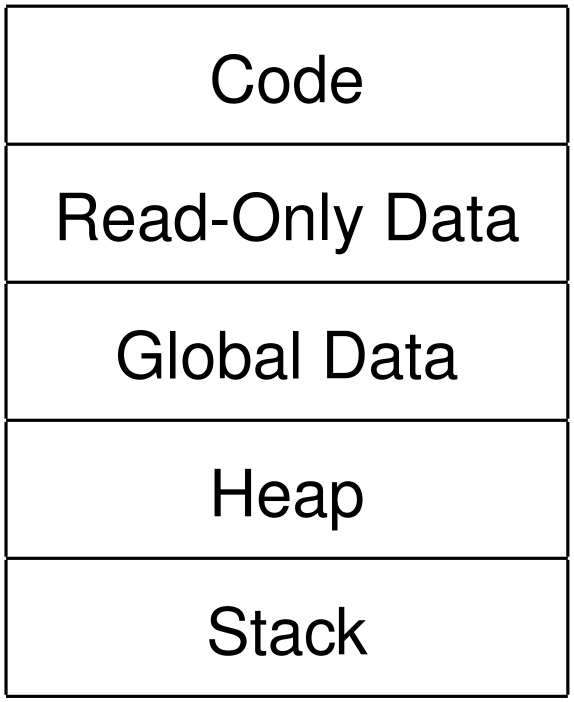
A `float` in our `Seashell` environment uses 32 bits: 24 bits for the *mantissa* and 8 bits for the *exponent*.

A `double` uses 64 bits (53 + 11).

`float`s 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:



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 write your 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:

```
int n = 0;
```

The location of the reserved space depends on whether the variable is **global** or **local**. First, we discuss global variables.

All of the 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**.

We discuss local variables and the **stack** section after control flow.

The read-only and global memory sections are created and initialized when the code is compiled.

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).

```c
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`. This location is known as the ***return address***.

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

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

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.

Space for a *global* variable is reserved before the program begins execution.

Space for a *local* variable is reserved **when the function is called**. The space is reserved within the newly created stack frame.

When the function `return`s, the variable (and the entire frame) “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

```
 1  int h(int i) {
 2    int r = 10 * i;
 3    return r;
 4  }
 5
 6  int g(int y) {
 7    int c = y * y;
 8    ⇒ return c;
 9  }
10
11  int f(int x) {
12    int b = 2*x + 1;
13    return g(b + 3) + h(b);
14  }
15
16  int main(void) {
17    int a = f(2);
18    //...
19  }
```

```
=========================
g:
  y: 8
  c: 64
  return address: f:13
=========================
f:
  x: 2
  b: 5
  return address: main:17
=========================
main:
  a: ???
  return address: OS
=========================
```

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.

example: recursion

```
1  int sum_first(int n) {
2    ⇒if (n == 0) {
3      return 0;
4    } else {
5      return n + sum_first(n-1);
6    }
7  }
8
9  int main(void) {
10   int a = sum_first(2);
11   //...
12 }
```

```
==============================
sum_first:
  n: 0
  return address: sum_first:5
==============================
sum_first:
  n: 1
  return address: sum_first:5
==============================
sum_first:
  n: 2
  return address: main:10
==============================
main:
  a: ???
  return address: OS
```

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.

```
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.

```
int g = 0;
```

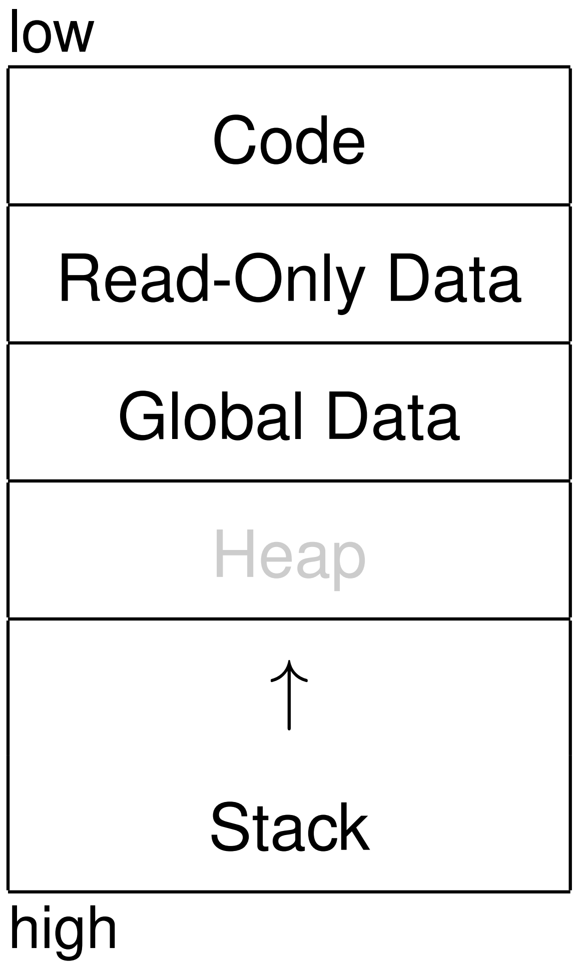
A **local** variable (on the *stack*) that is uninitialized has an **arbitrary** initial value.

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

`Seashell` gives you a warning if you access 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)



Model

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

Any 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 all local variables are placed in the frame

return

We have already seen the `return` control flow statement.

When a function `return`s:

- 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 execute more than one statement when the expression is true, braces (`{}`) are used to insert a compound statement block (a sequence of statements) in place of a single statement.

```
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.)*

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

The `if` statement also supports an `else` statement, where a second statement (or block) is executed if the expression is false.

```
if (expression) {
  statement(s)
} else {
  statement(s)
}
```

`else`s can be combined with more `if`s for multiple conditions.

```
if (expression) {
  statement(s)
} else if (expression) {
  statement(s)
} else if (expression) {
  statement(s)
} else {
  statement(s)
}
```

If there is an `if` condition to `return`, there may not be a need for an `else` block. In this simple example, there is not much difference but in larger examples it can reduce the number of indentation levels required.

```
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) return 0;
  return k + sum(k-1);
}
```

Braces are sometimes necessary to avoid a “dangling” `else`.

```
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 structure a similar 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 `goto`s, you must also have *labels* (code locations).

```
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 ***looping***.

```
  while (expression) statement
```

`while` is similar to `if`: the `statement` is only executed `if` the `expression` is true.

The difference is, `while` **repeatedly** *"loops back"* and executes the `statement` **until the `expression` is false**.

Like with `if`, you should always use braces (`{}`) for a *compound statement*, even if there is only a single statement.

example: while loop

variable	value
i	2

```
⇒ int i = 2;
while (i >= 0) {
  printf("%d\n", i);
  --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);
}
```

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

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

Loops can be “nested” within each other.

```
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:

```
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 code will produce an endless loop.

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

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

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

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);
```

break

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.

```
while (1) {
  // stuff
  if (early_exit_condition) break;
  // more stuff
}
```

continue

The `continue` control flow statement skips over the rest of the statements in the current block (`{}`) and “continues” with the loop.

```
// only concerned with fun numbers
int i = 0;
while (i <= 9999) {
  ++i;
  if (!is_fun(i)) continue;
  //...
}
```

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.

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

This `while` example

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

is equivalent to

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

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 “initialization” statement can be a *definition* instead.

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

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

For the above `for` loop, the equivalent `while` loop would have an extra block.

```
{
  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 already initialized

for (;;) {...}              // endless loop
```

You can use the *comma operator* (`,`) to use more than one expression in the “init” and “update” statements of the `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
- describe the `float` and `double` types
- 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

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.

```
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"`.

memory sections (so far)

We can see how the different memory sections are arranged.

```
const int r = 42;
int g = 15;

int main(void) {

  int s = 23;

  printf("the address of main is:   %p\n", &main); // CODE
  printf("the address of    r is:   %p\n", &r);    // READ-ONLY
  printf("the address of    g is:   %p\n", &g);    // GLOBAL DATA
  printf("the address of    s is:   %p\n", &s);    // STACK
}
```

```
  the address of main is:   0x46c060
  the address of    r is:   0x477620
  the address of    g is:   0x68a9e0
  the address of    s is:   0x7fff4f6031a0
```

To illustrate the behaviour of the stack “growing down”, we can display the address of the parameter n in each recursive call:

```
int sum_first(int n) {
  printf("the address of n is: %p\n", &n);
  if (n <= 0) return 0;
  return n + sum_first(n-1);
}
```

```
the address of n is: 0x7fff2f7037d0
the address of n is: 0x7fff2f703700
the address of n is: 0x7fff2f703630
the address of n is: 0x7fff2f703560
the address of n is: 0x7fff2f703490
the address of n is: 0x7fff2f7033c0
the address of n is: 0x7fff2f7032f0
the address of n is: 0x7fff2f703220
...
```

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.

```
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 *`).

The **value** of a pointer is an **address**.

```
int i = 42;
int *p = &i;
int *q = p;

printf("address of i  (&i) = %p\n", &i);
printf("value of p     (p) = %p\n",  p);
printf("value of q     (q) = %p\n",  q);
```

```
address of i  (&i) = 0xf004
value of p     (p) = 0xf004
value of q     (q) = 0xf004
```

To make working with pointers easier in these notes, we often use shorter, simplified ("fake") addresses.

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.

Note: `sizeof(int *)` and `sizeof(char *)` are both 8 bytes.

```
int i = 42;
char c = 'c';
int *pi = &i;
char *pc = &c;

printf("sizeof(i)      = %zd\n", sizeof(i));
printf("sizeof(c)      = %zd\n", sizeof(c));
printf("sizeof(pi)     = %zd\n", sizeof(pi));
printf("sizeof(pc)     = %zd\n", sizeof(pc));
printf("sizeof(int *)  = %zd\n", sizeof(int *));
printf("sizeof(char *) = %zd\n", sizeof(char *));
```

```
sizeof(i)      = 4
sizeof(c)      = 1
sizeof(pi)     = 8
sizeof(pc)     = 8
sizeof(int *)  = 8
sizeof(char *) = 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".

```
int i = 42;
int *p = &i;      // pointer p points at i
int j = *p;       // integer j is 42
```

The value of ∗&i is simply the value of i.

example: indirection

```
int i = 42;
int *p = &i;

printf("address of i                (&i) = %p\n",  &i);
printf("value of i                   (i) = %d\n\n", i);

printf("address of p                (&p) = %p\n",  &p);
printf("value of p                   (p) = %p\n",   p);
printf("value of what p points at (*p) = %d\n",  *p);
```

```
address of i                (&i) = 0xf004
value of i                   (i) = 42

address of p                (&p) = 0xf008
value of p                   (p) = 0xf004
value of what p points at (*p) = 42
```

The * symbol is used in three different ways in C:

- as the *multiplication operator* between expressions

```
k = i * i;
```

- in pointer *declarations* and pointer *types*

```
int *pi = &i;
s = sizeof(int *);
```

- as the *indirection operator* for pointers

```
i = *pi;
```

`(*pi * *pi)` is a confusing but valid C expression.

C mostly ignores white space, so these are equivalent

```
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 `pi` is an "`int *`".

However, *in the declaration* the `*` "belongs" to the `pi`, not the `int`, and so style A is used in this course and in CP:AMA.

This is clear with multiple declarations: (not encouraged)

```
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

In the following code, “`pi` points at `i`”, but what if we want a pointer to “point at `pi`”?

```
int i = 42;
int *pi = &i;      // pointer pi points at i
```

In C, we can declare a **pointer to a pointer**:

```
int **ppi = &pi;   // pointer ppi points at pi
```

C allows any number of pointers to pointers. More than two levels of “pointing” is uncommon.

example: pointers to pointers

```
int i     = 42;
int *pi   = &i;
int **ppi = &pi;
```

```
address of i                                 (&i) = 0xf004
value of i                                    (i) = 42

address of pi                               (&pi) = 0xf008
value of pi                                  (pi) = 0xf004
value of what pi points at                  (*pi) = 42

address of ppi                             (&ppi) = 0xf00C
value of ppi                                (ppi) = 0xf008
value of what ppi points at                (*ppi) = 0xf004
value of what ppi points at points at     (**ppi) = 42
```

`(**ppi * **ppi)` is a confusing but valid C expression.

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).

Pointer assignment

Consider the following code

```
int i = 5;
int j = 6;

int *p = &i;
int *q = &j;

p = q;
```

The statement `p = q;` is a ***pointer assignment***. It means “`p` now points 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`**.

Using the same initial values,

```
int i = 5;
int j = 6;

int *p = &i;
int *q = &j;
```

the statement

```
*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.

example: aliasing

```
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:

```
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.

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”.

```
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 (int *).

```
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 bugs: the best strategy is to use straightforward code.

example: mutation side effects

```
//  effects: swaps the contents of *x and *y
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main(void) {
  int x = 3;
  int y = 4;
  printf("x = %d, y = %d\n", x, y);
  swap(&x, &y);                          // Note the &
  printf("x = %d, y = %d\n", x, y);
}
```

```
x = 3, y = 4
x = 4, y = 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.

```
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).

```
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.

example: pointer return types

The return type of a function can also be an address (pointer).

```
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.

```
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 `return`s, the stack frame “disappears”, and all memory within the frame should be considered **invalid**.

Passing structures

Recall that when a function is called, a **copy** of each argument value is placed into the stack frame.

For structures, the *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 *especially* problematic with recursive functions, and may even cause a *stack overflow* to occur.

To avoid structure copying, it is common to pass the *address* of a structure to a function.

```
int sqr_dist(struct posn *p1, struct posn *p2) {
  const int xdist = (*p1).x - (*p2).x;
  const int ydist = (*p1).y - (*p2).y;
  return xdist * xdist + ydist * ydist;
}

int main(void) {
  const struct posn p1 = {2,4};
  const struct posn p2 = {5,8};

  assert(sqr_dist(&p1,&p2) == 25);     // note the &
}
```

```
int sqr_dist(struct posn *p1, struct posn *p2) {
  const int xdist = (*p1).x - (*p2).x;
  const int ydist = (*p1).y - (*p2).y;
  return xdist * xdist + ydist * ydist;
}
```

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**.

```
int sqr_dist(struct posn *p1, struct posn *p2) {
  const int xdist = p1->x - p2->x;
  const 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 p->x and p->y by multiplying by f

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.

```
void cannot_change_posn(const struct posn *p) {
  p->x = 5;   // INVALID
}
```

The `const` should be placed 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.

```
int *p;                 // p can change, can point at any int

const int *p;           // p can change, must point at 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.

```
const int i = 42;
int const i = 42;
```

Goals of this Section

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

- declare and de-reference pointers
- use the two new operators (& $*$)
- use pointers to structures as parameters and explain why parameters are often pointers to structures

I/O & Testing

Readings: CP:AMA 2.5

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 on a hard drive or even a different computer.

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 (**f**ile `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.

```
#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.

```
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).

You can even 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.

```
int TRACELEVEL = 2;

int main(void) {
  if (TRACELEVEL >= 1) printf("starting main\n");
  int sum = 0;
  if (TRACELEVEL >= 2) printf("before loop: sum = %d\n",sum);
  for (int i=0; i < 10; ++i) {
    if (TRACELEVEL >= 3) printf("loop iteration: i = %d\n",i);
    sum += i;
    if (TRACELEVEL >= 3) printf("sum has changed: sum = %d\n",sum);
  }
  if (TRACELEVEL >= 2) printf("after loop: sum = %d\n",sum);
  if (TRACELEVEL >= 1) printf("leaving main\n");
}
```

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 **E**nd **O**f **F**ile (EOF) has been reached.

`EOF` is a special value to indicate that there is no more input.

In our `Seashell` environment, a `Ctrl-D` ("Control D") keyboard sequence sends an `EOF`.

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>
```

scanf

In C, the `scanf` function is the counterpart to the `printf` function.

```
scanf("%d",&i); // read in an integer, store in i
```

Just as with `printf`, you can read more than one value in a single call to `scanf` with multiple placeholders.

However, in this course **only read in one value per `scanf`**.

This will help you debug your code and facilitate our testing, `scanf` requires a **pointer** to a variable to **store the input value**.

```
  count = scanf("%d",&i); // read in an int, store in i
```

The return value of `scanf` is the number (count) of values successfully read, so in this course a value of one is “success”.

`scanf("%d",&i)` will ignore whitespace (spaces and newlines) and read in the next integer. However, if the next input to be read is not a valid integer (e.g., a letter), it will stop reading and return zero.

The return value can also be the special constant value `EOF`. You should **not assume that** `EOF` is zero (or `false`).

`scanf` can be used to read in `char`acters (`"%c"`).

You may or may not want to ignore whitespace when reading in a `char`.

```
// reads in next character (may be whitespace character)
count = scanf("%c",&c);

// reads in next character, ignoring whitespace
count = scanf(" %c",&c);
```

The extra leading space in the second example indicates that whitespace should be ignored.

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)
  name)
```

example: interactive Racket

```
(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)
```

example 1: interactive C

```c
int main(void) {
  int count = 0;
  int i = 0;
  int sum = 0;

  printf("how many numbers should I sum? ");
  scanf("%d",&count);
  for (int j=0; j < count; ++j) {
    printf("enter #%d: ", j+1);
    scanf("%d", &i);
    sum += i;
  }
  printf("the sum of the %d numbers is: %d\n", count, sum);
}
```

example 2: interactive C

```
int main(void) {
  int count = 0;
  int i = 0;
  int sum = 0;

  printf("keep entering numbers, press Ctrl-D when done.\n");
  while (1) {
    printf("enter #%d: ", count+1);
    if (scanf("%d", &i) != 1) break;
    sum += i;
    ++count;
  }
  printf("\n");
  printf("the sum of the %d numbers is: %d\n", count, sum);
}
```

Interactive testing

In DrRacket, the *interactions window* was quite a useful tool for debugging our programs.

In `Seashell`, we can create **interactive testing modules**.

Consider an example with a simple arithmetic module.

```
// 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 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);
      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* 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 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 become quite tedious to rely on human input at the keyboard.

Fortunately, the `Seashell` environment has support to *automate* interactive testing.

Seashell testing Demo

(to be done in class)

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 (`INT_MAX` or `INT_MIN`)
- test for special argument values (`-1`, `0`, `1`, `NULL`)

Module testing

When testing a *module* that has side effects, testing each individual function may not be sufficient.

You may also have to test the interaction between functions, by testing *sequences* of function calls.

When you test the interaction **between** modules and **groups** of modules at once, it is known as ***integration testing***. Whenever a module is changed, a full integration test should be run.

Goals of this Section

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

- use the I/O terminology introduced
- use the input functions `read` in Racket and `scanf` in C to make interactive programs
- use the `Seashell` testing environment effectively

Arrays & Strings

Readings: CP:AMA 8.1, 8.3, 9.3, 10, 12.1, 12.2, 12.3, 13

Arrays

The only two types of "compound" data storage *built-in* to C are `struct`ures and ***arrays***.

```
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]`. The last element is `my_array[5]`.

In computer science we often start counting at `0`.

example: accessing array elements

Each individual array element can be used as if it was a variable.

```
int a[6] = {4, 8, 15, 16, 23, 42};

int j = a[0];           // j is 4
int k = a[j];           // k is 23
int *p = &a[k-20];      // p points at a[3]

a[2] = a[a[0]];         // a[2] is now 23
a[0]++;                 // a[0] is now 5
++a[1];                 // a[1] is now 9
```

example: arrays & iteration

Arrays and iteration are a powerful combination.

```
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

Like variables, an uninitialized array

```
int a[5];
```

is zero-filled if the array is *global*. If the array is *local*, it is filled with arbitrary ("garbage") values from the stack.

As with structures, the array braces (`{}`) syntax is only valid in **initialization**.

If there are not enough elements in the braces, the remaining values are initialized to zero (even with local arrays).

```
int b[5] = {1, 2, 3};     // b[3] & b[4] = 0
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.

```
  int a[] = {4, 8, 15, 16, 23, 42};  // int a[6] = ...
```

This syntax is only allowed if the array is initialized.

```
  int b[];  // INVALID
```

Similar to structures, C99 supports a partial initialization syntax.

```
  int a[100] = { [50] = 1, [25] = -1, [75] = 3 };
```

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.

```
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` has six elements (`int a[6];`) the size of `a` is $(6 \times$ `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
```

addresses	contents (4 bytes)
`0x5000 ... 0x5003`	4
`0x5004 ... 0x5007`	8
`0x5008 ... 0x500B`	15
`0x500C ... 0x500F`	16
`0x5010 ... 0x5013`	23
`0x5014 ... 0x5017`	42

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.

Often, the array length is stored in a separate variable.

```
const int a_length = 6;
int a[a_length] = {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.

```
  int len = sizeof(a) / sizeof(a[0]);
```

The CP:AMA textbook uses this on occasion.

However, in practice, this should never be done, as the `sizeof` operator only properly reports the array size in some circumstances.

The array identifier

The value of an array identifier (the array name) can be used as a pointer to the first element of the array.

```
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
```

The **value** of `a` is the same as the **address** of the array (`&a`), which is also the address of the first element (`&a[0]`).

Dereferencing the array (`*a`) is equivalent to referencing the first element (`a[0]`).

example: array identifier

In an expression, `*a` is the same as `a[0]`.

```
int a[3] = {0, 0, 0};
printf("a[0] = %d\n", a[0]);
*a = 5;
printf("a[0] = %d\n", a[0]);
```

```
a[0] = 0;
a[0] = 5;
```

The array identifier itself cannot be changed or assigned to and is effectively “constant”.

```
int a[3] = {0, 0, 0};
int b[3] = {1, 2, 3};
a = b;                        // INVALID
```

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

```
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.

```
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 prevent mutation and communicate that no mutation occurs.

```
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:

```
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.

```
int slower(struct mystruct s) {
  ...
}

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:

$$\mathtt{p} + \mathtt{i} \times \mathtt{sizeof}(*\mathtt{p}).$$

- 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:

  $$(\mathrm{p} - \mathrm{q})/\mathtt{sizeof}(*\mathrm{p}).$$

  In other words, if `p = q + i` then `i = p - q`.
- Pointers (of the same type) can be compared with the comparison operators: <, <=, ==, !=, >=, >
  (*e.g.*, `if (p < q) ...`).

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.

```
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`:

```
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 headers (declarations):

```
  int sum_array(const int a[], int len) {...}   // a[]
  int sum_array(const int  *a, int len) {...}   //  *a
```

In most contexts, there is no practical difference between an array identifier and a *(constant)* pointer.

The subtle differences between an array and a pointer are discussed at the end of this Section.

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.

```
1 2 3
7 8 9
```

We can represent the data in a simple one-dimensional array.

```
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 `data[r*3 + c]`.

In general, it would be `data[row * NUMCOLS + col]`.

C supports multiple-dimension arrays, but they are not covered in this course.

```
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.

Abstract array functions

In Racket, **Abstract List Functions (ALFs)** (*e.g.*, `filter`, `map`, `foldl`) are powerful and flexible tools for processing lists.

In C, we can create ***Abstract Array Functions (AAFs)***.

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

int main(void) {
  int a[] = {4, 8, 15, 16, 23, 42};
  print_array(a, 6);
  array_map(add1, a, 6);   // we need to define this
  print_array(a, 6);
}
```

```
4 8 15 16 23 42
5 9 16 17 24 43
```

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. However, all of the aforementioned things can be done with ***function pointers***.

A *function pointer* stores the starting address of a function within the code section. The value of a function identifier is its starting address.

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.

The declaration for a function pointer includes the *return type* and all of the *parameter types*, which makes them a little messy.

For example, consider the following function pointer `fp` that points at the function `add1`:

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

int (*fp)(int) = add1;
```

The syntax to declare a function pointer with name `fp` is:

```
return_type (*fp)(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.

Array map

Aside from the function pointer parameter syntax, the definition of `array_map` is straightforward.

```
// effects: replaces each element a[i] with f(a[i])

void array_map(int (*f)(int), int a[], int len) {
  for (int i=0; i < len; ++i) {
    a[i] = f(a[i]);
  }
}
```

example: Array map

```c
#include "array_map.h"

int add1(int i) { return i + 1; }
int sqr(int i) { return i * i; }
int print_element(int i) {
  printf("%d ", i); return i;
}

int main(void) {
  int a[] = {4, 8, 15, 16, 23, 42};
  array_map(print_element, a, 6);  printf("\n");
  array_map(add1, a, 6);
  array_map(print_element, a, 6);  printf("\n");
  array_map(sqr, a, 6);
  array_map(print_element, a, 6);  printf("\n");
}
```

```
4 8 15 16 23 42
5 9 16 17 24 43
25 81 256 289 576 1849
```

Alternatively, the function passed to `array_map` could be a `void` function that accepts a pointer to each element.

```
void add1(int *p) {
  *p += 1;
}

void alt_array_map(void (*f)(int *), int a[], int len) {
  for (int i=0; i < len; ++i) {
    f(a + i);                            // same as f(&a[i])
  }
}
```

Our `array_map` only works with arrays of integers.

In practice, a generic AAF would likely be implemented with `void` pointers, which are described in Section 12.

Array foldl

```
// foldl(f, base, a, len) returns:
//  f(a[len-1], f(a[len-2], ..., f(a[1], f(a[0], base))))

int array_foldl(int (*f)(int, int), int base, int a[],int len) {
  for (int i=0; i < len; ++i) {
    base = f(a[i], base);
  }
  return base;
}

// the only change for foldr would be a different for loop:
//   for (int i=len-1; i >= 0; --i) {
```

In Section 10 we introduce dynamic arrays, where a `filter` AAF makes more sense.

Because C does not have anonymous functions or the ability to dynamically generate closure functions (*e.g.*, `lambda`), AAFs are not as convenient or as popular as ALFs in Racket.

In practice, it is not much more effort to write a function to perform an action on an entire array instead of a single element.

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

void array_add1(int a[], int len) {
  for (int i=0; i < len; ++i) {
    a[i]++;
  }
}
```

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***.

```
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'` (ASCII `0`) is different than `'0'` (ASCII 48), which is the character for the symbol zero.

String initialization

In addition to the regular array initialization syntax, `char` arrays also support a double quote (`"`) notation. When combined with the automatic size declaration (`[]`), the size includes the null terminator.

The following definitions create equivalent 4-character arrays:

```
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")`).

String literals

The C strings used in statements (*e.g.*, with `printf` and `scanf`) are known as ***string literals***.

```
printf("i = %d\n", i);
printf("the value of j is %d\n", j);
```

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

```
void foo(int i, int j) {
  printf("i = %d\n", i);
  printf("the value of j is %d\n", j);
}
```

Although no array name is actually given to each literal, it is helpful to imagine that one is:

```
const char foo_string_literal_1[] = "i = %d\n";
const char foo_string_literal_2[] = "the value of j is %d\n";

void foo(int i, int j) {
  printf(foo_string_literal_1, i);
  printf(foo_string_literal_2, j);
}
```

You should not try to modify a string literal. The behaviour is undefined, and it causes an error in `Seashell`.

Null termination

Because strings are null terminated, we do not have to pass the array length to every function.

```
// e_count(s) counts the # of e's in string s

int e_count(const char *s) {
  int count = 0;
  while (*s) {  // not the null terminator
    if ((*s == 'e')||(*s == 'E')) ++count;
    ++s;
  }
  return count;
}
```

As with “regular” arrays, it is good style to have `const` parameters to communicate that no changes (mutation) occurs to the string.

strlen

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.

```
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.

```
int my_strlen(const char *s) {
  const char *p = s;
  while (*p) {
    ++p;
  }
  return (p-s);
}
```

Lexicographical order

Characters can be easily compared (`c1 < c2`) as they are numbers, so the character **order** is determined by the ASCII table.

If we try to compare two strings (`s1 < s2`), C compares their *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 the 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.

```
int my_strcmp(const char *s1, const char *s2) {
  while (*s1 == *s2) {
    if ((*s1 == '\0') && (*s2 == '\0')) return 0;
    ++s1;
    ++s2;
  }
  if (*s1 < *s2) 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.

```
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:

```
(define (lon<=? lon1 lon2)
  (cond [(empty? lon1) #t]
        [(empty? lon2) #f]
        [(< (first lon1) (first lon2)) #t]
        [(< (first lon2) (first lon1)) #f]
        [else (lon<=? (rest lon1) (rest lon2))]))

(lon<=? '(4 9 1 2 1) '(4 5 9)) ; => #f
(lon<=? '(4 3) '(4 3 2))       ; => #t
```

String I/O

The `printf` placeholder for strings is `%s`.

```
char a[] = "cat";
printf("the %s in the hat\n", a);
```

`printf` prints out characters until the null character is encountered.

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.

```
char name[81];
printf("What is your first name? ");
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**.

In this example, the array is `81` characters and can accommodate first names with a length of up to `80` characters.

What if someone has a *really* long first name?

example: scanf

```
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 size 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 appropriately sized array and using `scanf` is “good enough”.

In practice you would **never** use this insecure method for reading in a string.

The `gets` function does not stop when a space is encountered, and reads in characters until a newline (`\n`) is encountered. It is also very susceptible to overruns, but is convenient to use in this course.

```
char name[81];
printf("What is your full name? ");
gets(name);
```

There are C library functions that are more secure than `scanf` and `gets`.

A 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:

`strcpy(char *dest, const char *src)` overwrites the contents of `dest` with the contents of `src`.

`strcat(char *dest, const char *src)` copies (appends or con**cat**enates) `src` to the end of `dest`.

You should always ensure that the `dest` array is large enough (and don't forget the null terminator).

With this simple implementation of `strcpy`:

```
char *strcpy(char *dst, const char *src) {
   char *d = dst;
   do {
      *d = *src;
      ++d; ++s;
   } while (*src);
   return dst;
}
```

you can crash your program:

```
char c[] = "spam";
strcpy(c + 4, c);
```

Because the start of the destination is also the null terminator of the source, the source never terminates and it fills up the 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.

```
char c[3] = "cat";   // NOT properly terminated!
printf("%s\n", c);
printf("The length of c is: %d\n", strlen(c));
```

```
cat?????????????????

The length of c is: ??
```

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`.

Arrays vs. pointers

Earlier, we said arrays and pointers are *similar* but **different**.

Consider the following two string definitions:

```
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

```
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.

```
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.

```
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 very similar to a **constant** pointer.

```
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 is the same as &a, while p and &p have different values
- `sizeof(a)` is 24, while `sizeof(p)` is 8

p also requires an additional four bytes of storage.

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
- 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 ($\approx$ 800 A.D.).

Problem: Write a Racket function to determine if the sum of a list of positive integers is greater than a specific positive integer.

```
;; (sum>? lon k) determines if sum of lon is > k
;; sum>?: (listof Int) Int-> Bool
;; requires: all elements are positive
```

algorithm 1: (total sum) Calculate the total sum of `lon`. Produce `#t` if the sum is greater than `k`, `#f` otherwise.

algorithm 2: If the first element of `lon` (call this `f`) is greater than `k`, produce `#t`, otherwise reduce `k` by `f` and then repeat for each remaining element. If no more elements remain, produce `#f`.

We can *implement* the two algorithms.

```
(define (tsum>? lon k)
  (> (foldl + 0 lon) k))

(define (rsum>? lon k)
  (cond [(empty? lon) #f]
        [(< k (first lon)) #t]
        [else (rsum>? (rest lon) (- k (first lon)))]))
```

Both algorithms solve the problem.

How do we determine which one is “better”?

What do we mean by “better”?

How do we **compare** algorithms?

There are many objective and subjective methods for comparing algorithms:

- How easy is it to understand?
- How easy is it to implement?
- How robust is it?
- How accurate is it?
- 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 ***time efficiency***, or **how long** it takes an algorithm to solve a problem. Unless we specify otherwise, we **always mean *time efficiency***.

Another efficiency measure is ***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 *efficiency* of an algorithm may depend on its *implementation*.

To avoid any confusion, we always measure the efficiency of a *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? Is the 0.037 relevant?

Measuring *running times* in seconds can be problematic.

What are the alternatives?

In Racket, we can measure the total number of (substitution) **steps** required to apply a function.

```
   (rsum>? '(10 5 1) 11)
=> ...                         ;; skipping 9 steps
=> (rsum>? '(5 1) 1)
=> (cond [(empty? '(5 1)) #f]
         [(< 1 (first '(5 1))) #t]
         [else (rsum>? (rest '(5 1)) (- 1 (first '(5 1))))])
=> (cond [#f  #f] [(< 1 (first '(5 1))) #t] ...)
=> (cond [(< 1 (first '(5 1))) #t] ...)
=> (cond [(< 1 5) #t] ...)
=> (cond [#t #t] ...)
=> #t                          ;; 16 total steps
```

We have to use caution when measuring Racket steps, as some built-in functions can be deceiving. For example, the built-in functions `foldl` and `last` may *appear* to only require one substitution step, but we must consider how the functions are internally implemented and how many “hidden” steps there are.

We revisit this issue later.

Do not worry about *precisely* calculating the number of steps in a Racket expression. **You are not expected to count exact the number of steps in an expression.**

We introduce some simplification shortcuts soon.

In C, one measure might be how many *machine code* instructions are executed. Unfortunately, this is very hard to measure and is highly dependent on the machine and the environment.

A popular efficiency measurement in C is the number of **operators** executed, or ***operations***.

For consistency, we also call each C operation a *step*.

```
sum = 0;             // 1
i = 0;               // 1
while (i < 5) {      // 6
  sum = sum + i;     // 10
  i = i + 1;         // 10
}
```

Like counting Racket steps, this can be a little tedious.

Input size

What is the *running time* (number of steps) required for this implementation of `sum`?

```
;; (sum lon) finds the sum of all numbers in lon
(define (sum lon)
  (cond [(empty? lon) 0]
        [else (+ (first lon) (sum (rest lon)))]))
```

The running time **depends on the length** of the list (`lon`).

If there are n items in the list, it requires $7n + 2$ steps.

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 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 **T**ime* 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 important in addition to size.

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.

Worst case analysis

Consider the running time of our `tsum>?` implementation (using the `sum` helper function instead of the deceiving `foldl`).

```
  (define (sum lst)
    (cond [(empty? lst) 0]
          [else (+ (first lst) (sum (rest lst)))]))

  (define (tsum>? lon k)
    (> (sum lon) k))
```

The running time of `tsum>?` is $T(n) = 7n + 4$, where n is the length of the list.

Now consider the running time of our `rsum>?` implementation.

```
(define (rsum>? lon k)
  (cond [(empty? lon) #f]
        [(< k (first lon)) #t]
        [else (rsum>? (rest lon) (- k (first lon)))]))
```

Unfortunately, without knowing `k` and `lon`, we cannot determine how many steps are required.

```
(rsum>? '(10 9 8 7 6 5 4 3 2 1)  1)  ;;   5 steps
(rsum>? '(10 9 8 7 6 5 4 3 2 1) 60)  ;; 102 steps
```

For `rsum>?`, the ***best case*** is when *only the first element* in the list is "visited" and the ***worst case*** is when *all* of the list elements are visited.

For `tsum>?`, the best case is the same as the worst case.

`rsum>?`	$T(n) = 5$	(best case)
	$T(n) = 10n + 2$	(worst case)
`tsum>?`	$T(n) = 7n + 4$	(all cases)

Which implementation is more efficient?

Is it more "fair" to compare against the best case or the worst case?

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, the `tsum>?` implementation $(7n + 4)$ is more efficient than `rsum>?` $(10n + 2)$.

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 $(7n + 4)$ and $(10n + 2)$.

We are interested in the ***order*** of a running time. The order is the "dominant" term in the running time with any constant coefficients removed.

The dominant term in both $(7n + 4)$ and $(10n + 2)$ is n, and so they are both "*order* n".

To represent *orders*, we use ***Big O notation***. Instead of "*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

- $2013 = O(1)$
- $100000 + n = O(n)$
- $n + n \log n = O(n \log n)$
- $999n + 0.01n^2 = O(n^2)$
- $\frac{n(n+1)(2n+1)}{6} = O(n^3)$
- $n^3 + 2^n = O(2^n)$

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 `tsum>?` and `rsum>?` 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.

With experience and intuition, determining the order becomes second nature: *“clearly, the running time of* `sum` *is* $O(n)$*”*

```
(define (sum lon)
  (cond [(empty? lon) 0]
        [else (+ (first lon) (sum (rest lon)))]))
```

In this course, our goal is to give you experience and work toward building your intuition. We also introduce some helpful introductory tools to help perform analysis.

Analyzing simple functions

First, consider **simple** functions (without recursion or iteration).

```
int max(int a, int b) {
  if (a > b) return a;
  return b;
}
```

In C, all operations (operator executions) are $O(1)$.

Without iteration or recursion, there must be a fixed number of operators, so the running time of all operations is

$$O(1) + O(1) + \ldots + O(1) = O(1)$$

For a simple function that calls no other functions, the running time is $O(1)$, otherwise it depends on the other functions called.

If a simple function `f` calls `g` and `h`, the running time of `f` is

$$T_f(n) = O(1) + T_g(n) + T_h(n)$$

which we know is

$$T_f(n) = \max(T_g(n), T_h(n))$$

If the parameters of `f` determine which functions are called, remember to use the **worst case**.

```
int f(int n) {
  if (n % 2) {
    return fast(n);
  } else {
    return slow(n);
  }
}
```

$$T_f(n) = T_{slow}(n)$$

Consider the following two implementations.

```
;; (single? lst) determines if lst
;;   has exactly one element

(define (a-single? lst)
  (= 1 (length lst)))

(define (b-single? lst)
  (and (not (empty? lst)) (empty? (rest lst))))
```

The running time of a is $O(n)$, while the running time of 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.

Racket running times (lists)

$O(1)$: `cons cons? empty empty? rest`

`first second...tenth`

$O(n)$: `length last reverse member`$^\diamond$ `remove`$^\diamond$

`list-ref`^{i} `drop-right`^{i} `take-right`^{i} `append`^{1}

`filter`* `map`* `foldl`* `foldr`* `build-list`* `apply`*

$O(n \log n)$: `sort`

$^\diamond$: we discuss `member` later (`remove` is similar)

^{i}: $O(i)$, where i is the position, *e.g.*, `(list-ref lst i)`

^{1}: where n is the length of the *first* list (being appended to)

*: when used with a $O(1)$ function, *e.g.*, `(filter even? lst)`

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 a list (*e.g.*, `list-ref`) is $O(i)$.

To access the i-th element in an array (*e.g.*, `a[i]`) is always $O(1)$.

Racket has a *vector* data type that is very similar to arrays in C.

```
  (define v (vector 4 8 15 16 23 42))
```

Like C's arrays, any element of a vector can be accessed by the `vector-ref` function in $O(1)$ time.

For the list construction function `list` or the mathematical function `max`, you might expect their running times to be $O(n)$.

However, these functions do not accept lists, but rather a finite (constant) number of parameters.

```
(list 1 2 3 4 5 6 7 8 9 10)
(max 4 8 15 16 23 42)
```

Because the number of parameters is a constant, their running times are $O(1)$.

Racket running times (numeric)

In C, all arithmetic operations are $O(1)$ and in this course we assume that all built-in numeric functions in Racket are $O(1)$.

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 the story is a 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 (equality)

We assume `=` (numeric arguments) 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`. `(remove e lst)` is similarly $O(mn)$.

* This highlights another difference between symbols & strings.

C running times

The <`string.h`> library functions `strlen`,`strcpy` and `strcmp` are $O(n)$, where n is the length of the string (or the smallest string for `strcmp`).

The <`stdio.h`> library functions `printf` and `scanf` are $O(1)$ except when working with strings (`"%s"`), in which case they 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")`).

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:

$T(n) = O(1) + T(n - k_1)$	$= O(n)$
$T(n) = O(n) + T(n - k_1)$	$= O(n^2)$
$T(n) = O(n^2) + T(n - k_1)$	$= O(n^3)$
$T(n) = O(1) + T(\frac{n}{k_2})$	$= O(\log n)$
$T(n) = O(1) + k_2 \cdot T(\frac{n}{k_2})$	$= O(n)$
$T(n) = O(n) + k_2 \cdot T(\frac{n}{k_2})$	$= O(n \log n)$
$T(n) = O(1) + T(n - k_1) + T(n - k_1')$	$= O(2^n)$

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)

Examples: recurrence relations

For simplicity and convenience (and to avoid any `equal?` issues) we use lists of integers in these examples.

```
(define (member1 e lon)
  (cond [(empty? lon) #f]
        [(= e (first lon)) #t]
        [else (member1 e (rest lon))]))
```

$$T(n) = O(1) + T(n-1) = O(n)$$

```
(define (member2 e lon)
  (cond [(zero? (length lon)) #f]
        [(= e (first lon)) #t]
        [else (member2 e (rest lon))]))
```

$$T(n) = O(n) + T(n-1) = O(n^2)$$

```
(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))
```

$$T(n) = O(n)$$

Although not common, we can use recursion with arrays.

```
int rsum_array(const int *a, int len) {
  if (len == 0) {
    return 0;
  }
  return a[0] + rsum_array(a + 1, len - 1);
}
```

$$T(n) = O(1) + T(n-1) = O(n)$$

With arrays, it is much more common to use **iteration**.

Iterative analysis

iterative analysis uses **summations**, not *recurrence relations*.

```
for (i = 1; i <= n; ++i) {
  printf("*");
}
```

$$T(n) = \sum_{i=1}^{n} O(1) = \underbrace{O(1) + \ldots + O(1)}_{n} = n \times O(1) = O(n)$$

Because we are primarily interested in *orders*,

$\sum_{i=0}^{n-1} O(x)$, $\sum_{i=1}^{10n} O(x)$, or $\sum_{i=1}^{\frac{n}{2}} O(x)$ are equivalent* to $\sum_{i=1}^{n} O(x)$

* unless x is exponential (*e.g.*, $O(2^n)$).

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

```
sum = 0;
for (i = 0; i < n; ++i) {
  sum += i;
}
```

$$\sum_{i=1}^{n} O(1) = O(n)$$

Common summations

$$\sum_{i=1}^{\log n} O(1) = O(\log n)$$

$$\sum_{i=1}^{n} O(1) = O(n)$$

$$\sum_{i=1}^{n} O(n) = O(n^2)$$

$$\sum_{i=1}^{n} O(i) = O(n^2)$$

$$\sum_{i=1}^{n} O(i^2) = O(n^3)$$

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.

```
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.

$$\sum_{i=1}^{n} O(1) = O(n)$$

When the loop counter changes *geometrically*, the number of iterations is often logarithmic.

```
k = n;                 // n is size of the input
while (k > 0) {
  printf("*");
  k /= 10;
}
```

There are $\log_{10} n$ iterations.

$$\sum_{i=1}^{\log n} O(1) = O(\log n)$$

When working with *nested* loops, evaluate the *innermost* loop first.

```
for (j = 0; j < n; ++j) {
  for (i = 0; i < j; ++i) {
    printf("*");
  }
  printf("\n");
}
```

Inner loop: $\sum_{i=0}^{j-1} O(1) = O(j)$

Outer loop: $\sum_{j=0}^{n-1} (O(1) + O(j)) = O(n^2)$

```
bool member(int item, const int a[], int len) {
  for (int i=0; i < len; ++i) {
    if (a[i] == item) {
      return true;
    }
  }
  return false;
}
```

$$T(n) = O(1) + \sum_{i=1}^{n} O(1) = O(n)$$

```
bool has_duplicates(const int a[], int len) {
  for (int i=0; i < len - 1; ++i) {
    for (int j=i+1; j < len; ++j) {
      if (a[i] == a[j]) return true;
    }
  }
  return false;
}
```

Inner loop: $\sum_{j=i+1}^{n-1} O(1) = O(n-i)$

Outer loop: $\sum_{i=0}^{n-2} (O(1) + O(n-i)) = O(n^2)$

Note: $\sum_{i=1}^{n} (n-i) = \sum_{i=0}^{n-1} i = O(n^2)$

Do **NOT** put the `strlen` function within a loop.

```
int char_count(char c, const 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.

Sorting algorithms

No introduction to efficiency is complete without a discussion of **sorting algorithms**.

In this course we discuss several sorting algorithms in C (with arrays) and in Racket (with lists).

In this Section, we only sort **numbers**. Like the built-in Racket `sort` function, these sort functions would be more useful if they were provided with a comparison function. In Section 12 we discuss how this can be done in C.

In ***insertion sort***, we start with an empty (sorted) sequence, and then **insert** each element into the sorted sequence, maintaining the order after each insert. The Racket version has been seen in previous course(s).

```
(define (insert n slon)
  (cond
    [(empty? slon) (cons n empty)]
    [(<= n (first slon)) (cons n slon)]
    [ else (cons (first slon) (insert n (rest slon)))]))
```

$$T(n) = O(1) + T(n-1) = O(n)$$

```
(define (insertion-sort lon)
  (cond
    [(empty? lon) empty]
    [else (insert (first lon) (insertion-sort (rest lon)))]))
```

$$T(n) = O(n) + T(n-1) = O(n^2)$$

In the following C implementation of insertion sort, `a[0]...a[i-1]` is sorted, and then we look for the correct position `pos` to insert `a[i]` into, "shifting" all of the elements `a[pos]...a[i-1]` one position greater.

```
void insertion_sort(int a[], int len) {
  for (int i=1; i < len; ++i) {
    int val = a[i];
    int pos = i;
    while ((pos > 0) && (a[pos-1] > val)) {
      a[pos] = a[pos-1];
      --pos;
    }
    a[pos] = val;
  }
}
```

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} O(1) = O(n^2)$$

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.

```
(define (selection-sort lon)
  (cond [(empty? (rest lon)) lon]
        [else (define m (find-min lon))  ; O(n)
              (cons m (selection-sort (remove m lon)))]))
```

Selection sort is $O(n^2)$, even in the best case.

Note: Insertion sort is $O(n)$ in the best case (when the list is already sorted).

In this C implementation of selection sort, the elements `a[0]...a[i-1]` are sorted and we must find the position (`pos`) of the next smallest element, and then “swap” it with `a[i]`.

```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
  }
}
```

This C implementation is also $O(n^2)$.

Merge Sort

In ***merge sort***, the list is split into two separate lists. After the two lists are sorted they are `merge`d together.

This approach is known as ***divide and conquer***, where a problem is *divided* into two (or more) smaller problems. Once the smaller problems are completed (*conquered*), the results are combined to solve the original problem.

We only present merge sort in Racket.

For *merge sort*, we need a function to `merge` two sorted lists.

```
(define (merge slon1 slon2)
  (cond
    [(empty? slon1) slon2]
    [(empty? slon2) slon1]
    [(< (first slon1) (first slon2))
     (cons (first slon1) (merge (rest slon1) slon2))]
    [else (cons (first slon2)
                (merge slon1 (rest slon2)))]))
```

If the size of the two lists are m and p, then the recursive calls are either $[(m - 1)$ and $p]$ or $[m$ 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 `merge-sort`.

```
(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 [(<= len 1) lon]
        [else (merge (merge-sort left)
                     (merge-sort right))]))
```

$$T(n) = O(n) + 2T(n/2) = O(n \log n)$$

The built-in Racket function `sort` uses `merge-sort`.

Quick Sort

In *quick sort*, an element is selected as a “pivot”. The list is then **divided** into two sublists: a list of elements *less than* (or equal to) the pivot and a list of elements *greater than* the pivot. Each sublist is sorted (**conquered**) and then appended together (along with the original pivot).

Quicksort is also known as partition-exchange sort or as Hoare’s quicksort (named after the author).

```
(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))]))
```

In our C implementation of quick sort, we:

- select the first element of the array as our “pivot”
- move (“swap”) 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 len, int first, int last) {

  if (last <= first) return; // size 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 + 1; --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, len, first, pos-1);
  quick_sort_range(a, len, pos+1, last);
}

void quick_sort(int a[], int len) {
  quick_sort_range(a, len, 0, len-1);
}
```

When the pivot is in “the middle” it splits the sublists equally, so

$$T(n) = O(n) + 2T(n/2) = O(n \log n)$$

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

Algorithm	best case	worst case
insertion sort	$O(n)$	$O(n^2)$
selection sort	$O(n^2)$	$O(n^2)$
merge sort	$O(n \log n)$	$O(n \log n)$
quick sort	$O(n \log n)$	$O(n^2)$

Binary search

Earlier we saw a C `member` function that is $O(n)$.

```
bool member(int item, int a[], int len) {
  for (int i=0; i < len; ++i) {
    if (a[i] == item) {
      return true;
    }
  }
  return false;
}
```

But what if the array was previously *sorted*?

We can use **binary search** to write an $O(\log n)$ `sorted-member` function.

```
bool sorted_member(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 true;
    } else if (a[mid] < item) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return false;
}
```

The range (`high-low`) starts out at n (`len`) and decreases by $\frac{1}{2}$ each iteration, so the running time is $O(\log n)$.

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 $g(n)$.

$$n^2 \in O(n^{100})$$
$$n^3 \in O(2^n)$$

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.

A slightly more formal definition of Big O is

$$\boxed{\begin{array}{l} f(n) \in O(g(n)) \Leftrightarrow f(n) \le c \cdot g(n) \\ \text{for large } n \text{ and some positive integer } c \end{array}}$$

This definition makes it clear why we *“ignore”* constant coefficients.

For example,

$9n \in O(n)$ for $c = 10,\quad 9n \le 10n$

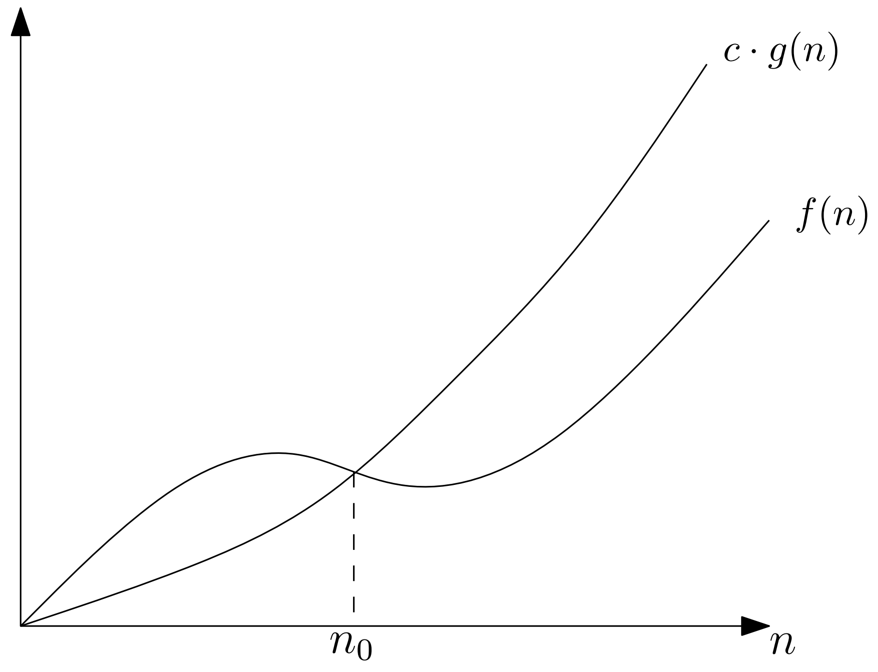
and

$0.01n^3 + 1000n^2 \in O(n^3)$
for $c = 1001,\quad 0.01n^3 + 1000n^2 \le 1001n^3$

The full definition of Big O is

$$\boxed{f(n) \in O(g(n)) \Leftrightarrow \exists c, n_0 > 0, \forall n \geq n_0, f(n) \leq c \cdot g(n)}$$

$f(n)$ is in $O(g(n))$ if there exists a positive c and n_0 such that for any value of $n \geq n_0$, $f(n) \leq c \cdot g(n)$.



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 $O(n^2)$.

In later CS courses, the formal definition of Big O is used to *prove* algorithm behaviour more rigourously. In this course, we only expect a basic understanding of the asymptotic nature of Big O.

There are other asymptotic functions in addition to Big O.

$$f(n) \in \omega(n) \Leftrightarrow c \cdot g(n) < f(n)$$
$$f(n) \in \Omega(n) \Leftrightarrow c \cdot g(n) \leq f(n)$$
$$f(n) \in \Theta(n) \Leftrightarrow c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$
$$f(n) \in O(n) \Leftrightarrow f(n) \leq c \cdot g(n)$$
$$f(n) \in o(n) \Leftrightarrow f(n) < c \cdot g(n)$$

$O(n)$ is often used when $\Theta(n)$ is more appropriate.

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.

```
;; merge-sort: (listof Int) -> (listof Int)
;; time: O(n*logn), n is the length of lon
(define (merge-sort lon) ...)
```

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.

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum (rest lst)))]))

(define (asum lst)
  (define (asum/acc lst sofar)
    (cond [(empty? lst) sofar]
          [else (asum/acc (rest lst)
                          (+ (first lst) sofar))]))
  (asum/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 `asum` uses *accumulative* recursion.

If we examine the substitution steps of `sum` and `asum`, we get some insight into their differences.

```
(sum '(1 1 1))
=> (+ 1 (sum '(1 1)))
=> (+ 1 (+ 1 (sum '(1))))
=> (+ 1 (+ 1 (+ 1 (sum empty))))
=> 3

(asum '(1 1 1))
=> (asum/acc '(1 1 1) 0)
=> (asum/acc '(1 1) 1)
=> (asum/acc '(1) 2)
=> (asum/acc empty 3)
=> 3
```

The `sum` expression “grows” and eventually has $O(n)$ +’s before the final result can be calculated. However, the `asum` expression does not grow and is $O(1)$.

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.

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` or `length`
- 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

Readings: CP:AMA 17.1, 17.2, 17.3, 17.4

Dynamic memory

When defining a C array, the length of the array must be known, but Racket lists can grow to be arbitrarily large without knowing in advance the length of the list.

Racket lists are `cons`tructed with **Dynamic memory**.

Dynamic memory is allocated from the **heap** *while the program is running* (more on this later).

The heap is the final section of memory in our memory model.

In Section 11 we use dynamic memory to construct Racket-like lists in C.

Maximum-length arrays

Before we introduce dynamic memory, we discuss a less sophisticated alternative.

In some applications, it may be "appropriate" to have a **maximum length** for an array. For example, in Section 08 we used `scanf` to store a name in an array with a maximum number of characters.

In general, maximums should only be used when appropriate. They can be very wasteful if the maximum is excessively large.

Many "real-world" systems have maximums. UW login ids are restricted to 8 characters and last names are restricted to `40`.

When working with maximum-length arrays, we need to keep track of

- the "**actual**" **length** of the array, and
- the **maximum possible length**.

```
const int numbers_max = 100;
int numbers[100];
int numbers_len = 0;

void append_number(int i) {
  assert(numbers_len < numbers_max);
  numbers[numbers_len] = i;
  numbers_len++;
}
```

This approach does not use dynamic memory, but the length of the array is "dynamic" (can change during run-time).

What if the maximum length is exceeded?

- An error message can be displayed.
- The program can `exit`.
- A special return value can be used.

```
// returns true if i was successfully appended,
// false otherwise
bool append_number(int i) {
  if (numbers_len == numbers_max) return false;
  numbers[numbers_len] = i;
  numbers_len++;
  return true;
}
```

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.

```
#include <stdlib.h>

// effects: prints a message and exit upon failure
void append_number(int i) {
  if (numbers_len == numbers_max) {
    printf("FATAL ERROR: numbers_max exceeded\n");
    exit(EXIT_FAILURE);
  }
  //...
}
```

The `exit` argument is the same as the `main return` value.

For convenience, <`stdlib.h`> defines `EXIT_SUCCESS` (`0`) and `EXIT_FAILURE` (non-zero).

To make our maximum-length arrays more convenient and our code more re-usable, we can store the array information in a structure.

```
struct max_array {
  int *data;
  int len;
  int max;
};

void append_to_max_array(struct max_array *ma, int i) {
  if (ma->len < ma->max) {
    ma->data[ma->len] = i;
    ma->len++;
  } else { ... } // error
}

//example:
const int numbers_max = 100;
int numbers_data[100];
struct max_array numbers = {numbers_data, 0, numbers_max};
```

The heap

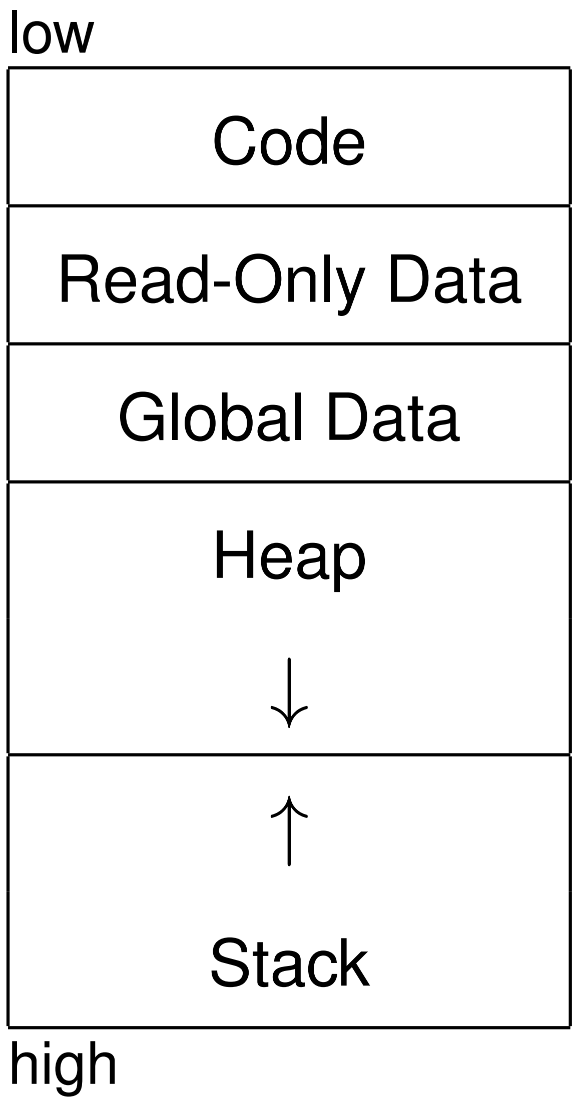
The ***heap*** is the final section in the C memory model.

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 reused. We call this ***deallocation***.

If too much memory has already been allocated, attempts to borrow additional memory fail.


low
Code
Read-Only Data
Global Data
Heap
↓
↑
Stack
high

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” stuck.

As we see later, there is also an *abstract data type* known as a stack, but because its behaviour is similar to “the stack”, its name is far less confusing.

malloc

The `malloc` (**m**emory **alloc**ation) function obtains memory from the heap dynamically (it is part of <`stdlib.h`>).

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

`size_t` is the type of integer produced by the `sizeof` operator. You should always use `sizeof` with malloc to improve portability and to improve communication.

```
int *pi = malloc(sizeof(int));
struct posn *pp = malloc(sizeof(struct posn));
```

Strictly speaking, `size_t` and `int` are different types.

`Seashell` allows `malloc(4)` instead of `malloc(sizeof(int))`, but the latter is much better style and is more portable.

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`.

`malloc` returns an address of type (`void *`) (*void pointer*) which can be assigned to a pointer variable of any type. Thus any type can be stored in the heap.

In this course, you should not run out of memory, but in practice it's good style to check every `malloc` return value and gracefully handle a `NULL` instead of crashing.

```
int *p = malloc(sizeof(int));
if (p == NULL) {
  printf("sorry dude, out of memory! I'm exiting.\n");
  exit(EXIT_FAILURE);
}
```

Unless otherwise noted, you do **not** have to check for a `NULL` return value in this course. The check is often omitted in these notes.

The memory on the heap returned by `malloc` is **uninitialized**.

```
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

```
// free(p) returns memory at p back to the heap
// requires: p must be from a previous malloc
// effects:  the memory at p is no longer valid
void free(void *p);
```

For every block of memory obtained through `malloc`, you should eventually `free` the memory (when the memory is no longer in use).

You can assume that `free` is $O(1)$.

In the `Seashell` environment, you **must** `free` every block.

Once a block of memory is `free`d, it cannot be accessed (and can not be `free`d a second time). It may be returned by a future `malloc` and become valid again.

memory sections

```
const int r = 42;
int g = 15;
int main(void) {
  int s = 23;
  int *h = malloc(sizeof(int));
  *h = 16;
  printf("the address of main is:  %p\n", main); // CODE
  printf("the address of    r is:  %p\n", &r);   // READ-ONLY
  printf("the address of    g is:  %p\n", &g);   // GLOBAL DATA
  printf("the value   of    h is:  %p\n", h);    // HEAP
  printf("the address of    s is:  %p\n", &s);   // STACK
  free(h);
}
```

```
the address of main is:  0x46c060
the address of r is:     0x477700
the address of g is:     0x68a9e0
the value of h is:       0x60200000eff0
the address of s is:     0x7fff05ffc570
```

Memory leaks

A memory leak occurs when allocated memory is not eventually `free`d.

Programs that leak memory may suffer degraded performance or eventually crash.

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

In this example, the address from the original `malloc` has been overwritten (it is lost) and so it can never be `free`d.

Invalid after free

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

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

```
int *p = malloc(sizeof(int));
free(p);
int k = *p;   // INVALID
*p = 42;      // INVALID
free(p);      // INVALID
p = NULL;     // GOOD STYLE
```

It is often good style to assign `NULL` to a `free`d pointer variable to avoid misuse.

Garbage collection

In Racket, we are not concerned with `free`ing memory when it is no longer needed because Racket has a ***garbage collector***. Many modern languages have a garbage collector.

A garbage collector detects when memory is no longer in use and automatically returns the memory to the heap.

The biggest disadvantage of a garbage collector is that it can affect performance, which is a concern in high performance computing.

Many programmers believe the benefits of a garbage collector outweigh any disadvantages (many also believe that it is important to learn how to program without a garbage collector).

Dynamically allocating arrays

We can create a ***dynamic array*** in the heap.

```
// effects: allocates an array on heap (caller must free)
//          (description of error behaviour...)
int *create_heap_array(int len) {
  assert(len > 0);
  int *a = malloc(sizeof(int) * len);     // array size
  if (a == NULL) { ... }                 // can error check!
  return a;     // array can exist beyond function call
}
```

One of the key advantages of dynamic (heap) memory is that a function can "create" new memory that persists **after** the function has `return`ed.

The caller is usually responsible for `free`ing the memory (the contract should communicate this).

The <string.h> function strdup duplicates a string by creating a new dynamically allocated array.

```
// my_strdup(s) makes a duplicate of s
//   returns NULL if there is not enough memory
// effects: allocates memory: caller must free
char *my_strdup(const char *s) {
  char *new = malloc(sizeof(char) * (strlen(s) + 1));
  strcpy(new,s);
  return new;
}
```

Recall that the strcpy(dest,src) copies the characters from src to dest, and that the dest array must be large enough.

strdup is not officially part of the C standard, but common.

Resizing arrays

`malloc` is passed the size of the block of memory to be allocated.

On its own, this does not 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
- `free`ing the old array

Usually this is used to make a *larger* array, but if a smaller array is requested the extra elements are discarded.

```
// resize_array(old, oldlen, newlen) changes the length
//   of array old from oldlen to newlen
//   returns new array or NULL if out of memory
//   if larger, new elements are uninitialized
// requires: old must be a malloc'd array of size oldlen
// effects:  frees the old array, caller must free new array
// time: O(n), where n is min(oldlen,newlen)

int *resize_array(int *old, int oldlen, int newlen) {
  int *new = malloc(sizeof(int) * newlen);
  int copylen = oldlen;
  if (newlen < oldlen) copylen = newlen;
  for (int i=0; i < copylen; i++) {
    new[i] = old[i];
  }
  free(old);
  return new;
}
```

To make resizing arrays easier, there is a `realloc` library function.

```
void *realloc(void *ptr, size_t s);
```

The `ptr` parameter in `realloc` must be a value returned from a **previous** `malloc` (or `realloc`) call.

Similar to our `resize_array`, `realloc` preserves the contents from the previous `malloc`, discarding the memory if `s` is smaller, and providing *uninitialized* memory if `s` is larger.

For this course, you can assume that `realloc` is $O(s)$.

`realloc(NULL,s)` behaves the same as `malloc(s)`.

`realloc(ptr,0)` behaves the same as `free(ptr)`.

`realloc` makes our `resize_array` more straightforward:

```
//  (oldlen is no longer required)
int *resize_array(int *old, int newlen) {
  return realloc(old, sizeof(int) * newlen);
}
```

The return pointer of `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 `free`d and is now **invalid**. It's common to overwrite the old pointer with the new value.

```
my_array = realloc(my_array, newsize);
```

In practice,

```
  my_array = realloc(my_array, newsize);
```

could cause a memory leak.

In C99, if an “out of memory” condition occurs during `realloc`, the return value of `realloc` is `NULL` and `realloc` does not `free` the original memory block.

Amortized analysis

Consider the following code to dynamically increase an array every time a number is added:

```
int *numbers = NULL;
int numbers_len = 0;

void append_number(int i) {
  if (numbers_len == 0) {
    numbers = malloc(sizeof(int));
  } else {
    numbers = realloc(numbers, sizeof(int) * (numbers_len + 1));
  }
  numbers[numbers_len] = i;
  numbers_len++;
}
```

The time required to add n numbers is $\sum_{i=1}^{n} O(i) = O(n^2)$.

The previous `append_number` function called `realloc` *every* time a number was added.

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. We can use a structure to keep track of the lengths.

```
struct dyn_array {
  int *data;
  int len;
  int max;
};
```

```
struct dyn_array {
  int *data;
  int len;
  int max;
};

void append_number(struct dyn_array *da, int i) {
  if (da->len == da->max) {
    if (da->max == 0) {
      da->max = 1;
      da->data = malloc(sizeof(int));
    } else {
      da->max *= 2;
      da->data = realloc(da->data, sizeof(int) * da->max);
    }
  }
  da->data[da->len] = i;
  da->len++;
}

// definition example
struct dyn_array numbers = {NULL, 0, 0};
```

With our “doubling” strategy, calls to `append_number` are $O(n)$ when resizing is necessary, and $O(1)$ otherwise.

Element-copying cost for first 33 calls to `append_number` would be:

0,1,2,0,4,0,0,0,8,0,0,0,0,0,0,0,16,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,32

For $n+1$ calls to `append_number`, the total copying cost is:

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 = 2n - 1 = O(n).$$

If we add 1 to each call for assigning the new value, the *total time* for n calls to `append_number` is

$$O(n) + O(n) = O(n).$$

Thus, the ***amortized*** (“averaged”) run-time of `append_number` is

$$O(n)/n = O(1).$$

You will use *amortized* analysis in CS 240 and in CS 341.

For arrays that can also *"shrink"* (with deletions), a popular strategy is to reduce the size when the length reaches $\frac{1}{4}$ of the capacity. Although more complicated, this also has an *amortized* run-time of $O(1)$ for an arbitrary sequence of inserts and deletions.

Some languages have a built-in resizable array (often called a `vector`) that uses a similar "doubling" strategy.

example: create & destroy a dynamic array

```
struct dyn_array *create_dyn_array(int initial_max) {
  struct dyn_array *da = malloc(sizeof(struct dyn_array));
  da->len = 0;
  da->max = initial_max;
  if (da->max > 0) {
    da->data = malloc(sizeof(int) * initial_max);
  } else {
    da->data = NULL;
  }
  return da;
}

void destroy_dyn_array(struct dyn_array *da) {
  if (da->max > 0) {
    free(da->data);
  }
  free(da);
}
```

Earlier we discussed how `struct` function parameters are typically "passed by pointer" to avoid copying the contents of the structure.

For a similar reason, functions also rarely `return` structures.

Typically, either:

- the function creates a **new** `struct` with `malloc` and returns a pointer (see the previous example),

  ```
  struct posn *make_me_a_new_posn(...) {...}
  ```

- or the calling function passes (a pointer to) an *existing* `struct` for the function to update (or initialize).

  ```
  void init_my_posn(struct posn *p, ...) {...}
  ```

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 the `exit` function if a program runs out of memory
- describe memory leaks, how they occur, and how to prevent them
- explain the advantages of creating an array in the heap instead of the stack

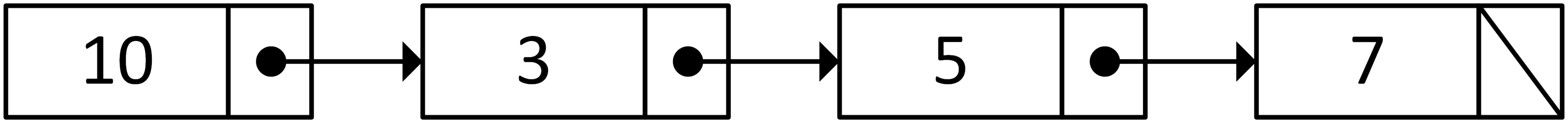
- describe the maximum-length array strategy, and how a similar strategy can be used to manage dynamic arrays to achieve an amortized $O(1)$ run-time for appends
- create dynamic resizable arrays in the heap
- write functions that create and return a new `struct`
- document dynamic memory side-effects 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*** **(first)** and a *link* to the ***next*** node **(rest)**.

There is no "official" way of implementing a linked list in C.

In this unit we present a typical linked list structure and several strategies for working with this structure.

In general, a linked list is a pointer to a *linked list node* (`llnode`).

```
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.

If a linked list is a pointer to a node, then an empty list is represented by a `NULL` pointer.

In the following slides we present both a **functional** (Racket-like) approach and an **imperative** (C-like) approach to working with linked lists. They are not mutually exclusive and can be combined (they both use `llnode`s).

The two approaches are not formal distinctions, but they help illustrate the respective programming paradigms.

We also present a **wrapper** strategy and an **augmentation** strategy that can be combined and used with either approach.

Functional approach

Helper functions to create a *“functional”* atmosphere:

```
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 the functional approach is the `cons` function.

In Racket, `cons` acquires dynamic memory (similar to C's `malloc`).

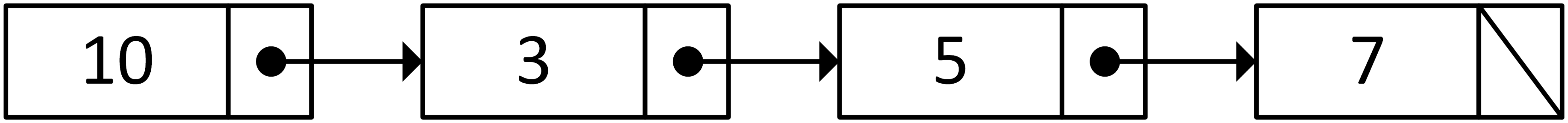
Our C `cons` `return`s a **new** node that *links* to the rest.

```
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.

```
struct llnode *my_list = cons(10, cons(3, cons(5,
                                   cons(7, empty()))));
```



We can also use cons in different ways (*e.g.*, with mutation).

```
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 the functional approach, we can write recursive functions that closely mirror their Racket equivalents.

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

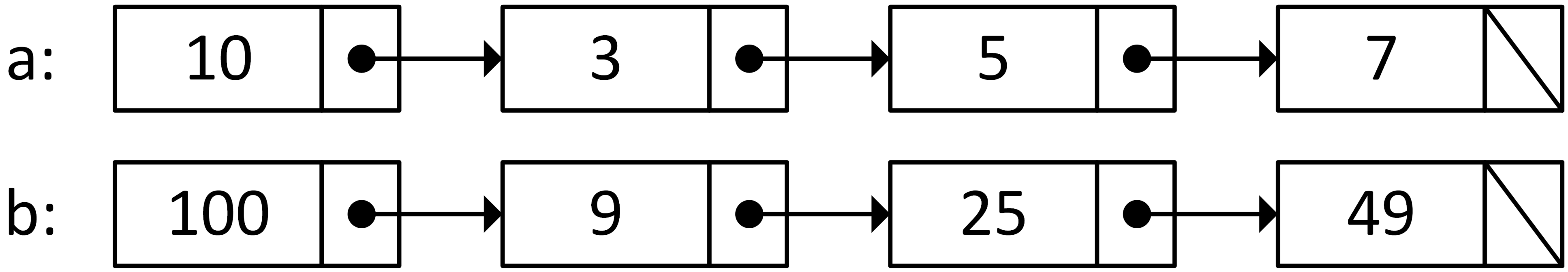
```
int length(struct llnode *lst) {
  if (is_empty(lst)) {
    return 0;
  }
  return 1 + length(rest(lst));
}
```

It is also possible to write some functions iteratively.

```
int length_iterative(struct llnode *lst) {
  int length = 0;
  while (!is_empty(lst)) {
    length++;
    lst = rest(lst);
  }
  return length;
}
```

In Racket, *and the functional programming paradigm*, most functions that **produce** a list construct a **new list**.

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

(define a '(10 3 5 7))
(define b (sqr-list a))
```



A C function written with a functional approach also `return`s a **new list**.

```
struct llnode *sqr_list(struct llnode *lst) {
  if (is_empty(lst))  return empty();
  return cons(first(lst) * first(lst),
              sqr_list(rest(lst)));
}
```

As an exercise, try writing a non-recursive (iterative) version of `sqr_list` that returns a new list (it's tricky).

Since we have been working with C imperatively, it might now seem more "natural" that a `sqr_list` function would square (or *mutate*) each item in the list.

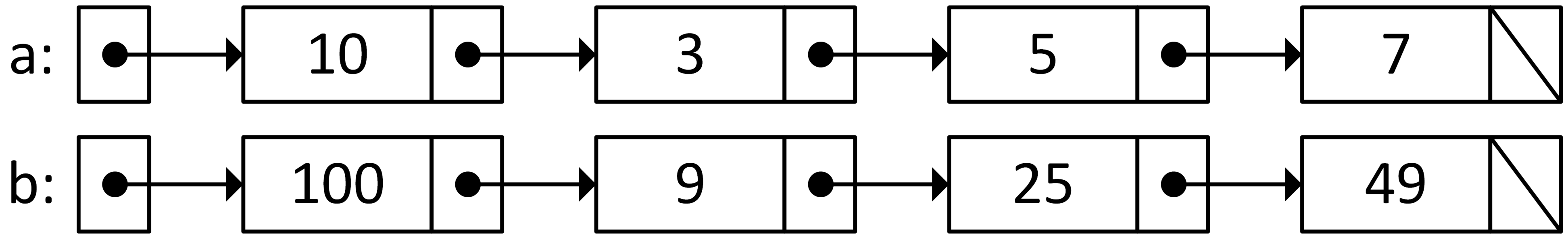
However, this is not a functional approach. We introduce an imperative version of `sqr_list` shortly.

In Racket, lists are immutable, and there is a special `mcons` function to generate a mutable list.

In the Scheme language, lists are mutable. This is one of the significant differences between Racket and Scheme.

To correctly use the `sqr_list` function, the result should be stored in a separate variable.

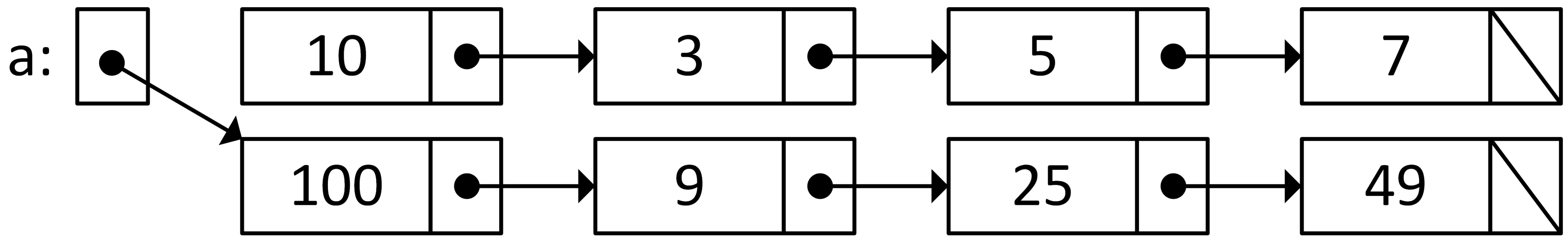
```
struct llnode *a = cons(10, cons(3, cons(5, cons(7, empty()))));
struct llnode *b = sqr_list(a);
```



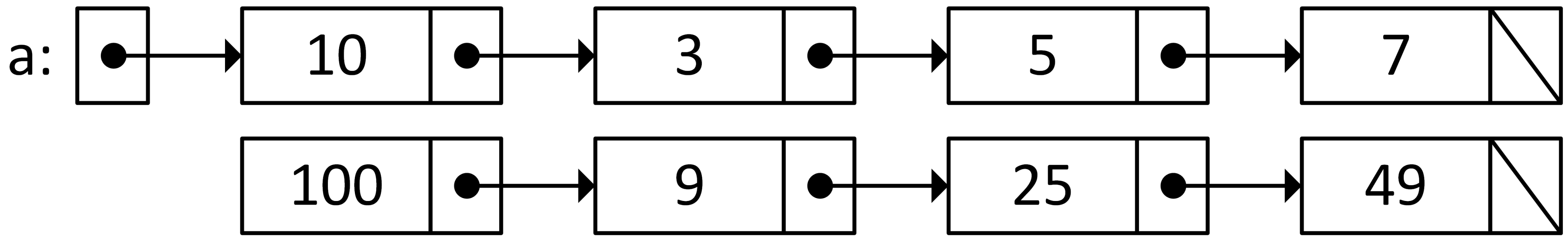
Unfortunately, if the function is misunderstood or used **incorrectly**, it can create a **memory leak**.

The following two statements each create a memory leak.

```
a = sqr_list(a);  // original list is lost
```



```
sqr_list(a);      // new list is lost
```



The best strategy to avoid a memory leak is to provide a clear contract.

The `cons` function can cause a similar problem:

```
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.

In Racket, we are not concerned about `free`ing a list. However, this is necessary in C.

```
void free_list(struct llnode *lst) {
  if (lst != NULL) {
    free_list(rest(lst));
    free(lst);
  }
}
```

Note that it is important to `free` the rest of the list before `free`ing the first node.

After `free(lst)`, any use of `lst` is invalid.

Both of these *iterative* `free_list` functions are equivalent.

```
void free_list_iterative_1(struct llnode *lst) {
  while (lst != NULL) {
    struct llnode *backup = lst;
    lst = rest(lst);
    free(backup);
  }
}

void free_list_iterative_2(struct llnode *lst) {
  while (lst != NULL) {
    struct llnode *backup = rest(lst);
    free(lst);
    lst = backup;
  }
}
```

This “backup” technique (using temporary pointers) is used more frequently when we introduce the imperative approach.

To further illustrate how `cons` is used, consider the `insert` from *insertion sort* (insert into a *sorted* list of numbers).

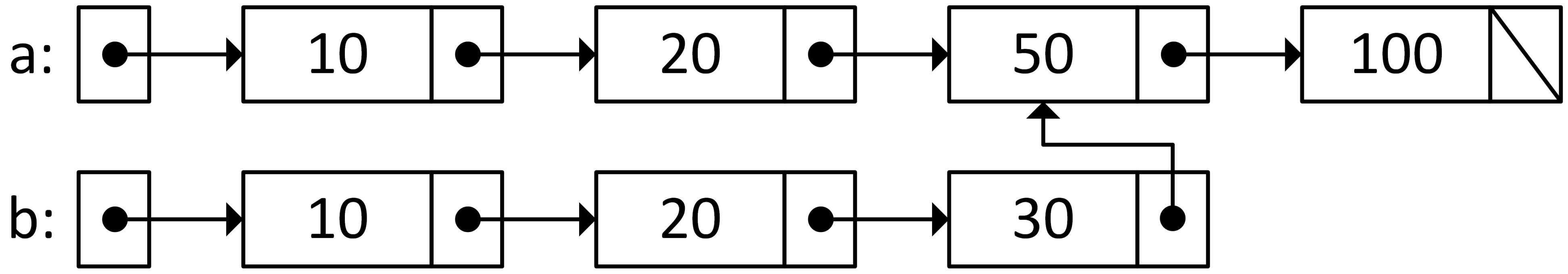
```
(define (insert n slst)
  (cond
    [(empty? slst) (cons n empty)]
    [(<= n (first slst)) (cons n slst)]
    [ else (cons (first slst) (insert n (rest slst)))]))
```

```
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 {
    return cons(first(slst), insert(n, rest(slst)));
  }
}
```

Consider this example:

```
struct llnode *a = cons(10,cons(20,cons(50,cons(100,empty()))));
struct llnode *b = insert(30, a);
```

The lists **share the last two nodes**.



Freeing either list causes an invalid memory access when the other is accessed.

```
free_list(a);
free_list(b);   // INVALID
```

This problem can occur with just the `cons` function.

```
struct llnode *a = cons(7, empty());
struct llnode *b = cons(5, a);
a = cons(3, a);

free_list(a);
free_list(b);  // INVALID
```

Once again, Racket's **garbage collector** automatically `free`s memory, so this is not a concern in Racket.

The functional approach works well for some applications, but is not well suited for environments where there are specific memory management issues and no garbage collection.

Imperative approach

In an "**imperative**" approach to linked lists, individual nodes are manipulated and changed (mutated) directly.

An imperative list function may *change* a list instead of generating a new list.

For example, the following `sqr_list_m` (for mutate) function *changes* the items in the list.

```
void sqr_list_m(struct llnode *lst) {
  while (lst != NULL) {
    lst->item *= lst->item;
    lst = lst->next;
  }
}
```

If a new list is desired, a *copy* of the original list can be made.

```
struct llnode *copy_list(struct llnode *lst) {
  if (lst == NULL) return NULL;
  return cons(lst->item, copy_list(lst->next));
}
```

The copy can then be changed.

```
struct llnode *orig = cons(10, cons(3, cons(5,
                             cons(7, empty()))));

struct llnode *dup = copy_list(orig);
sqr_list_m(dup);
```

As an alternative to the `cons` function, we write a function that inserts a new node at the start (front) of an existing list.

For example, we would like to have the following code sequence.

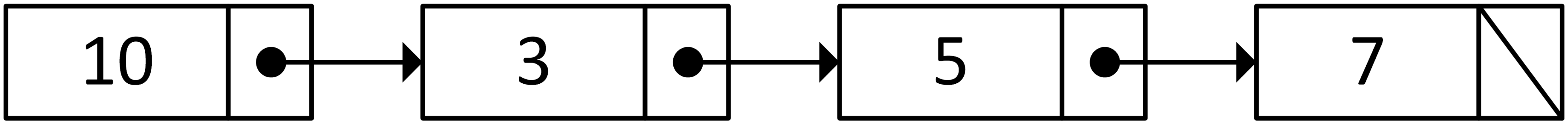
```
struct llnode *lst = NULL;

add_to_front(7, lst);   // won't work
add_to_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.

```
add_to_front(7, &lst);
add_to_front(5, &lst);
```

Since `lst` is already a pointer, we need to pass a **pointer to a pointer**.

```
void add_to_front(int n, struct llnode **ptr_front) {
  struct llnode *new = malloc(sizeof(struct llnode));
  new->item = n;
  new->next = *ptr_front;
  *ptr_front = new;
}
```



```
struct llnode *lst = NULL;
add_to_front( 7, &lst);
add_to_front( 5, &lst);
add_to_front( 3, &lst);
add_to_front(10, &lst);
```

With the imperative approach, nodes can also be *removed*.

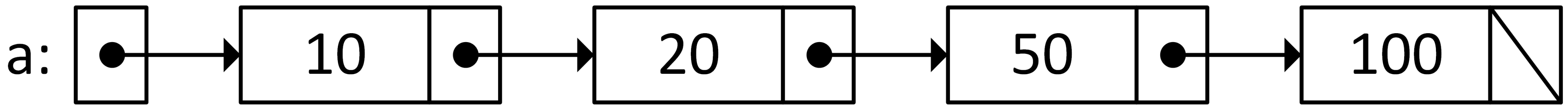
```
int remove_from_front(struct llnode **ptr_front) {
  struct llnode *front = *ptr_front;
  int retval = front->item;
  *ptr_front = front->next;
  free(front);
  return retval;
}
```

Instead of `return`ing nothing (`void`), it is more useful to `return` the value of the item being removed.

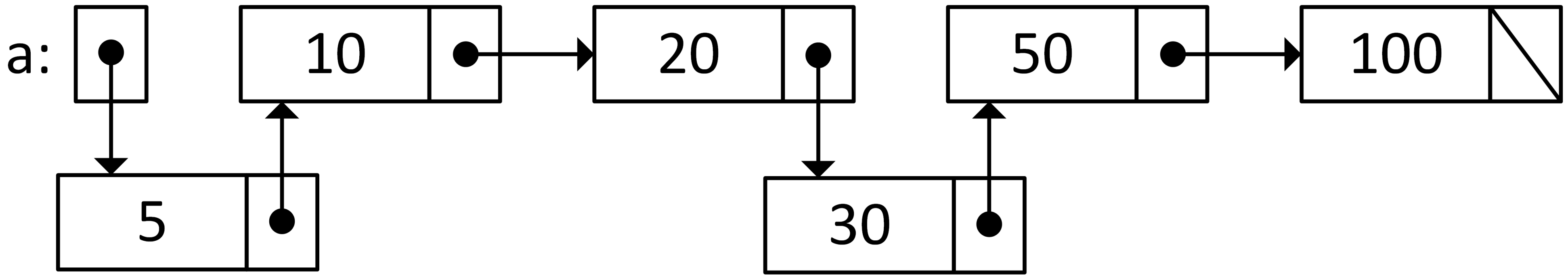
This function `inserts` a node into an *existing* sorted list.

```
void insert(int n, struct llnode **ptr_front) {
    struct llnode *cur = *ptr_front;
    struct llnode *prev = NULL;
    while ((cur != NULL) && (n > cur->item)) {
      prev = cur;
      cur = cur->next;
    }
    struct llnode *new = malloc(sizeof(struct llnode));
    new->item = n;
    new->next = cur;
    if (prev == NULL) { // new first node
      *ptr_front = new;
    } else {
      prev->next = new;
    }
  }
}
```

Revisiting the previous (functional) insert example,



```
insert( 5, &a);
insert(30, &a);
```



An alternative *"destructive"* approach uses a Racket-like programming interface (functions produce *new* lists), but each list passed to a function may be destroyed (`free`d).

```
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.

The imperative approach is still susceptible to misuse if we have pointers to the middle of the the list.

```
struct llnode *a = NULL;
add_to_front(5, &a);
struct llnode *b = a;
add_to_front(7, &a);
add_to_front(9, &b);
```

Freeing either list a or b makes the other list invalid.

Wrapper strategy

In the ***wrapper strategy***, we "**wrap**" each list so it is inside of another structure. The definition of a linked list changes slightly to be a pointer to this new structure, and not just a single node.

```
struct llist {
  struct llnode *front;
};
```

This may seem insignificant, but it has 3 advantages:

- a cleaner interface (avoids double pointers)
- less susceptible to misuse
- it can store **additional information**

A wrapper approach would typically have a “create” and a “destroy” function.

```
struct llist *create_list(void) {
  struct llist *lst = malloc(sizeof(struct llist));
  lst->front = NULL;
  return lst;
}

void destroy_list(struct llist *lst) {
  free_list(lst->front);
  free(lst);
}
```

Many of the previous functions introduced can be “wrapped” inside of another function.

```
void add_to_front(int n, struct llist *lst) {
  previous_add_to_front(n, &lst->front);
}
```

Overall, this provides a cleaner interface.

```
struct llist *lst = create_list();
add_to_front(5, lst);
add_to_front(7, lst);
destroy_list(lst);
```

A significant advantage of the wrapper approach is that **additional information** can be stored in the list structure.

The run time of the previous `length` functions are $O(n)$.

With the wrapper, we can store (or "cache") the length of *in the wrapper structure*, so the length can be retrieved in $O(1)$ time.

```
struct llist {
  struct llnode *front;
  int length;
};

// TIME: O(1)
int length(struct llst *lst) {
  return lst->length;
}
```

Naturally, other list functions would have to update the `length` when necessary.

```
struct llist *create_list(void) {
  struct llist *lst = malloc(sizeof(struct llist));
  lst->front = NULL;
  lst->length = 0;  // *****NEW
  return lst;
}

void add_to_front(int n, struct llist *lst) {
  previous_add_to_front(n, &lst->front);
  lst->length++;    // *****NEW
}
```

A common addition to a wrapper structure is a back pointer that points to the *last node* in the list.

This allows for an $O(1)$ add_to_back function.

```
void add_to_back(int n, struct llist *lst) {
  struct llnode *new = malloc(sizeof(struct llnode));
  new->item = n;
  new->next = NULL;
  if (lst->length == 0) { // empty list
    lst->front = new;
    lst->back = new;
  } else {
    lst->back->next = new;
    lst->back = new;
  }
  lst->length++;
}
```

Other functions would also have to be changed to update back.

By working with a *list* structure instead of directly working with *node* structures, there is less chance for misuse (*e.g.,* pointing to nodes in the middle).

However, the wrapper approach introduces entirely new ways that the information can be corrupted. For example, what if the `length` field does not accurately reflect the true length?

Whenever the same information is stored in more than one way, it is susceptible to *integrity* (consistency) issues.

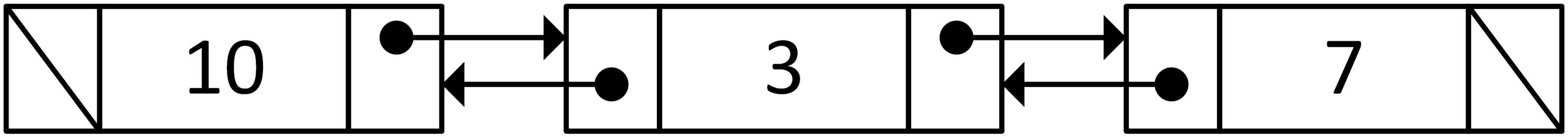
For example, a naïve user may think that the following statement removes all of the nodes from the list.

```
  lst->length = 0;
```

Augmentation strategy

In an **augmentation strategy**, each *node* is *augmented* to include additional information about the node or the structure.

The most common augmentation for a liked 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.



- The **functional** approach is appropriate when there is no mutation and extensive memory management is not required (*i.e.,* when there is a garbage collector),
- the **imperative** approach is appropriate when there is mutation or memory management is necessary,
- the **wrapper** strategy provides a cleaner interface and can be used to store additional information in a *wrapper structure*, and
- the **augmentation** strategy can be used to store additional information in *each node*.

These approaches and strategies also apply to other linked data structures (*e.g.,***trees**).

BSTs

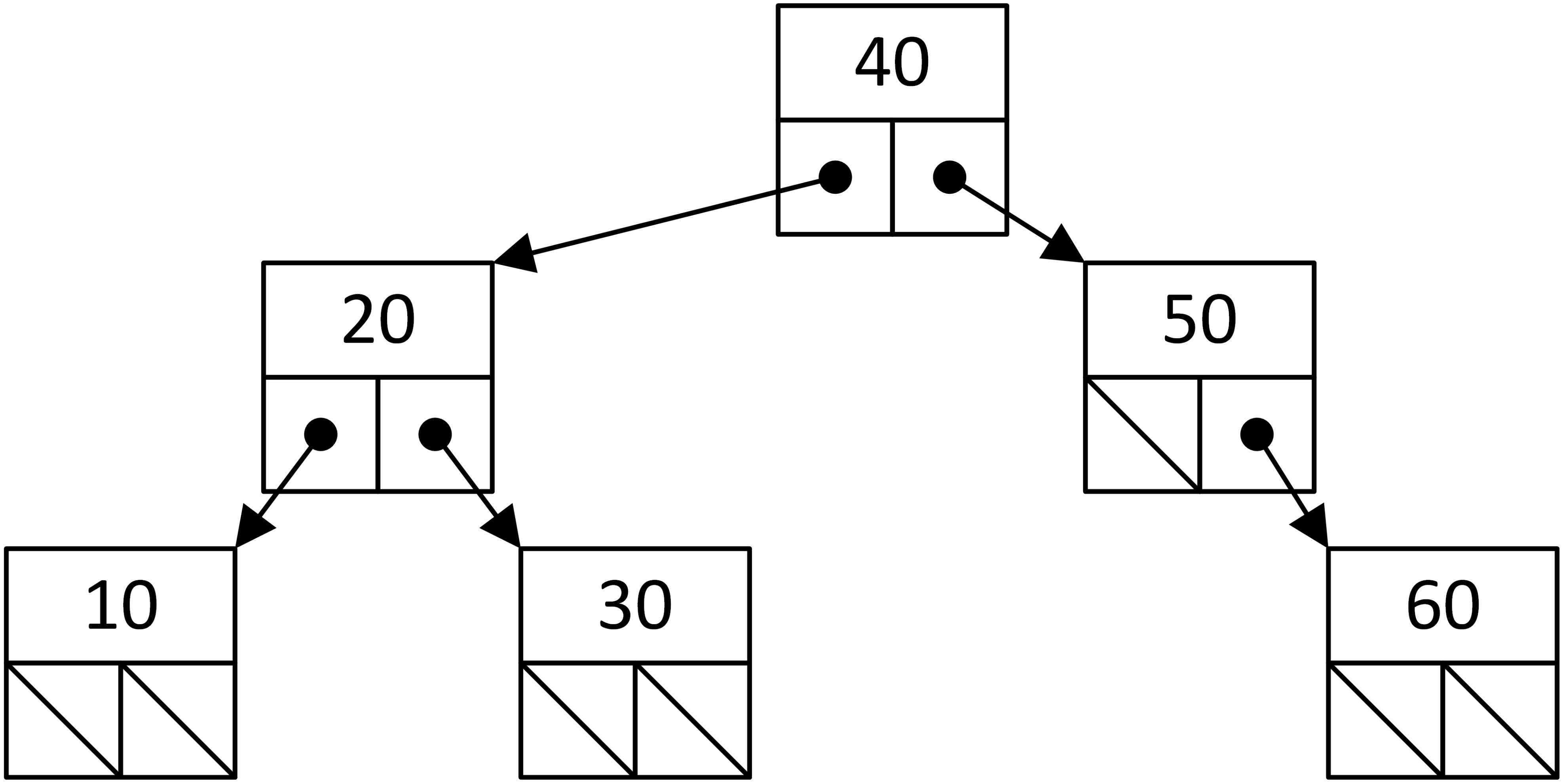
In previous courses, we used key/value pairs with *Binary Search Trees (BSTs)*

In this course, we only store **single items** (keys) in BSTs. **Values are considered an *augmentation*.**

Similar to linked lists, a BST is pointer to *a BST node* (`bstnode`), and an empty tree is `NULL`.

```
struct bstnode {
  int item;
  struct bstnode *left;
  struct bstnode *right;
};
```

The definition of a BST includes the **ordering property**.



Additional tree definitions

The ***depth*** of a node is the number of nodes from the root to the node, including the root. The *depth* of the root node is 1.

The ***height*** of a tree is the maximum *depth* in the tree. The *height* of an empty tree is 0.

For the tree on the previous slide, the *depth* of the node with 50 is 2, and *height* of the tree is 3.

Other courses may use slightly different definitions of *depth* and *height*.

Making new nodes

In Racket, dynamic memory is used to *“make”* a structure.

For example, `(posn 3 4)` (previously `(make-posn 3 4)`) obtains space from the Racket heap for the structure.

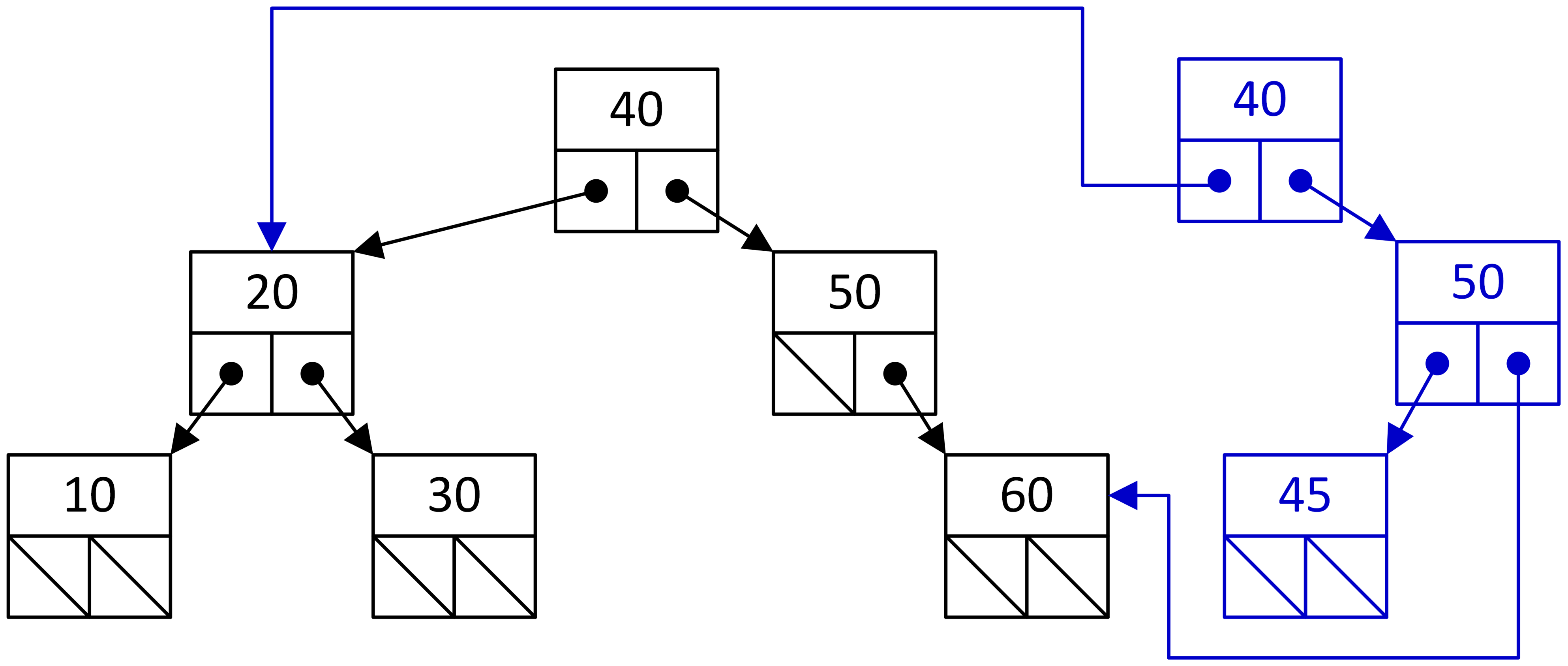
In a **functional** approach, we can similarly *“make”* a bstnode.

```
struct bstnode *make_bstnode(int i, struct bstnode *l,
                                    struct bstnode *r) {

  struct bstnode *new = malloc(sizeof(struct bstnode));
  new->item = i;
  new->left = l;
  new->right = r;
  return new;
}
```

Using a functional approach with BSTs may cause memory management problems similar to the problem shown with the linked list `insert` function.

```
// functional approach
struct bstnode *bst_insert(int i, struct bstnode *t) {
  if (t == NULL) {
    return make_bstnode(i, NULL, NULL);
  } else if (i == t->item) {
    return t;
  } else if (i < t->item) {
    return make_bstnode(t->item,
                        bst_insert(i, t->left), t->right);
  } else {
    return make_bstnode(t->item, t->left,
                        bst_insert(i, t->right);
  }
}
```

```
struct bstnode my_tree = make_bstnode(40,
  make_bstnode(20, make_bstnode(10,NULL,NULL),
                   make_bstnode(30,NULL,NULL)),
  make_bstnode(50,NULL, make_bstnode(60,NULL,NULL)));

struct bstnode new_tree = bst_insert(45, my_tree);
```



An imperative approach can be used to *change* an *existing* tree.

Like with linked lists, we must pass a *pointer to a pointer* to the root BST node.

```
// imperative approach (recursive)
void bst_insert(int i, struct bstnode **ptr_root) {
  struct bstnode *t = *ptr_root;
  if (t == NULL) { // tree is empty
    *ptr_root = make_bstnode(i, NULL, NULL);
  } else if (t->item == i) {
    return;
  } else if (i < t->item) {
    bst_insert(i, &(t->left));
  } else {
    bst_insert(i, &(t->right));
  }
}
```

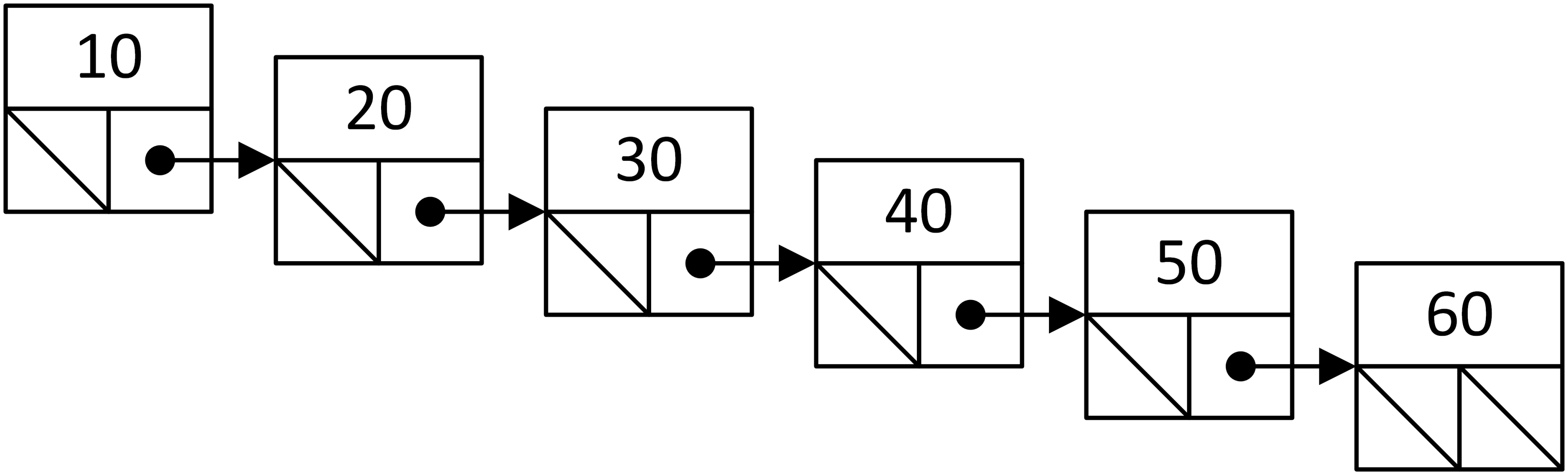
```
// imperative approach (iterative)
void bst_insert(int i, struct bstnode **ptr_root) {
  struct bstnode *prev = NULL;
  struct bstnode *cur = *ptr_root;
  while (cur != NULL) {
    if (i == cur->item) return;
    prev = cur;
    if (i < cur->item) cur = cur->left;
    else cur = cur->right;
  }
  struct bstnode *new = make_bstnode(i, NULL, NULL);
  if (prev == NULL) { // empty tree
    *ptr_root = new;
  } else if (i < prev->item) {
    prev->left = new;
  } else {
    prev->right = new;
  }
}
```

Most tree functions are more easily written and understood with a recursive approach.

Trees and efficiency

What is the efficiency of `bst_insert`?

The *worst case* is when the tree is ***unbalanced***, and *every* node in the tree must be visited.



In this example, the running time of `bst_insert` is $O(n)$, where n is the number of nodes in the tree.

The running time of `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 ***balanced tree*** is a tree where the height (h) is $O(\log n)$.

Conversely, an **un**balanced tree is a tree with a height that is **not** $O(\log n)$. The height of an unbalanced tree is $O(n)$.

Using the `bst_insert` function we provided, inserting the nodes in *sorted order* creates an *unbalanced* tree.

With a **balanced** tree, the running time of the `insert`, `remove` and `search` functions 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, the `insert` and `remove` 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 an augmentation strategy.

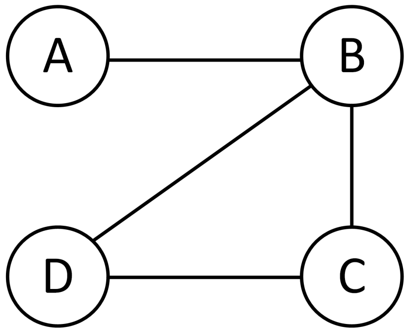
A popular tree **augmentation** is to store in *each node* the **size** of its subtree.

```
struct bstnode {
  int item;
  struct bstnode *left;
  struct bstnode *right;
  int size;                    // *****NEW
};
```

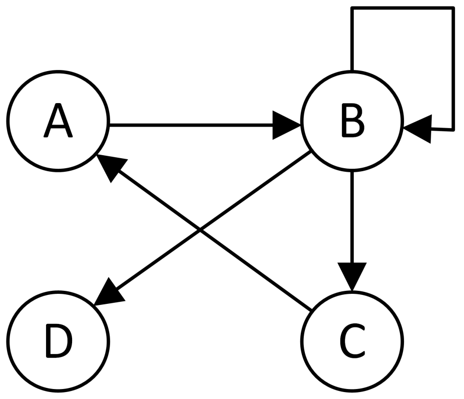
This augmentation allows us to retrieve the size of the tree in $O(1)$ time. It also allows us to implement a `select` function in $O(h)$ time. `select(k)` finds the `k`-th smallest item in the tree.

Graphs

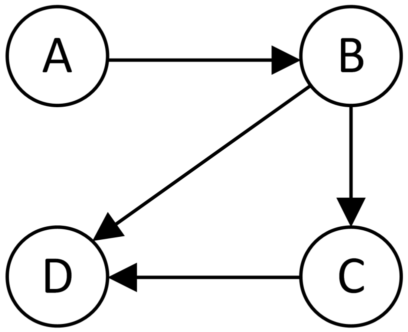
Linked lists and trees can be thought of as *"special cases"* of a ***graph*** data structure. Graphs are the only core data structure we are **not** working with in this course.



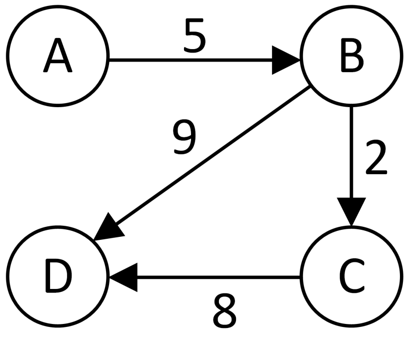
i



ii



iii



iv

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 functional or imperative approach and with a wrapper or augmented strategy
- explain the memory management issues related to working with linked lists and trees

- explain why an unbalanced tree can affect the efficiency of tree functions
- explain how a wrapper or augmentation strategy can be used to improve efficiency

Abstract Data Types (ADTs)

Readings: CP:AMA 19.3, 19.4, 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

Function	Dynamic Array	Linked List
`item_at`	$O(1)$	$O(n)$
`search`	$O(n)$	$O(n)$
`insert_at`	$O(n)$	$O(n)$
`insert_front`	$O(n)$	$O(1)$
`insert_back`	$O(1)^*$	$O(n)^\dagger$
`remove_at`	$O(n)$	$O(n)$
`remove_front`	$O(n)$	$O(1)$
`remove_back`	$O(1)$	$O(n)^\diamond$

$*$ amortized

$\dagger$ $O(1)$ with a wrapper strategy and a `back` pointer

$\diamond$ $O(1)$ with a *doubly* linked list.

Data structure comparison: unsequenced (sorted) data

Function	Sorted Dynamic Array	Sorted Linked List	Regular BST	Self-Balancing BST
`select`	$O(1)$	$O(n)$	$O(n)^*$	$O(n)^*$
`search`	$O(\log n)$	$O(n)$	$O(h)$	$O(\log n)$
`insert`	$O(n)$	$O(n)$	$O(h)$	$O(\log n)$
`remove`	$O(n)$	$O(n)$	$O(h)$	$O(\log n)$

* $O(h)$ 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.

Abstract Data Types (ADTs)

Formally, an ***Abstract Data Type (ADT)*** is a mathematical model for storing and accessing data through ***operations***. As mathematical models they transcend any specific computer language or implementation.

As discussed in Section 02, ADTs are **implemented** as data storage modules that only allows access to the data through the interface functions (ADT *"operations"*).

The underlying data structure and implementation of an ADT is hidden from the client.

example: account ADT

To re-create the “account” ADT example from Section 02 in C we define an account structure.

```
struct account {
  char *username;
  char *password;
};
```

`create_account` returns a *pointer* to a **new** account. In addition, it makes **duplicates** of the username and password strings provided by the client.

```
struct account *create_account(const char *username,
                               const char *password) {

  struct account *a = malloc(sizeof(struct account));

  a->username = malloc((strlen(username)+1)*sizeof(char));
  strcpy(a->username, username);

  a->password = malloc((strlen(password)+1)*sizeof(char));
  strcpy(a->password, password);

  return a;
}
```

In C, our ADT also requires a `destroy_account` to `free` the memory created.

```
void destroy_account(struct account *a) {
  free(a->username);
  free(a->password);
  free(a);
}
```

The remaining operation `is_correct_password` verifies passwords.

```
bool is_correct_password(struct account *a,
                         const char *password) {
  return (strcmp(a->password,password) == 0);
}
```

The interface code for our ADT would be:

```
struct account {
  char *username;
  char *password;
};

struct account *create_account(const char *username,
                               const char *password);

void destroy_account(struct account *a);

bool is_correct_password(struct account *a,
                         const char *password);
```

In this interface, the structure is “transparent” and visible to the client. The client can access the `username` and `password` fields directly.

This is not good information hiding.

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.

```
struct account;                // INCOMPLETE DECLARATION

struct account my_account;     // INVALID

struct account *a;             // VALID
```

By providing only an incomplete declaration in the interface we achieve information hiding and gain security and flexibility.

To further improve our interface, we can create a new `Account` type that is a pointer to a `struct account`.

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).

```
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).

Account: new interface

```
struct account;

typedef struct account *Account;

// operations:

Account create_account(const char *username,
                       const char *password);

void destroy_account(Account a);

bool is_correct_password(Account a, const char *password);
```

Some programmers consider it poor style to use `typedef` to “abstract” that a type is 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.,* `AccountPtr`).

The Linux kernel programming style guide recommends avoiding `typedefs` altogether.

Collection ADTs

We presented the `Account` ADT to illustrate how an ADT can be implemented in C, but as we discussed in Section 02, `Account` is not a “typical” ADT.

In this section we present five ***Collection ADTs*** that are designed to store an arbitrary number of items: stack, queue, sequence, dictionary and set.

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 most recently pushed item is accessible.

The stack ADT is not the same as the (call) stack section of memory, but conceptually they are very similar. Each item in the call stack is a frame. Each function call *pushes* a new frame on to the call stack, and each `return` *pops* the frame off of the call stack. The last function called is the first returned.

Stacks are often used in browser histories (“back”) and text editor histories (“undo”). In some circumstances, a stack can be used to “simulate” recursion.

Typical stack ADT operations:

- **push(s,i)** (or *add_front*) adds an item to the stack
- **pop(s)** (or *remove_front*) removes the “top” item on the stack
- **top(s)** (or *peek*) returns the next item to be popped
- **is_empty(s)** checks if the stack is empty

stack: interface

```
// A real ADT would be MUCH better documented
struct stack;
typedef struct stack *Stack;

// create_Stack()    returns a new Stack
// destroy_Stack(s)  frees all of the memory
Stack create_Stack(void);
void destroy_Stack(Stack s);

// push(s, i) puts i on top of the stack
void push(Stack s, int i);

// pop(s) removes the top and returns it
int pop(Stack s);

// top(s) returns the top but does not pop it
int top(Stack s);

bool is_empty(Stack s);
```

stack: implementation

```
#include "stack.h"

struct stack {
  struct llnode *topnode;     // our stack is a linked list
};

Stack create_Stack(void) {
  Stack new = malloc(sizeof(struct stack));
  new->topnode = NULL;
  return new;
}

bool is_empty(Stack s) {
  return s->topnode == NULL;
}

int top(Stack s) {
  assert(!is_empty(s));
  return s->topnode->item;
}
```

```
void push(Stack s, int i) {
  struct llnode *new = malloc(sizeof(struct llnode));
  new->item = i;
  new->next = s->topnode;
  s->topnode = new;
}

int pop(Stack s) {
  assert(!is_empty(s));
  int ret = s->topnode->item;
  struct llnode *backup = s->topnode;
  s->topnode = s->topnode->next;
  free(backup);
  return ret;
}

void destroy_Stack(Stack s) {
  while (!is_empty(s)) {
    pop(s);
  }
  free(s);
}
```

In this imperative example, the `pop` operation returned the value that was being “popped”.

In a functional approach, any operation to “change” the stack (*i.e.*, `push` and `pop`) would return the **new** stack, and so `pop` would return the new stack instead of the value being ‘popped”.

In a functional approach, the `top` operation is more important (and essential before a `pop`).

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(q,i)** (or *push_back, enqueue*)
- **remove_front(q)** (or *pop_front, dequeue*)
- **front(q)** (or *peek, next*) returns the item at the front
- **is_empty(q)**

Sequence ADT

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

Typical sequence ADT operations:

- **item_at(s,k)** returns the item at position k (k $<$ length)
- **insert_at(s,k,i)** inserts i at position k and increments the position of all items after k
- **remove_at(s,k)**
- **length(s)** (or *size*)

Dictionary ADT

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. Dictionaries are unsequenced, and are often used when fast *lookups* are required.

Typical dictionary ADT operations:

- **lookup(d,k)** returns the value for a given key k, or “not found”
- **insert(d,k,v)** (or *add*) inserts a new key value pair (or replaces)
- **remove(d,k)** (or *delete*) removes a key and its value

Set ADT

The set ADT is similar to a mathematical set (or a dictionary with no values) where every item (element) is unique. Sets are unsequenced and usually *sorted* (the unsorted ADT is less common).

Typical set ADT operations include: **member(s,i)** (*is_element_of*), **add(s,i)**, **union(s1,s2)**, **intersection(s1,s2)**, **difference(s1,s2)**, **is_subset(s1,s2)**, **size(s)**.

Because many set ADT operations produce new sets and *copies* of items, they are more common in languages with garbage collection.

Implementing collection ADTs

A significant benefit of a collection ADT is that a client can use it “abstractly” without worrying about how it is are implemented.

ADT modules are usually well-written, optimized and have a well documented interface.

However, 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 **Set**: self-balanced BSTs or hash tables*.

* A hash table is typically a an array of linked lists (more on hash tables in CS 240).

Beyond integers

The stack implementation we presented only supported integers.

What if we want to have a stack of a different type?

There are three common strategies to solve this 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).

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 push(Stack S, ItemType i) {...}
ItemType 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 `destroy_Stack` 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**.

```
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

```
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).

```
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

// push(s, i) puts item i on top of the stack
//   NOTE: The caller should not free the item until it is popped
void push(Stack s, void *i);

// top(s) returns the top but does not pop it
//   NOTE: The caller should not free the item until it is popped
void *top(Stack s);

// pop(s) removes the top item and returns it
//   NOTE: The caller is responsible for freeing the item
void *pop(Stack s);

// destroy_Stack(s) destroys the stack
// requires: The stack must be empty (all items popped)
void destroy_Stack(Stack s);
```

example: client interface

```
#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.

```
// 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.

```
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 create_Dictionary(DictKeyCompare f);

// lookup key k in Dictionary d
void *lookup(Dictionary d, void *k);
```

```
// dictionary.c (partial implementation)

struct bstnode {
  void *key;
  void *value;
  struct bstnode *left;
  struct bstnode *right;
};

struct dictionary {
  struct bstnode *root;
  DictKeyCompare key_compare;   // function pointer
};


Dictionary create_Dictionary(DictKeyCompare f) {
  struct dictionary *newdict = malloc(sizeof(struct dictionary));
  newdict->key_compare = f;
  //...
}
```

This implementation of `lookup` illustrates how the comparison function would work.

```
void *lookup(Dictionary d, void *k) {
  struct bstnode *t = d->root;
  while (t) {
    int result = d->key_compare(k, t->key);
    if (result < 0) {
      t = t->left;
    } else if (result > 0) {
      t = t->right;
    } else { // key found!
      return t->value;
    }
  }
  return NULL; // (no key found)
}
```

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.

```
void qsort(void *arr, int len, size_t size,
           int (*compare)(void *, void *));
```

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 (`bsearch`) function that searches any sorted array for a key, and either return a pointer to the element if found, or `NULL` if not found.

```
void *bsearch(void *key,
              void *arr,
              int len,
              size_t size,
              int (* compare)(void *, void *));
```

Goals of this Section

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

- describe the collection ADTs introduced (stack, queue, sequence, dictionary, set) and their operations
- implement any of the collection ADTs or be able to use them as a client
- determine an appropriate data structure or ADT for a given design problem
- deduce the running time of an ADT operation for a particular data structure implementation

- use opaque structures (incomplete declarations) and `typedef`
- 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:

```
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

```
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

```
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”.

Preprocessing

In the preprocessing step the preprocessing *directives* are carried out (Section 03).

For example, the `#include` directive “cut and pastes” the contents of one file into another file.

The C preprocessor is not *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 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
```

gcc

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 `main` function does not have any parameters, but that is not exactly true. They are optional.

```
int main(int argc, char *argv[]) {
  //...
}
```

`argv` is an array of strings, and `argc` is the length of the array.

The length of the array is always at least one, because `argv[0]` contains the name of the executable program itself. The number of parameters is `(argv - 1)`.

```c
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`.

```c
// 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 `#include`s to determine the required modules, then compile and link all of the modules together
- if no `.in` files exist, execute the program, otherwise, for each `.in` file execute the program redirecting from `.in` files and to `.out` files

```
$ ./program < mytest.in > mytest.out
```

- if `.expect` files exist, compare the `.out` files to the `.expect` files and store the differences in `.check` files

Full C language

We have skipped many C language features, including:

- `union`s and `enum`erations
- `int`eger 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

This Appendix contains some of the content presented in tutorials, including additional examples and language syntax details.

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).

$$f(x, y) = (x + y)^2$$

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

CS 135 terminology

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

(f 3 4)     ; => 49
```

For the above example, we first *define* a function `f`, which has two ***parameters*** (`x` and `y`).

We then ***apply*** the function `f`, which ***consumes*** two ***arguments*** (the ***values*** `3` and `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.

```
(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
```

Conditionals

```
(cond
  [q1 a1]
  [q2 a2]
  [else a3])
```

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

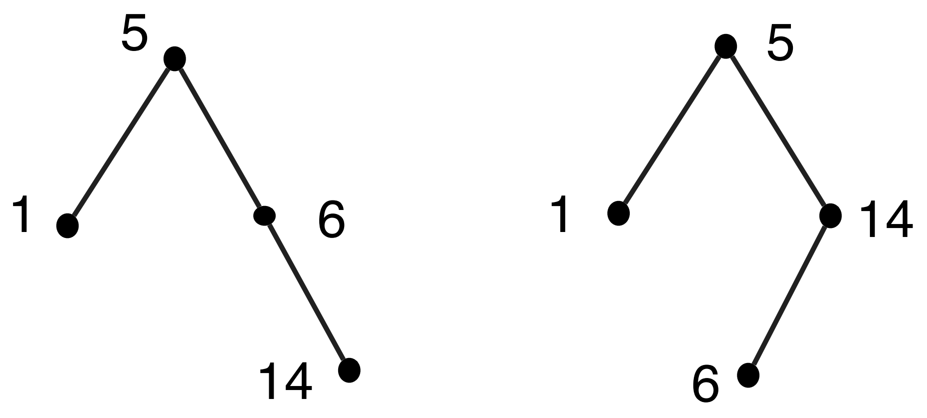
In CS 135 we saw the Binary Search Tree (BST), where each node stores a `key` and a `value`

```
;; 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:

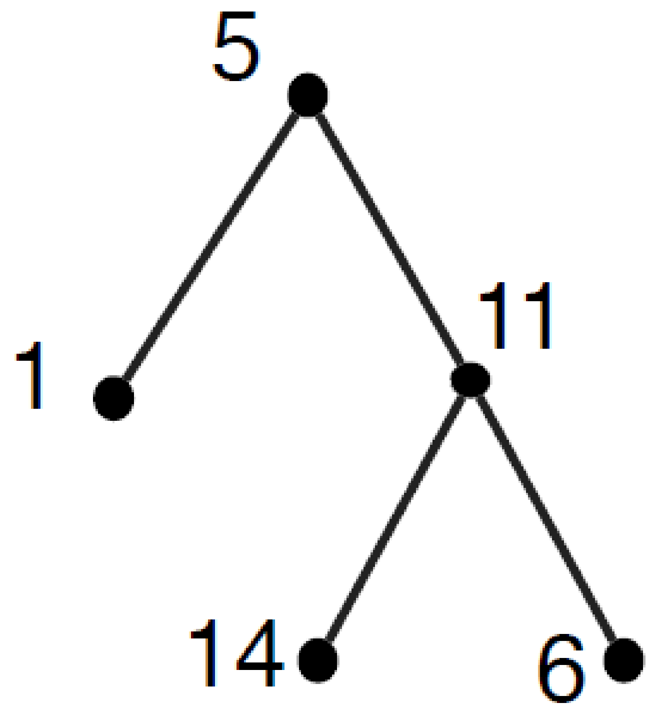
(we often only show the keys in a BST diagram)



```
(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 `filter`, `map`, `foldr`, `foldl`, and `build-list`.

```
(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
```

Lambda

In Racket, `lambda` can be used to generate an anonymous function when needed.

```
(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)
```

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.

```
(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

```
(cond
  [q1 a1]
  [q2 a2]
  [else a3])
```

In full Racket, the questions do not have to be Boolean values because any value that is not `#f` is considered true.

In full Racket, `cond` does not produce an error if all questions are false: it produces `#<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`)

```
(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).

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

Lists

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.

```
(define (t-area a b c)
  (local
    [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

Is equivalent to:

```
(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`.

```
(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)
```

A.3 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.,* $0 \ldots 9999$).

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 $(0 \ldots 1111)$ or $(0 \ldots 15)$ in base 10.

In CS 251 / CS 230 you will learn more about binary notation.

Hexadecimal notation

In ***hexadecimal (hex)*** representation (also known as ***base 16***) there are **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 *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

Dec	Bin	Hex	Dec	Bin	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

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:

```
0x1234 = 0001 0010 0011 0100
0x2A9F = 0010 1010 1001 1111
0xFACE = 1111 1010 1100 1110
```

In C, a number with the prefix `0x` is interpreted as a hex number:

```
int num = 0x2A9F; //same as num = 10911
```

A number prefixed with a zero `0` is interpreted as an *octal* numbers (base 8).

```
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:	Secondary Storage:
• very fast (nanoseconds)	• (≈20x-1000x) slower
• medium capacity (≈GB)	• large capacity (≈TB)
• high cost ($) per byte	• low cost ($) per byte
• harder to remove	• removable or portable
• erased on power down	• 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 **RA**M 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”*.