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 (Winter 2019)

Instructors: Alice Gao, Tim Brecht, Nomair Naeem, Adrian Reetz, Joe Istead, Lesley Istead

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: Wednesdays

Be sure to explore the course website: *Lots* of useful info!

About me (your instructor)

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

Curriculum

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.

CS 135	$\Rightarrow$	CS 136	$\Rightarrow$	CS 246
functional		*imperative*		*object-oriented*

Each course incorporates a wide variety of CS topics and is **much more** than the paradigm taught.

Programming languages

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

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

We present C language features and syntax only as needed.

We occasionally use Racket to illustrate concepts and highlight the similarities (and differences) between the two languages.

What you learn in this course can be transferred to most languages.

Programming environment (Seashell)

We use our own customized “`Seashell`” development environment.

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

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

Course materials

Textbooks:

- “C Programming: A Modern Approach” (CP:AMA) by K. N. King.
  (strongly recommended)
- “How to Design Programs” (HtDP) by Felleisen, Flatt, Findler, Krishnamurthi
  (very optional)
  Available for free online: `http://www.htdp.org`

Course notes:

Available on the web page and as a printed coursepack from media.doc (MC 2018).

Several different styles of “boxes” are used in the notes:

Important information appears in a thick box.

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

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

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

Appendices

The course notes are supplemented by several online ***appendices***, which are not included in the printed notes. The appendices include:

- additional content and examples
- C syntax details
- Racket language details

Some content from the appendices may appear on exams (this will be made very clear before exams).

Marking scheme

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

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

Class participation

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

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

Participation grading

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

To achieve a perfect participation mark

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

Assignments

Assignments are *weekly* (approximately 10 per term).

Each assignment is weighted equally (except A0).

- **read the assignment instructions carefully**
- read the official piazza post frequently
- rules & requirements may change throughout the course

A0 does not count toward your grade, **but must be completed** before you can receive any other assignment marks.

Assignment *questions* are colour-coded as either “black” or “gold” to indicate if any collaboration is permitted.

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

- You can discuss assignment *strategies* openly (including online)
- You can search the Internet for strategies or code examples

- You can discuss your code with *individuals*, but **not** online or electronically
  (piazza, facebook, github, email, IM, *etc.*)

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

If you submit any work that is not your own, you must still cite the origin of the work in your source code.

For **GOLD** questions, **no collaboration** is permitted:

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

You may always discuss your code **with course staff**.

Academic integrity is strictly enforced for gold questions.

Assignments: second chances

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

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

Marmoset

Assignments are submitted to the Marmoset submission system:

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

There are two types of Marmoset *tests*:

- **Public** (*basic / simple*) tests results are available immediately and ensure your program is “runnable”.
- **Private** (*comprehensive / correctness*) tests are available after the deadline and fully assess your code.

Public tests do **not** thoroughly test your code.

- Marmoset uses the best result from all of your submissions (there is never any harm in resubmitting).
- For questions that are *hand-marked*, the most recent submission (before the deadline) with the highest score is marked.
- You can *submit* your assignments via `Seashell` and view *public* test results.
- Every submission is stored (backed up) for your convenience.

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

Testing strategies

You are expected to test your own code.

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

We will discuss multiple testing strategies throughout this course.

Design recipe

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

The design recipe has two main goals:

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

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

Documentation

In this course, every function you write must have:

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

Unless otherwise stated, you are **not** required to provide:

templates, data definitions or examples.

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

Hand-marking

Questions that are hand-marked for *“style”* may be evaluated for:

- documentation and comments
- code readability
- whitespace and indentation
- identifiers (variable & function names)
- appropriate use of helper functions
- testing methodology

The purpose of hand-marking is not to “punish” or “torture” you.

It is **formative feedback** to improve your learning.

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

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

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

View your formative feedback on `MarkUs`.

Getting help

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

Course announcements made on piazza are **mandatory reading** (including official assignment and exam posts).

Piazza etiquette

- **read** the *official assignment post* before asking a question
- **search** to see if your question has already been asked
- **use** meaningful titles
- **ask** *clarification questions* for assignments
  (do not ask *leading questions* for **GOLD** questions)
- **do not** discuss strategies for **GOLD** questions
- **do not** post any of your assignment code *publicly*
- you can post your **commented** code *privately*, and an ISA or Instructor *may* provide some assistance.

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

- 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

An Introduction to C

Readings: CP:AMA 2.2, 2.3, 2.7, 4.1, 5.1, 9.1

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

The primary goal of this section is to be able to write and test simple functions in C.

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 04 and Section 05).

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

First, we learn to write simple functions in C.

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

In Section 03 we introduce new programming concepts (imperative programming).

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

```
// C comment                    ; Racket comment

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

Expressions

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

```
// C Expressions:                   ; Racket Expressions:
3 + 3                               (+ 3 3)
1 + 3 * 2                           (+ 1 (* 3 2))
(1 + 3) * 2                         (* (+ 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` is equivalent to `(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.

```
(4 * 5) / 2    ⇒   10

4 * (5 / 2)    ⇒    8

  -5 / 2       ⇒   -2
```

Remember, use parentheses to clarify the order of operations.

[†] C99 standardized the “(round toward zero)” behaviour.

The % operator

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

```
9 % 2    ⇒   1
9 % 3    ⇒   0
9 % 5    ⇒   4
```

The value of `(a % b)` is equal to: `a - (a / b) * b`.

In this course, avoid using `%` with negative integers.

`(i % j)` has the same sign as `i` (see CP:AMA 4.1).

Function definitions

```
// C function:

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

```
; Racket function:

; my-sqr: Int -> Int
(define (my-sqr n)
  (* n n))
```

There are many subtle and important differences between defining a function in Racket and in C.

We explore those differences over the next few slides.

Identifiers

C identifiers (“names”) must start with a letter, and can only contain letters, underscores and numbers.

For example, use `my_sqr` instead of `my-sqr`.

`underscore_style` is the most popular style in C, and the style we use in this course.

In other languages, `camelCaseStyle` is popular alternative.

C identifiers can start with a leading underscore (`_name`) but they may interfere with reserved keywords. Avoid them in this course as they may interfere with marmoset tests.

Function block and return

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

```
; my-sqr: Int -> Int
(define (my-sqr n)
  (* n n))
```

Braces (`{}`) indicate the beginning and end of the function **body**, known in C as the function ***block***.

The expression is preceded by the `return` keyword and followed by a semicolon (`;`).

Note the placement of the braces (`{}`) and the use of whitespace and indentation (more on this later).

Dynamic typing in Racket

Racket uses ***dynamic typing***: types are determined **while** the program is running.

```
;; dtype: (anyof Int Str) -> (anyof Bool Sym)

(define (dtype param)
  (cond [(integer? param) true]
        [(string? param)  'dynamic]))
```

The types of the `param`eter and the produced (*"returned"*) value are both *dynamic*.

We communicate the types in the *contract* as a **comment**.

Static typing in C

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

```
; my-sqr: Int -> Int
(define (my-sqr n)
  (* n n))
```

C uses ***static typing***: all types **must** be known **before** the program is run and they cannot change.

The `return` type of `my_sqr` is an `int` (for `int`eger).

Parameter `n` is also an `int`.

C Types

For now, we only use **integers** in C. More types are introduced soon.

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

In Racket, a contract violation may cause a “type” runtime error.

```
  (my-sqr "hello")    ; Racket runtime error
```

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

```
  my_sqr("hello")    // does not run in C
```

If you omit the type in a function definition:

```
int my_sqr(int n) {   // properly typed
  return n * n;
}

my_bad_sqr(n) {       // missing types
  return n * n;
}
```

C assumes the type is an `int` and may display a warning such as:

```
type specifier missing, defaults to 'int'
```

This is **very bad style**: you should specify *every* type.

When working with integer values in C, do not add a leading (preceding) zero (`0`) to the value. For example, do not write `017` if you want to represent the number `17`.

A leading zero may seem harmless, but it is not:

```
  trace_int(17);                 17 => 17
  trace_int(017);                017 => 15
  trace_int(my_sqr(010));        my_sqr(010) => 64
```

In C, integer values that start with a zero are evaluated in octal (base 8), so `010` is equivalent to 8.

Integer values that start with `0x` are evaluated in hexadecimal, so `0x10` is equivalent to `16`.

Function terminology

In this course, we use more common terminology:

We ***call*** a function by ***passing*** it arguments and it ***returns*** a value.

apply	$\Rightarrow$	call
consume	$\Rightarrow$	pass
produce	$\Rightarrow$	return

In “functional” CS 135 terminology, we ***apply*** a function, which ***consumes*** arguments and it ***produces*** a value.

More function syntax

Multiple arguments and parameters are separated by commas (`,`).

```
int my_add(int x, int y) {
  return x + y;
}

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

Use `void` to define a function with no parameters.

Use `()` to call a parameterless function: *e.g.*, `my_num()`.

Also, in C you cannot have *local* (or “nested”) functions defined inside of other functions.

If you omit the `void` in a parameterless function definition:

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

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

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

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

Function documentation

Racket and C purpose statements are nearly identical:

```
// my_sqr(n) squares n

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

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

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

In C, the contract *types* are part of the function definition so no type contract documentation is necessary. However, you should still add a **requires** comment if appropriate.

```
  // my_divide(x, y) ....
  // requires: y is not 0

  int my_divide(int x, int y) {
    return x / y;
  }
```

Whitespace

C mostly ignores whitespace.

```
// The following three functions are equivalent

int my_add(int x, int y) {                   // GOOD
  return x + y;
}

int my_add(int x,int y){return x+y;}         // BAD

int my_add(int x, int                        // RIDICULOUSLY
y){return x+                                 // BAD
y ; }
```

You should follow the course style. The course staff and markers may not understand your code if it is poorly formatted.

CS 136 Style

```
int my_add(int x, int y) {
  return x + y;
}
```

- a block start (open brace `{`) appears at the end of a line
- a block end (close brace `}`) is aligned with the line that started it, and appears on a line by itself
- indent a block **2** (recommended), 3 or 4 *spaces*: **be consistent**
- add a space after commas and around arithmetic operators

Typing `Ctrl-I` in `Seashell` will auto-indent your code for you.

When you have a function with a large number of parameters, or a really large expression, place code on the following line.

```
int my_super_long_function(int a, int b, int c, int d,
                           int e, int f, int g) {
  return a * b + b * c + c * d + d * e + e * f +
         f * g + g * a;
}
```

The “best” way to style your code (*e.g.*, block formatting) is a matter of taste and is often a topic of debate.

The style we have chosen is the most widely accepted style for C (and C++) projects (*e.g.*, it conforms to the Google style guide).

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 demonstrate how to run and test a simple C program.

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.

```
int main(void) {
  //...
  return 0;         // success!
}
```

The `return` value communicates to the OS the “error code”

(also known as the “exit code”, “error number” or just `errno`).

A successful program `return`s zero (no error code).

[†] `main` has *optional* parameters (discussed in Section 13).

`main` is a special function and does not require an explicit `return` value. The default value is success (zero).

```
// main() does not need a purpose statement

int main(void) {
  //...
  return 0;          // this is optional
}
```

In this course, `main` does **not** require a purpose statement, but in general it is good style to document the purpose of a program.

In this course, your `main` function should never `return` a non-zero value, as it causes your marmoset tests to fail.

Top-level expressions

In `DrRacket`, the final values of ***top-level expressions*** (code outside of a function) are displayed in the “interactions window”.

```
;; my racket program

(+ 1 1)                 ;; <-- top level

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

(my-sqr 7)              ;; <-- top level
```

```
2
49
```

In C, *top-level expressions* are **not** allowed.

```
// my C program

1 + 1;                  // INVALID

int my_sqr(int n) {
  return n * n;
}

my_sqr(7);               // INVALID
```

Tracing expressions

We have provided ***tracing tools*** to help you “see” what your code is doing. Here, we use `trace_int` inside of `main` to *trace* several expressions and display them to the screen (console):

```
int main(void) {
  trace_int(1 + 1);
  trace_int(my_sqr(7));
}
```

```
1 + 1 => 2
my_sqr(7) => 49
```

You can leave the *tracing* in your code. It is ignored in our tests and does not affect your results (no need to comment it out).

We’re now ready to run our first program.

```
// My first C program

#include "cs136.h"          // <-- more on this later

int my_sqr(int n) {
  return n * n;
}

int main(void) {
  trace_int(1 + 1);
  trace_int(my_sqr(7));
}
```

Note the necessary `#include` line at the top of the program.

For now, always add this line (it is explained later).

Function ordering

If you re-order the two functions in our program:

```
int main(void) {
  trace_int(1 + 1);
  trace_int(my_sqr(7));
}

int my_sqr(int n) {        // now below main
  return n * n;
}
```

You get an error such as:

```
implicit declaration of function 'my_sqr' is invalid
```

For now, always place function definitions **above** any other functions that reference them (so `main` is at the bottom).

Boolean expressions

To facilitate testing, we need Boolean expressions.

In C, a Boolean expression does not produce “true” or “false”.

An expression produces zero (`0`) for “false”, and one (`1`) for “true”.

In our environment, the constants `true` and `false` have been defined to be `1` and `0` (for convenience).

If used in a Boolean expression, **any non-zero value** is also considered “**true**”.

Only zero is “false”.

Equality operator

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

```
(3 == 3)   ⇒   1 (true)
(2 == 3)   ⇒   0 (false)
```

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

```
(2 != 3)   ⇒   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 03).

It is such a serious concern that it warrants an extra slide as a reminder.

The ***not***, ***and*** and ***or*** operators are respectively `!`, && and `||`.

```
!(3 == 3)                ⇒  0
(3 == 3) && (2 == 3)     ⇒  0
(3 == 3) && !(2 == 3)    ⇒  1
(3 == 3) || (2 == 3)     ⇒  1
2 && 3                   ⇒  1  (why?)
```

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

```
(a != 0) && (b / a == 2)
```

does not generate 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.

```
(2 < 3)    ⇒  1
(2 >= 3)   ⇒  0
```

`!(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.

You are not expected to “memorize” the order of operations.

When in doubt (or to add clarity) **add parentheses**.

negation	!
multiplicative	* / %
additive	+ -
comparison	< <= >= >
equality	== !=
and	&&
or	||

Assertions

The `assert` function can be used in place of Racket’s `check-expect`. The equivalent of:

```
(check-expect (my-sqr 7) 49)
```

is:

```
assert(my_sqr(7) == 49);
```

`assert(exp)` **stops** the program and displays a message if the expression `exp` is false (zero).

If `exp` is true (non-zero), it just continues to the next line of code.

```c
// My second C program (now with testing!)

#include "cs136.h"

int my_sqr(int n) {
  return n * n;
}

int main(void) {
  assert(my_sqr(0) == 0);
  assert(my_sqr(1) == 1);
  assert(my_sqr(2) == 4);
  assert(my_sqr(32) == 1024);
  assert(my_sqr(-1) == 1);
  assert(my_sqr(-32) == 1024);
}
```

We discuss additional testing methods later. For now, test your code with `assert`s in your `main` function as above.

Function requirements

The `assert` function is also very useful for **verifying function requirements**.

At the start of each function, you can add `assert`s.

```
// my_divide(x, y) ....
// requires: y is not 0

int my_divide(int x, int y) {
  assert(y != 0);                 // assert(y) also works
  return x / y;
}
```

Whenever it is feasible, `assert` any function requirements.

If you have a function with more than one requirement:

```
// my_function(x, y, z) ....
// requires: x is positive
//           y < z

int my_function(int x, int y, int z) {
  assert((x > 0) && (y < z));    // POOR

  assert(x > 0);                 // GOOD
  assert(y < z);
  //...
}
```

It is better to have several small `assert` statements. That way it is easier to determine which assertion failed (and which requirement was not met).

bool type

The `bool` *type* is an integer that can only have a value of 0 or 1.

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


bool my_negate(bool v) {
  return !v;
}
```

Conditionals

C's `if` *statement* allows us to have functions with conditional behaviour.

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

There can be more than one `return` in a function, but only one value is returned. The function "exits" when the first `return` is reached.

Emulating simple cond behaviour

Racket's `cond` special form consumes a sequence of question and answer pairs (where 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: using if, else and else if

```
// in_between(x, lo, hi) determines if lo <= x <= hi
// requires: lo <= hi

bool in_between(int x, int lo, int hi) {
  if (x < lo) {
    return false;
  } else if (x > hi) {
    return false;
  } else {
    return true;
  }
}
```

example: recursion in C

Recursion in C behaves the same as in Racket.

```
// sum_first(n) sums the natural numbers 0..n
// requires: n >= 0

int sum_first(int n) {
  if (n <= 0) {
    return 0;
  } else {
    return n + sum_first(n - 1);
  }
}
```

if vs. cond

Given the examples we have seen so far, it might appear that Racket's `cond` and C's `if` are "the same".

Fundamentally, they are quite different.

`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 04 after we understand how "statements" differ from expressions. For now, only use `if` as demonstrated.

Unlike C's `if` *statement*, the C *ternary conditional* operator (`?:`) does produce a value.

The value of the expression:

```
q ? a : b
```

is `a` if `q` is true (non-zero), and `b` otherwise.

For example:

```
(v >= 0) ? v : -v    // abs(v)
(a > b) ? a : b      // max(a, b)
```

You may use the `?:` operator in this course, but use it sparingly.

Overuse of the `?:` operator can make your code hard to follow.

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
- re-write a simple Racket function in C (and vice-versa)
- use the C operators introduced in this module

  (including `%` `==` `!=` `>=` `&&` `||`)
- explain the significance of the `main` function in C
- perform basic tracing in C using `trace_int`
- use `assert` for testing and to verify requirements
- provide the required documentation for C functions

Introduction to Imperative C

Readings: CP:AMA 2.4, 3.1, 4.2–4.5, 5.2, 10

- 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

The primary goal of this section is to be able to write programs that use I/O and mutation.

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 return a value, and the value depends **only on the argument value(s)**.

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

A programming *paradigm* can also be thought 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)` returns 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)` returns a **new** list that is sorted, but the original list `lon` does not change.

In this course, our focus is on the ***imperative programming paradigm***.

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

In imperative programming we use a **sequence of statements** (or “instructions”).

To highlight the difference, we consider an imperative example in Racket (which will seem bizarre).

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

Functions with multiple expressions

We have only seen Racket functions with a *single* expression:

```
(define (f n)
  (* n 4))

(f 10)
=> 40
```

In full Racket, functions can contain *multiple* expressions:

```
(define (mystery n)
  (/ n 2)
  (+ n 3)
  (* n 4))

(mystery 10)
=> ???
```

Racket (implicitly) always uses the `begin` special form:

```
(define (mystery n)
  (begin                        ; explicitly showing begin
    (/ n 2)
    (+ n 3)
    (* n 4)))
```

The behaviour of `begin` is as follows:

- each expression is evaluated **in order**
- the value of each expression is *discarded* (or “ignored”), except for the last expression
- `begin` produces the value of the *last* expression

In the *functional paradigm*, discarded expressions are **useless**.

Side effects

In the *imperative paradigm*, an expression can generate a ***side effect*** (in addition to producing a value).

A *side effect* changes the ***state*** of the program (or “the world”).

Functions and programs can also have side effects.

We borrow terminology from medicine, where a pill to cure an illness (*e.g.*, a headache) might also have a side effect (*e.g.*, it causes all of your hair to fall out).

example: side effects

Imagine that the following function exists:

```
(define (scary n)
  (begin
    (turn-the-lights 'off)
    (play-mp3 "scary.mp3")
    (shout "Boo!")
    (* n 4)))

(scary 10)

=> 40
```

(scary 10) still returns `40` as expected, but it also does *more*: it changes the state of the "world" by turning out the lights, playing scary music and shouting `Boo!`

```
(define (scary n)
  (begin
    (/ n 2)                          ;; not useful
    (turn-the-lights 'off)
    (play-mp3 "scary.mp3")
    (shout "Boo!")
    (* n 4)))
```

Adding an extra mathematical expression such as `(/ n 2)` is not useful because it does not have a *side effect*.

`begin` is an **imperative** special form that only makes sense if the discarded expressions have side effects.

In functional programming there are no side effects.

This is one of the significant differences between imperative and functional programming.

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

We are more relaxed: a function can have side effects.

The “imperative programming paradigm” is also closely related to the “procedural programming paradigm”.

Documentation: side effects

You should clearly communicate if a function has a side effect.

```
;; (scary n) multiplies n by four
;; scary: Int -> Int
;; effects: modifies the lights
;;          plays sound
;;          shouts

(define (scary n)
  (begin
    (turn-the-lights 'off)
    (play-mp3 "scary.mp3")
    (shout "Boo!")
    (* n 4)))
```

Add an **effects:** section to the function documentation if there are any side effects.

Types of side effects

In this course, we encounter two types of side effects:

- Input & Output (***I/O*** for short)
- Mutation

  (memory modification & variables)

Both involve **changing state** (more on this later).

In this section we first introduce *output*, then we explore *mutation* and conclude with *input*.

I/O

I/O is the term used to describe how programs *interact* with the “real world”. For example a program (“app”) on your phone may interact with you in many different ways.

Input may include: a touch screen (onscreen keyboard), your voice or the camera.

Output may include: the screen (display), sounds or vibrations.

A program may also interact with non-human entities such as: a file, a GPS, a printer or even a different computer on the internet.

Text I/O

In this course, we only use simple **text-based I/O**.

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

```
#include "cs136.h"

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

```
Hello, World
```

For now, make sure you have the above `#include`.

We omit it in the following slides to save space.

C blocks

What if we want to have more than one `printf`?

We have already seen C's equivalent of `begin`: a **block** (`{}`), also known as a ***compound statement***, is a **sequence of statements**[†].

(Unlike `begin`, a C block does not produce a value).

```
int main(void) {
  printf("Hello, World");
  printf("C is fun!");
}
```

[†] Blocks can also contain local variable *definitions*, which are not statements.

```
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!\n");
```

```
Hello, World
C is
fun!
```

The first parameter of `printf` is a `"string"`. Until we discuss strings in Section 09, this is one of the few places you are allowed to use strings.

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

```
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 argument. There can be multiple placeholders, each requiring an additional argument.

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

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

C uses different placeholders for each type. The *placeholder* we use for integers is `"%d"` (which means "**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!
```

Similarly,

- to print a backslash (\), use two (\\)
- to print a quote ("), add an extra backslash (\")

Many computer languages have a `printf` function and use the same placeholder syntax as C. The placeholders are also known as **f**ormat specifiers (the **f** in `printf`).

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.

Functions with side effects

Both functions below `return` the same value: an `int` (n^2).

```c
// quiet_sqr(n) squares n

int quiet_sqr(int n) {
  return n * n;
}


// noisy_sqr(n) squares n
// effects: produces output

int noisy_sqr(int n) {
  printf("I'm squaring %d\n", n);
  return n * n;
}
```

However, only `noisy_sqr` has a *side effect*.

I/O terminology

In the context of I/O, be very careful with your terminology.

```
int quiet_sqr(int n) {
  return n * n;
}
```

Informally, someone might say:

“if you input 7 into `quiet_sqr`, it outputs 49”.

This is **poor terminology**: `quiet_sqr` does not read input and does not print any output.

Instead, you should say:

*“if you **pass** 7 to `quiet_sqr`, it **returns** 49”.*

```
int noisy_sqr(int n) {
  printf("I'm squaring %d\n", n);
  return n * n;
}
```

For `noisy_sqr`, you should say:

*"if you **pass** 7 to `noisy_sqr`, it **outputs** a message and **returns** 49".*

It is common for beginners to confuse **output** (*e.g.*, via `printf`) and the **return value**.

Ensure you understand the correct terminology and **read your assignments very carefully.**

Testing I/O

We can use `assert`s to **test** that `quiet_sqr` and `noisy_sqr` `return` the correct values:

```
int main(void) {
  assert(quiet_sqr(3) == 9);
  assert(quiet_sqr(7) == 49);
  assert(noisy_sqr(3) == 9);
  assert(noisy_sqr(7) == 49);
}
```

But how do we test that `noisy_sqr` produced the correct **output**?

Testing functions with *side effects* is **much** more challenging.

Seashell: [RUN] vs. [I/O TEST]

In `Seashell` there are two ways of “running” your program.

The only difference is the I/O behaviour:

- With the **[RUN]** button, input is read from the **keyboard**. Any output is displayed in the console (“screen”).
- With the **[I/O TEST]** button, input is read from **input file(s)** (*e.g.,* `testfile.in`) instead of the keyboard.
  If a corresponding **output test file** exists (*e.g.,* `testfile.expect`), `Seashell` checks the output against the expected output test file to see if they match.

example: i/o test

```
int noisy_sqr(int n) {
  printf("I'm squaring %d\n", n);
  return n * n;
}

int main(void) {
  assert(noisy_sqr(3) == 9);
  assert(noisy_sqr(7) == 49);
}
```

We expect the output of this program to be:

```
I'm squaring 3
I'm squaring 7
```

So we can create an `.expect` file with that expected text. Even though this program does not read any input, we must also create a corresponding `.in` file for the I/O test to work correctly.

Tracing tools

In this course we have provided *tracing tools* to help you “see” what is happening in your program and to help **debug** your code.

- `trace_msg(msg);` displays a tracing message
- `trace_int(expression);` evaluates an (`int`) expression

The tracing tools do **not** interfere with your I/O testing.

For your assignments, you should never `printf` any unnecessary output as it may affect your correctness results.

You should always use our tracing tools to help debug your code.

example: tracing tools

```
int quiet_sqr(int n) {
  trace_msg("quiet_sqr was called");
  trace_int(n);
  return n * n;
}

int main(void) {
  trace_msg("main started");
  assert(quiet_sqr(3) == 9);
  trace_int(quiet_sqr(7));
}
```

```
"main started"
"quiet_sqr was called"
n => 3
"quiet_sqr was called"
n => 7
quiet_sqr(7) => 49
```

Our tracing tools print to a different I/O **stream**.

By default, `printf` outputs to the `stdout` (**st**an**d**ard **out**put) stream.

Our tracing tools output to the `stderr` (**st**an**d**ard **err**or) stream, which is used for errors and other diagnostic messages.

In `Marmoset`, and when `Seashell` performs an **[I/O TEST]**, only the `stdout` stream is tested.

When your **[RUN]** your code, the two streams may appear mixed together in the screen (console) output.

void functions

A function may be designed to *only* generate a side effect, and not return a value.

We have already used `void` to define functions with no parameters.

`void` is **also** used to define **a function that returns "nothing"**.

```
// say_hello() displays a friendly message
// effects: produces output

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

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

Expression statements

It might surprise you that C's `printf` is not a `void` function.

`printf` returns an `int` corresponding to the number of characters printed.

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

(note that the newline (`\n`) is a single character).

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

```
 printf("hello!\n");
```

example: expression statements

```
int main(void) {
  noisy_sqr(3);
  printf("These are ");
  printf("expression ");
  printf("statements.\n");
  10 + 3;
}
```

In the above code, there is a single **block** (*compound statement*) that contains a *sequence* of five *expression statements* with the values of 9, 10, 11, 12 and 13, respectively.

What happens to those values?

The **value** of an expression statement is **discarded**. (just like with Racket's `begin`).

The five values from the previous example are never used.

The only purpose of an expression statement is to generate **side effects**.

`Seashell` may give you a warning if you have an expression statement without an obvious side effect (`10 + 3;`) but when there is a function call (`quiet_sqr(3);`) there is no warning because it assumes it has a side effect (even if there is none).

Statements

There are only three types of C statements:

- **compound statements (blocks)**

  a sequence of statements

- **expression statements**

  for generating side effects

- ***control flow statements***

  control the order in which other statements are executed

  (*e.g.*, `return`, `if` and `else`)

We discuss control flow in more detail in Section 04.

State

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

We have seen how a side effect changes the *state* of a program (or "the world"). For example, `printf` changes the state of the output.

Another way of defining the *imperative programming paradigm* is that it **manipulates state**.

State refers to the value of some data (or "information") **at a moment in time**.

Consider a simple light switch: at any moment in time it is either in the “on” state or the “off” state.

For another example, 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 04.

Variables

We use ***variables*** to store state information (values).

To define a variable in C, we need (in order):

- the **type** (*e.g.*, `int`)
- the **identifier** (“name”)
- the **initial value**

```
int my_variable = 7;
```

The equal sign (=) and semicolon (`;`) complete the syntax.

Variable scope

Variables can have ***global scope*** or ***local scope***.

```
int my_global_variable = 7;

int main(void) {
  int my_local_variable = 11;
  //...
}
```

Global variables are defined *outside* of functions (at the "top level").

Local variables are defined *inside* of functions.

For now, always make sure you define your variables **above** any code that references them.

Local variables are also known as **block variables** because their scope is restricted to the *block* they are defined in.

Variables with the same name can *shadow* other variables from outer scopes, but this is obviously very poor style. The following code defines three **different** variables named n.

```
int n = 1;

int main(void) {
  trace_int(n);     // n => 1
  int n = 2;
  trace_int(n);     // n => 2
  {
    int n = 3;
    trace_int(n);   // n => 3
  }
}
```

Initialization

C allows you to define variables *without* **initializing** them, but it is very bad style.

```
int my_variable = 7;         // initialized
int another_variable;        // uninitialized (BAD!)
```

Always initialize your variables.

In Section 04 we discuss the behaviour of uninitialized variables.

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).

example: mutation

```
int main(void) {

  int m = 5;

  trace_int(m);

  m = 6;                // mutation!

  trace_int(m);
}
```

```
m => 5
m => 6
```

Assignment Operator

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

```
m = m + 1;
```

- The “left hand side” (LHS) of the assignment operator **must** be the name of a variable (for now).
- The RHS must be an expression with the same *type* as the LHS.
- The variable on the LHS is changed (mutated) to store the **value** of the RHS. In other words, the RHS value is **assigned** to the variable.
- This is a *side effect*: the state of the variable has changed.

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 asymmetric assignment.

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);              // never do this!
printf("y is %d\n", y = 5 + (x = 3)); // this is even worse!
z = 1 + (z = z + 1);                     // really bad style!
```

Remember, **always use a double == for equality**, not a single = (which we now know is the assignment operator).

```
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 defensive programmers get in the habit of writing `(1 == x)` instead of `(x == 1)`. This causes an error if they accidentally use a single =.

Initialization is not assignment

The `=` used in *initialization* is **not** the assignment operator.

Both initialization and assignment use the equal sign (=), but they have different semantics.

```
int n = 5;          // initialization syntax

n = 6;              // assignment operator
```

The distinction is not too important now, but the subtle difference becomes important later.

Mutating local variables

When documenting the side effects **of a function**, we are only concerned in documenting state changes that are *external* to (or visible *outside* of) the function.

```
int f(int n) {
  int k = 3;
  k = k + 1;
  return n * k;
}
```

The function `f` *mutates* the local variable `k`, but it does not modify any global variables or use any I/O.

As a result, we say that `f` itself has no side effects.

Mutating parameters

As we will explore in Section 04, C makes a **copy** of the argument values passed to a function (“pass by value” convention).

This means that parameters are nearly *indistinguishable* from local variables, and can be mutated.

```
int g(int n) {
  n = n * 4;
  return n;
}
```

As with the previous example, the function g itself does not have any side effects.

example: mutating a parameter

In the following example, the variable `x` in `main` is never changed because a **copy** of `x` is passed to `g`.

```
int g(int n) {
  n = n * 4;
  return n;
}

int main(void) {
  int x = 10;
  trace_int(x);
  trace_int(g(x));
  trace_int(x);
}
```

```
x => 10
g(x) => 40
x => 10
```

Mutating global variables

A function that mutates a global variable **does** have a side effect.

```
int count = 0;

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

int main(void) {
  trace_int(increment());
  trace_int(increment());
  trace_int(increment());
}
```

```
increment() => 1
increment() => 2
increment() => 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(int k) {
  return k + n;
}

int main(void) {
  trace_int(addn(5));
  n = 100;
  trace_int(addn(5));
}
```

```
addn(5) => 15
addn(5) => 105
```

Functions: different paradigms

Functional programming paradigm:

- functions have no side effects
- returned values *only depend on the argument value(s)*

Imperative programming paradigm:

- functions may have side effects
- in addition to the argument value(s), returned values may also depend on the *state* of the program

More assignment operators

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.

Constants

A ***constant*** is a “variable” that is **immutable** (not mutable).

In other words, the value of a constant cannot be changed.

```
const int my_constant = 42;
```

To define a C *constant*, we add the `const` keyword to the type.

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**” and “**constants**”.

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 help to optimize (speed up) your code.

We often omit `const` in the slides, even where it would be good style, to keep the slides uncluttered.

Text input

In this course we have provided some helper functions to make reading in input easier. For example:

```
// read_int() returns either the next int from input
//   or READ_INT_FAIL
// effects: reads input

// the constant READ_INT_FAIL is returned by read_int() when:
// * the next int could not be successfully read from input, or
// * the end of input (e.g., EOF) is encountered
```

The converse of the C output function `printf` is the input function `scanf`, but we are not quite ready to use it (we introduce `scanf` in Section 05).

example: reading input

```c
// count_even_inputs() counts the number of even
//   values read from input (until a read failure occurs)
// effects: reads input

int count_even_inputs(void) {
  int n = read_int();
  if (n == READ_INT_FAIL) {
    return 0;
  } else if (n % 2 == 0) {
    return 1 + count_even_inputs();
  } else {
    return count_even_inputs();
  }
}

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

example: reading input (continued)

If we **[RUN]** our program in `Seashell`, we can interactively enter `int` values via the keyboard.

To indicate that there is no more input, press the **[EOF]** (**E**nd **O**f **F**ile) button, or type **Ctrl-D**.

example: reading input (continued)

To test our program using **[I/O TEST]** in `Seashell`, we could add the following two test files:

`test1.in`

```
1
2
2
3
4
4
5
6
```

`test1.expect`

```
5
```

example: reading input (continued)

One of the great features of the **[I/O TEST]** in `Seashell` is that you can add **multiple** test files.

`test2.in`

```
1 3 3 7
```

`test2.expect`

```
0
```

`test3.in`

```
6 6 6
```

`test3.expect`

```
3
```

Input formatting

When C reads in `int` values, it skips over any whitespace (newlines and spaces).

The input:

```
1
2
3
4
5
```

and:

```
1 2      3
4     5
```

are indistinguishable to a function like `read_int`.

Reading input

When reading input in our `Seashell` environment you have to be careful: once you read in a value, **it can no longer be read again**.

Typically, you want to *store* the read value (*e.g.,* returned by `read_int`) in a variable so you can refer to it multiple times.

For example, consider this **incorrect** partial implementation:

```
if (read_int() == READ_INT_FAIL) {   // Bad!
  return 0;
} else if (read_int() % 2 == 0) {    // Bad!
  //...
```

The first `read_int()` reads in the first `int`, but then that value is now *"lost"*. The next `read_int()` reads in the *second* `int`, which is not likely the desired behaviour.

Invalid input

In this course, unless otherwise specified, you do **not** have to worry about us testing your code with **invalid input files**.

The behaviour of `read_int` on invalid input can be a bit tricky (see CP:AMA 3.2). For example, for the input:

```
4 23skidoo 57
```

- the first call to `read_int()` returns 4
- the second call to `read_int()` returns 23
- any additional calls to `read_int()` return `READ_INT_FAIL`.

Documenting side effects

So far, functions can only have three possible side effects:

- produce output
- read input
- mutate a global variable

You do not have to provide specifics in the “effects” section of the contract. Simply document the existence of the side effects.

example: documenting side effects

If the side effect does not always occur, preface it with “may” in your contract.

```c
// effects: reads input
//          may produce output
//          may mutate secret

void update_secret(void) {
  int n = read_int();
  if (n == READ_INT_FAIL) {
    printf("error: could not read in number\n");
  } else {
    secret = n;
  }
}
```

Goals of this Section

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

- explain what a side effect is
- document a side effect with an *effects* section
- print output with `printf` and read input using the provided functions (*e.g.*, `read_int`)
- define global and local mutable variables and constants
- use the C assignment operators
- use the new terminology introduced, including: mutation, expression statements and compound statements (`{}`)

C Model: Memory & Control Flow

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

Course Notes: Memory Appendix

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

The primary goal of this section is to be able to model how C programs execute.

Models of computation

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

To *call* (“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 two complimentary mechanisms:

- **control flow**
- **memory**

Control flow

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).

Types of control flow

In this course, we explore three types of control flow:

- function calls
- conditionals (*i.e.,* `if` statements)
- iteration (*i.e.,* loops)

```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.

Return

The `return` control flow statement changes the program location to go *back* to the **most recent** calling function.

Obviously, C needs to “keep track” of where to go.

We revisit this when we introduce memory later in this section.

Conditionals (if)

We introduced the `if` control flow statement in Section 02. We now discuss `if` in more detail.

The syntax of `if` is

```
  if (expression) statement
```

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

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

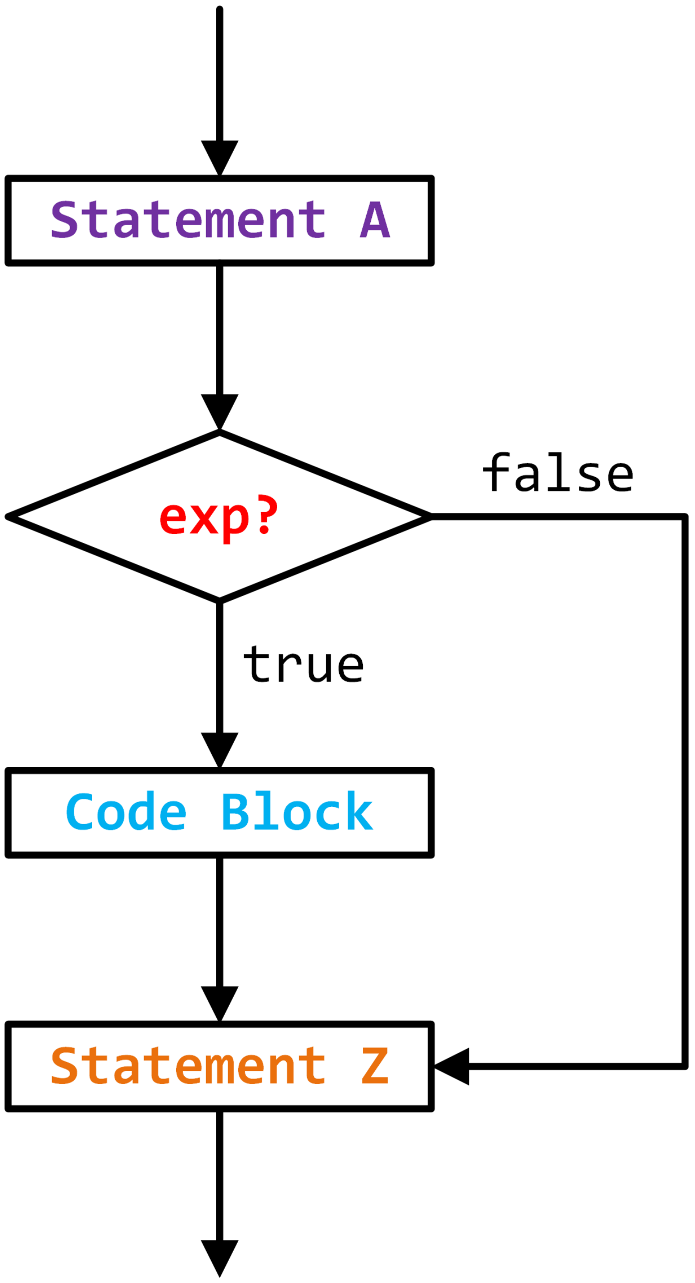
Remember: the `if` statement does not produce a value. It only controls the flow of execution.

The `if` statement only affects whether the *next* statement is executed. To conditionally execute more than one statement, braces (`{}`) are used to insert a compound statement block (a sequence of statements) in place of a single statement.

```
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.)*

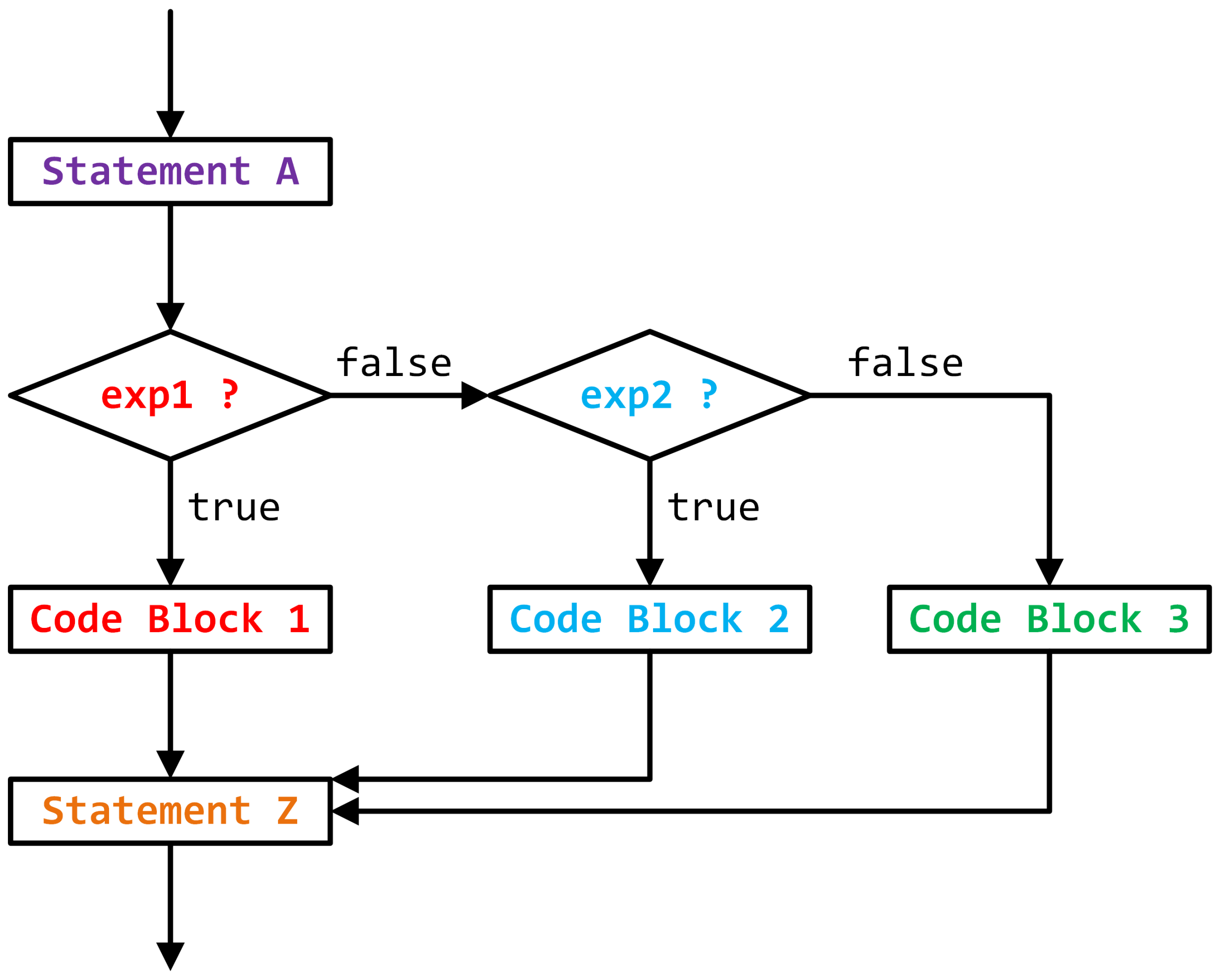
```
if (n <= 0) {
  printf("n is less than or equal to zero\n");
}
```



```
Statement A;
if (exp) {
  Code Block;
}
Statement Z;
```

As we have seen, the `if` statement can be combined with `else` statement(s) for multiple conditions.

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



```
Statement A;
if (exp1) {
  Code Block 1;
} else if (exp2) {
  Code Block 2;
} else {
  Code Block 3;
}
Statement Z;
```

If an `if` condition `return`s, there may be no need for an `else`.

```
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 != 7)
    printf("you lose");
else
  printf("you win!");  // when does this print?
```

The C `switch` control flow statement (see CP:AMA 5.3) has a similar structure to `else if` and `cond`, but very different behaviour.

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

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

Do not use `switch` in this course.

The C `goto` control flow statement (CP:AMA 6.4) is one of the most disparaged language features in the history of computer science because it can make *"spaghetti code"* that is hard to understand.

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

To use `goto`s, you must also have *labels* (code locations).

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

Do not use `goto` in this course.

Looping

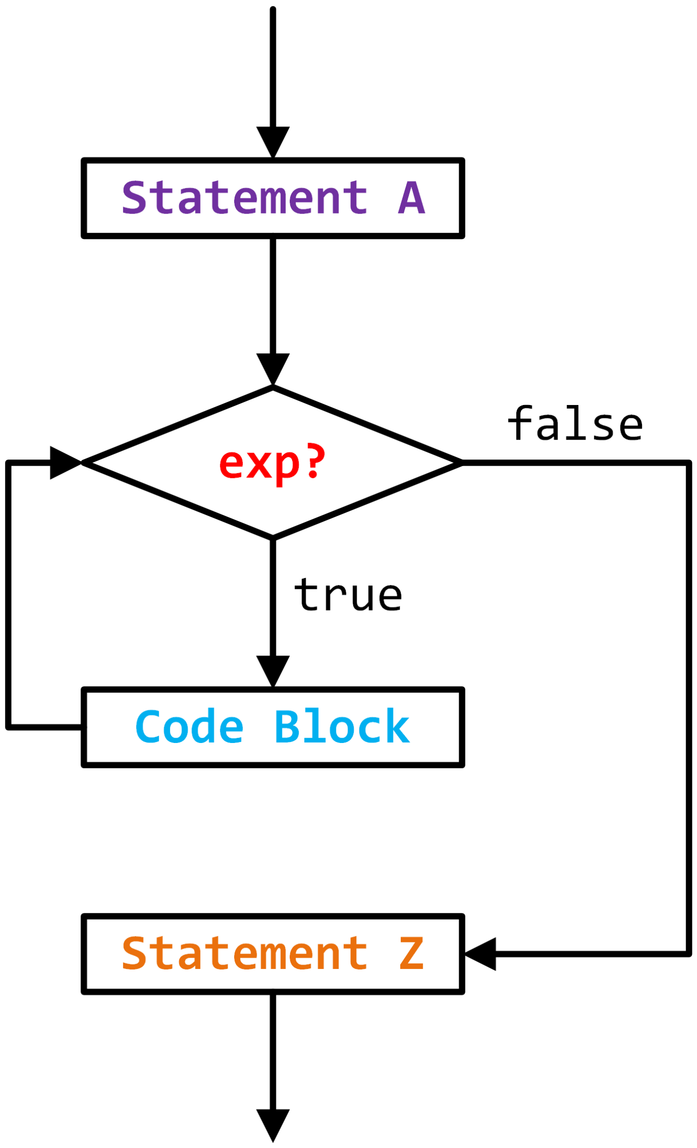
With mutation, we can control flow with a method known as ***looping***.

```
  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.



```
Statement A;
while (exp) {
  Code Block;
}
Statement Z;
```

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*.

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

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

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

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

Loops can be “nested” within each other.

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

```
******
*****
****
***
**
*
```

while errors

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

```
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
```

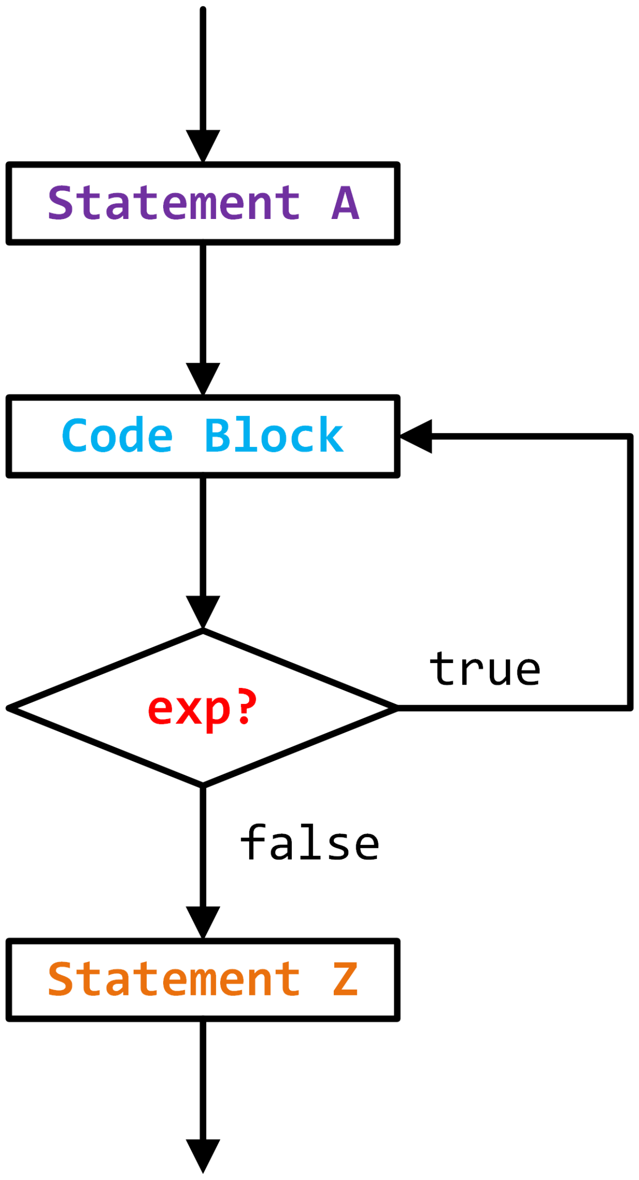
do ... while

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.

```
do {
  printf("try to guess my number!\n");
  guess = read_int();
} while (guess != my_number && guess != READ_INT_FAIL);
```



```
Statement A;
do {
   Code Block;
} while (exp);
Statement Z;
```

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.

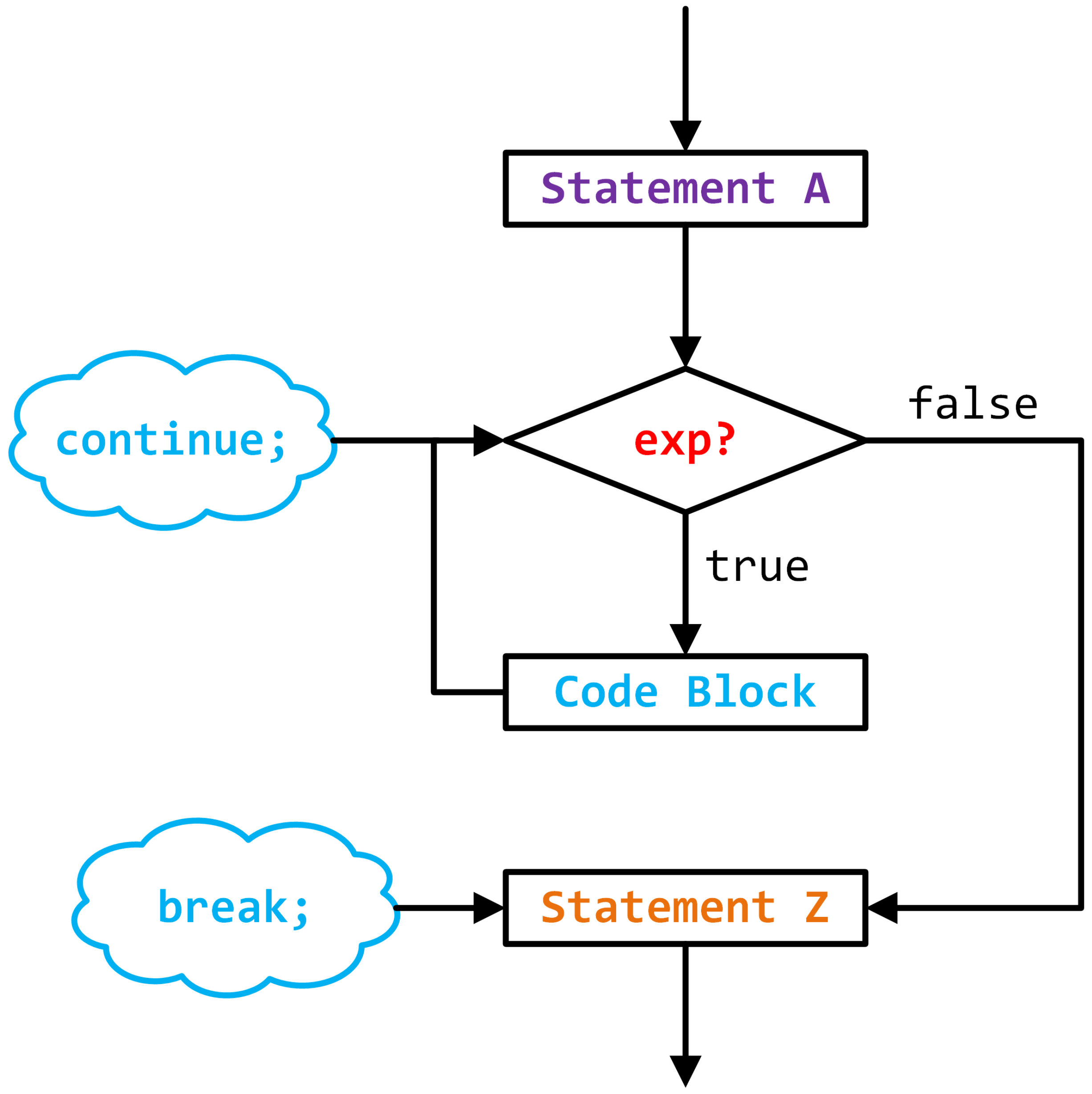
```
while (1) {
  n = read_int();
  if (n == READ_INT_FAIL) break;
  //...
}
```

`break` only terminates loops. You cannot `break` out of an `if`.

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
while (1) {
  n = read_int();
  if (n == READ_INT_FAIL) break;
  if (!is_fun(n)) continue;
  //...
}
```



```
Statement A;
while (exp) {
  Code Block;
}
Statement Z;
```

for loops

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

`for` loops are a "condensed" version of a `while` loop.

The format of a `while` loop is often of the form:

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

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

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

for vs. while

Recall the `for` syntax.

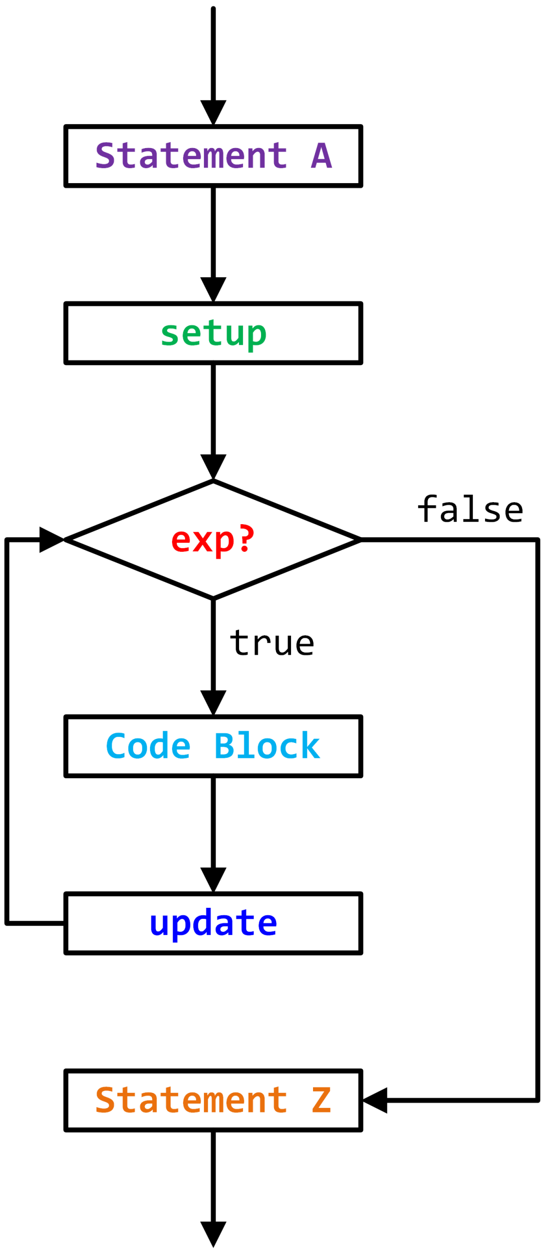
```
for (setup; expression; update) { body statement(s) }
```

This `while` example

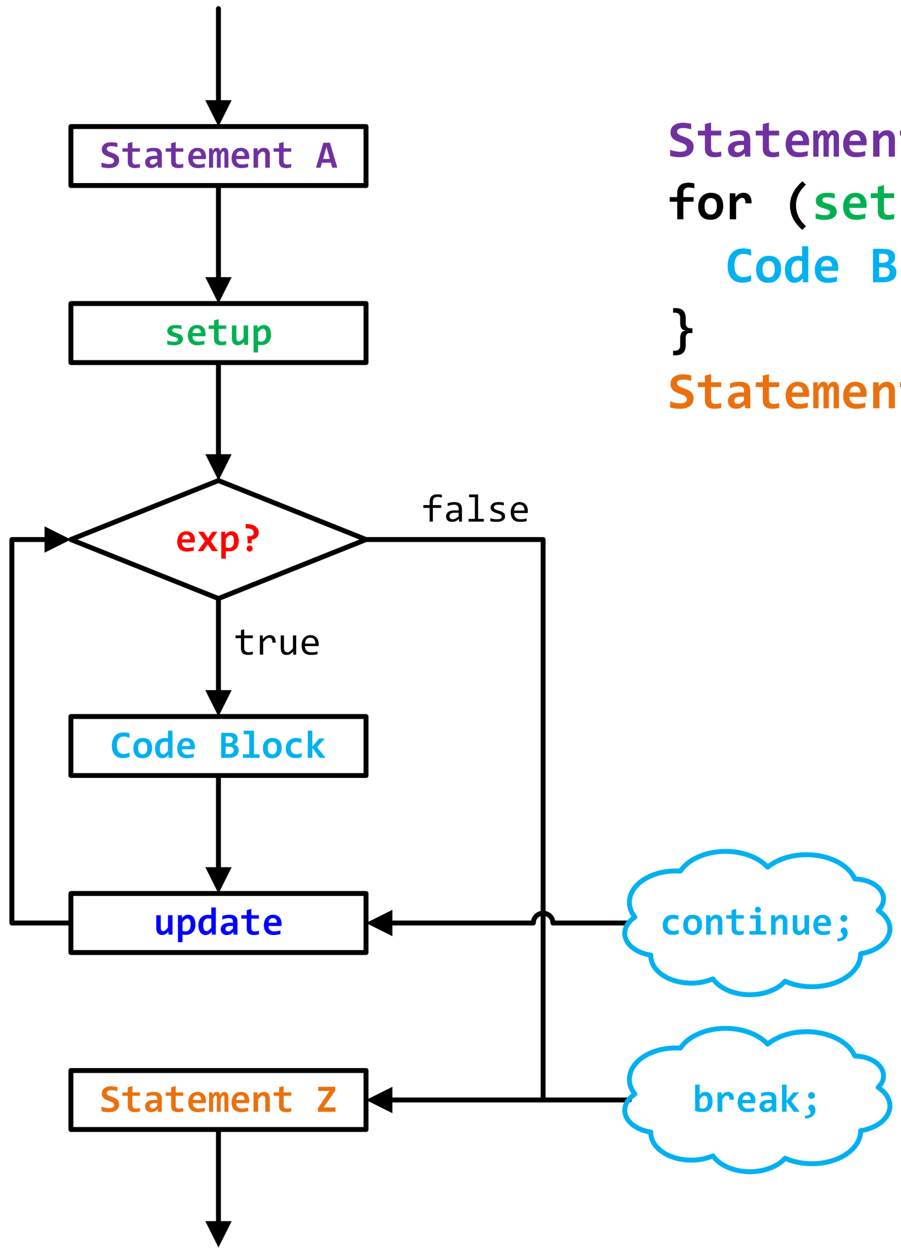
```
i = 100;                        // setup
while (i >= 0) {                // expression
  printf("%d\n", i);
  --i;                          // update
}
```

is equivalent to

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



```
Statement A;
for (setup; exp; update) {
  Code Block;
}
Statement Z;
```



```
Statement A;
for (setup; exp; update) {
  Code Block;
}
Statement Z;
```

Most `for` loops follow one of these forms (or “idioms”).

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

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

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

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

It is a common mistake to be “off by one” (*e.g.*, using < instead of <=). Sometimes re-writing as a `while` is helpful.

In C99, the *setup* statement can be a **definition**.

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

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

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 was setup previously

for (; i < 100;) {...}      // same as a while(i < 100)

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

You can use the *comma operator* (`,`) to use more than one expression in the *setup* and *update* statements of a `for` loop.

See CP:AMA 6.3 for more details.

```
for (i = 1, j = 100; i < j; ++i, --j) {...}
```

A `for` loop is *not always* equivalent to a `while` loop.

The only difference is when a `continue` statement is used.

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

In a `for` loop, the “update” statement is executed before jumping back to the expression.

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 the Appendix on Memory

Accessing memory

The smallest accessible unit of memory is a byte.

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

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

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

You can visualize 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`

The *contents* in the above table are arbitrary values.

Defining variables

When C encounters a **variable definition**, it

- reserves (or “finds”) space in memory to ***store*** the variable
- “keeps track of” the *address* of that storage location
- stores the initial value of the variable at that location (address).

For example, with the definition

```
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.

In our CS 135 substitution model, a variable is a “name for a value”.

When a variable appears in an expression, a *substitution* occurs and the name is *replaced* by its value.

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

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

sizeof

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

It depends on the **type** of the variable.

It may also depend on the *environment* (the machine and compiler).

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

```
int n = 0;
trace_int(sizeof(int));
trace_int(sizeof(n));
```

```
sizeof(int) => 4
sizeof(n) => 4
```

In this course, the size of an `int`eger is 4 bytes (32 bits).

In C, the size of an `int` depends on the machine (processor) and/or the operating system that it is running on.

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

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

In this course, you should only use `int`, and there are always 32 bits in an `int`.

example: variable definition

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

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

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.

In our CS 136 environment, 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.

In the `read_int` function we provide, the value of the constant `READ_INT_FAIL` is actually `INT_MIN`, so the smallest value of `int` that can be successfully read by our `read_int` function is $-2{,}147{,}483{,}647$.

Overflow

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

You should never assume what the value of an `int` will be after an overflow occurs.

The value of an integer that has overflowed is **undefined**.

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

In CS 251 / CS 230 you learn more about overflow.

example: overflow

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

trace_int(bil);
trace_int(four_bil);
trace_int(nine_bil);
```

```
bil => 1000000000
four_bil => -294967296
nine_bil => 410065408
```

Remember, do not try to “deduce” what the value of an `int` will be after overflow – its behaviour is undefined.

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`).

Additional types

Now that we have a better understanding of what an `int` in C is, we introduce some additional types.

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;
  }
}
```

Reading characters from input

In Section 03, we used the `read_int` function to read integers from input.

We have also provided `read_char` for reading characters.

When reading `int` values, we ignored whitespace in the input.

When reading in characters, you **may** or **may not** want to ignore whitespace characters, depending on your application.

`read_char` has a parameter for specifying if whitespace should be ignored.

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".

```
const int POP = 1;       // instead of 'pop
const int ROCK = 2;      // instead of 'rock

int my_favourite_genre = POP;
```

It is common to use an alternative naming convention (such as `ALL_CAPS`) when using numeric constants to represent symbols.

We have provided some tools for working with C "symbols" on your assignments.

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.

Floating point types

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

```
float pi = 3.14159;
float avogadro = 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

```
float penny = 0.01;
float money = 0;

for (int n = 0; n < 100; ++n) {
  money += penny;
}

printf("the value of one dollar is: %f\n", money);
```

```
the value of one dollar is: 0.999999
```

The `printf` placeholder to display a `float` is `"%f"`.

example 2: inexact floats

```
float bil = 1000000000;
float bil_and_one = bil + 1;

printf("a float billion is:     %f\n", bil);
printf("a float billion + 1 is: %f\n", bil_and_one);
```

```
a float billion is:     1000000000.000000
a float billion + 1 is: 1000000000.000000
```

In the previous two examples, we highlighted the precision errors that can occur with the `float` type.

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

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

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

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

Floats in memory

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

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

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

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

`float`s and their internal representation are discussed in CS 251 / 230 and in detail in CS 370 / 371.

Structures

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

```
struct posn {          // name of the structure
  int x;               // type and field names
  int y;
};                     // don't forget this ;
```

Because C is statically typed, structure definitions require the *type* of each field.

Do not forget the last semicolon (`;`) in the structure definition.

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 below.

```
struct posn p = {3, 4};    // note the use of {}

trace_int(p.x);
trace_int(p.y);
```

```
p.x => 3
p.y => 4
```

Instead of *selector functions*, C has a ***structure operator*** (`.`) which “selects” the requested field.

The syntax is `variablename.fieldname`

C99 supports an alternative way to initialize structures:

```
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:

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

Mutation with structures

The assignment operator can be used with `struct`s to copy all of the fields from another `struct`. Individual fields can also be mutated.

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

p = q;
p.x = 23;

trace_int(p.x);
trace_int(p.y);
```

```
p.x => 23
p.y => 4
```

The braces (`{}`) are **part of the initialization syntax** and can not simply be used in assignment. Instead, just mutate each field.

On rare occasions, you may want to define a new `struct` so you can mutate "all at once".

```
struct posn p = {1, 2};

p = {5, 6};                     // INVALID

p.x = 5;                        // VALID
p.y = 6;

// alternatively:
struct posn new_p = {5, 6};
p = new_p;
```

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

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

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

```
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)
```

Structures in the memory model

For a structure *definition*, no memory is reserved:

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

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

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

sizeof a struct

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

The amount of space reserved for a `struct` is **at least** the sum of the `sizeof` each field, but it may be larger.

```
trace_int(sizeof(struct mystruct));
```

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

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

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

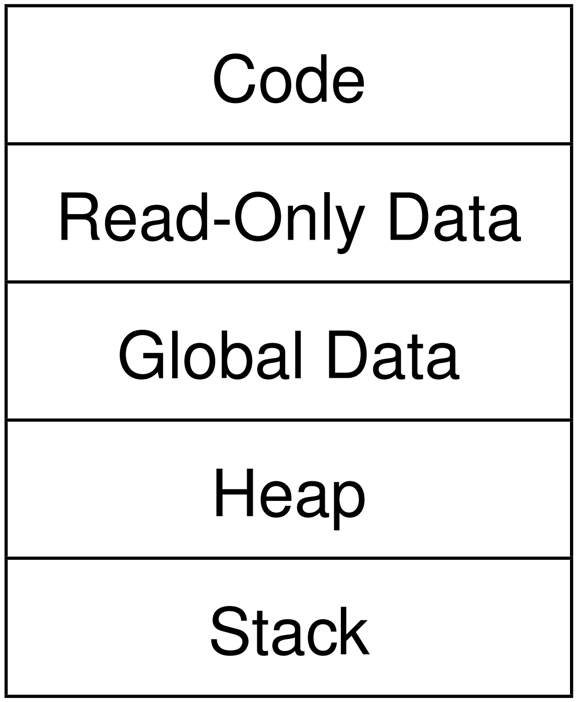
```
struct s1 {                    struct s2 {
  char c;                        char c;
  int i;                         char d;
  char d;                        int i;
};                             };

trace_int(sizeof(struct s1));
trace_int(sizeof(struct s2));
```

```
sizeof(struct s1) => 12
sizeof(struct s2) => 8
```

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

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).

Temporary results

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 are not concerned with this “temporary” storage.

Temporary storage is discussed in CS 241.

The code section

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

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

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

Converting source code into machine code is known as ***compiling***. It is briefly discussed in Section 13 and covered extensively in CS 241.

The read-only & global data sections

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

```
int n = 0;
```

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

First, we discuss global variables.

All global variables are placed in either the **read-only data** section (**constants**) or the **global data** section (**mutable variables**).

Global variables are available throughout the entire execution of the program, and the space for the global variables is reserved **before** the program begins execution.

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

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

The return address

When we encounter a `return`, we need to know: “what was the address we were at right before this function was called?”

In other words, 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 calling function and a line number (or an arrow) to represent the return address.

In practice, the *return address* is the address in the machine code immediately 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**
- all **local variables** (both mutable variables and constants) that appear within the function *block* (including any sub-blocks)
- the **return address**

The *return address* is a location from inside the *calling function*.

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

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

This is known as the “pass by value” convention.

Whereas space for a *global* variable is reserved *before* the program begins execution, space for a *local* variable is only reserved **when the function is called**.

The space is reserved within the newly created stack frame.

When the function `return`s, the variable (and the entire frame) is popped and effectively “disappears”.

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

```
 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    int d = g(b + 3) + h(b);
14    return d;
15  }
16
17  int main(void) {
18    int a = f(2);
19    //...
20  }
```

```
=========================
g:
  y: 8
  c: 64
  return address: f:13
=========================
f:
  x: 2
  b: 5
  d: ???
  return address: main:18
=========================
main:
  a: ???
  return address: OS
=========================
```

In void functions the return is optional, so a return automatically occurs when the end of the function block is reached.

```
void print_size(int n) {
  if (n > 1000000) {
    printf("n is huge\n");
  } else if (n > 10) {
    printf("n is big\n");
  } else {
    printf("n is tiny\n");
  }
}
```

Calling a function

We can now model all of the **control flow** when a function is called:

- a *stack frame* is created ("pushed" onto the Stack)
- a **copy** of each of the arguments is placed in the stack frame
- the current program location is placed in the stack frame as the *return address*
- the program location is changed to the start of the new function
- the initial values of local variables are set when their definition is encountered

return

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 is discussed further in CS 241.

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 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 allows you to define variables without any initialization.

```
int i;
```

For all **global** variables, C automatically initializes 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;
```

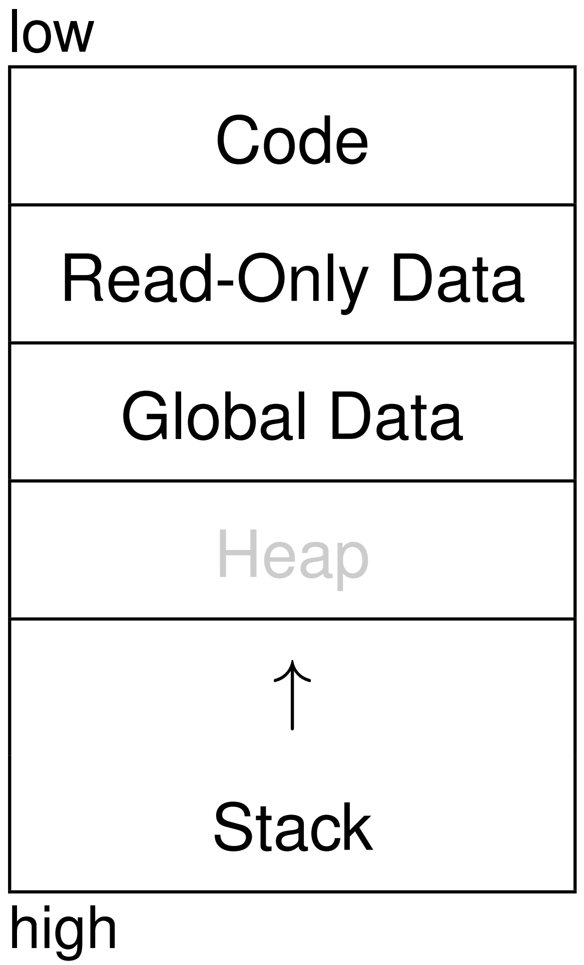
A **local** variable (on the *stack*) that is uninitialized has an **arbitrary** initial value.

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

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

In the example above, the value of `k` will likely be a leftover value from a previous stack frame.

Memory sections (so far)



Memory snapshot

You may be asked to draw a memory diagram (including the call stack) at a particular moment in the code execution.

For example, “draw the memory when line 19 is reached”.

- make sure you show any variables in the **global** and **read-only** sections, *separate* from the **stack**
- include *all* local variables in stack frames, including definitions that have not yet been reached (or are incomplete)
- local variables not yet fully initialized have a value of ???
- you do not have to show any *temporary* storage (*e.g.,* intermediate results of an expression)

When a variable is defined **inside of a loop**, only one occurrence of the variable is placed in the stack frame. The same variable is *re-used* for each iteration.

Each time the definition is reached in the loop, the variable is **re-initialized** (it does not retain its value from the previous iteration).

```
for (int j = 0; j < 3; ++j) {
  int k = 0;
  k = k + j;
  trace_int(k);
}
```

```
k => 0
k => 1
k => 2
```

Model

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

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

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

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

Goals of this Section

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

- use the introduced control flow statements, including (`return`, `if`, `while`, `do`, `for`, `break`, `continue`)
- re-write a recursive function with iteration and *vice versa*
- explain why C has limits on integers and why overflow occurs
- use the `char` type and explain how characters are represented in ASCII
- use structures in C

- 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
- model the execution of small programs by hand, and draw the stack frames at specific execution points

Introduction to Pointers in C

Readings: CP:AMA 11, 17.7

The primary goal of this section is to be able use 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 **location** of an identifier in memory (the **starting address** of where its value is stored).

```
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: 0x71a0a0
```

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

Pointers

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

A pointer is defined by placing a *star* (`*`) *before* the identifier (name).

The `*` is part of the definition syntax, not the identifier itself.

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

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

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

This definition:

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

is comparable to the following definition and assignment:

```
int *p;         // p is defined (not initialized)
p = &i;         // p now "points at" i
```

The * is part of the definition of p and is **not part of the variable name**. The name of the variable is simply p, not *p.

As with any variable, its value can be changed.

```
p = &j;         // p now "points at" j
p = &i;         // p now "points at" i
```

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

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

trace_int(i);
trace_ptr(&i);
trace_ptr(p);
trace_ptr(&p);
```

```
i => 42
&i => 0xf020
p => 0xf020
&p => 0xf024
```

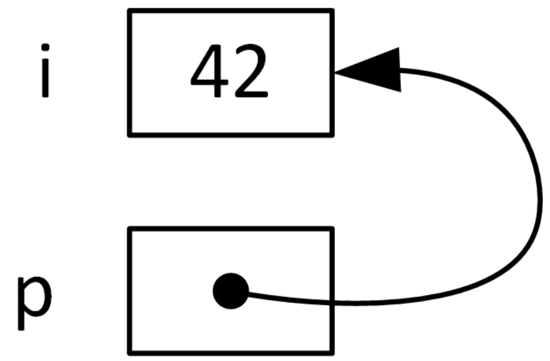
To make working with pointers easier in these notes, we often use shorter, simplified (“fake”) addresses.

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

identifier	type	address
`i`	`int`	`0xf020`
`p`	`int *`	`0xf024`

value
`42`
`0xf020`

When drawing a *memory diagram*, we rarely care about the value of the address, and visualize a pointer with an arrow (that “points”).



sizeof a pointer

In most k-bit systems, memory addresses are k bits long, so pointers require k bits to store an address.

In our 64-bit `Seashell` environment, the `sizeof` a pointer is always 64 bits (8 bytes).

The `sizeof` a pointer is **always the same size**, regardless of the type of data stored at that address.

```
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

trace_ptr(p);
trace_int(*p);
```

```
p => 0xf020
*p => 42
```

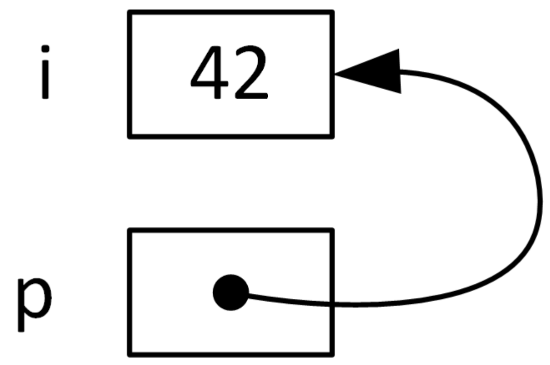
The value of ∗&i is simply the value of i.

The **address operator (&)** can be thought of as:

"get the address of this box".

The **indirection operator ($*$)** can be thought of as:

"follow the arrow to the next box and get its contents".



$*p \Rightarrow 42$

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

- as the *multiplication operator* between expressions

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

- in pointer *definitions* and pointer *types*

```
int *p = &i;
sizeof(int *)
```

- as the *indirection operator* for pointers

```
j = *p;
*p = 5;
```

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

C mostly ignores whitespace, so these are equivalent

```
int *p = &i;      // style A
int * p = &i;     // style B
int* p = &i;      // style C
```

There is some debate over which is the best style. Proponents of style B & C argue it's clearer that the type of `p` is an “`int *`”.

However, *in the definition* the `*` “belongs” to the `p`, not the `int`, and so style A is used in this course and in CP:AMA.

This is clear with multiple definitions: (not encouraged)

```
int i = 42, j = 23;
int *p1 = &i, *p2 = &j; // VALID
int * p1 = &i, p2 = &j;  // INVALID: p2 is not a pointer
```

Pointers to pointers

A common question is: *"Can a pointer point at itself?"*

```
int *p = &p;          // pointer p points at p ???
```

This is actually a **type error**:

- `p` is defined as `(int *)`, a pointer to an `int`, but
- the type of `&p` is `(int **)`, a pointer to a pointer to an `int`.

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

```
int i = 42;
int *p1 = &i;       // pointer p1 points at i
int **p2 = &p1;     // pointer p2 points at p1
```

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

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

A `void` pointer (`void *`) can point at anything, including a `void` pointer (itself).

The NULL pointer

`NULL` is a special pointer **value** to represent that the pointer points to "nothing".

If the value of a pointer is unknown at the time of definition, or what the pointer points at becomes *invalid*, it's good style to assign the value of `NULL` to the pointer.

```
int *p;                // BAD (uninitialized)

int *p = NULL;         // GOOD
```

Some functions return a `NULL` pointer to indicate an error.

`NULL` is considered “false” when used in a Boolean context (`false` is defined to be zero *or* `NULL`).

The following two are equivalent:

```
  if (p) ...

  if (p != NULL) ...
```

If you try to *dereference* a `NULL` pointer, your program will crash.

```
  p = NULL;
  i = *p;       // crash!
```

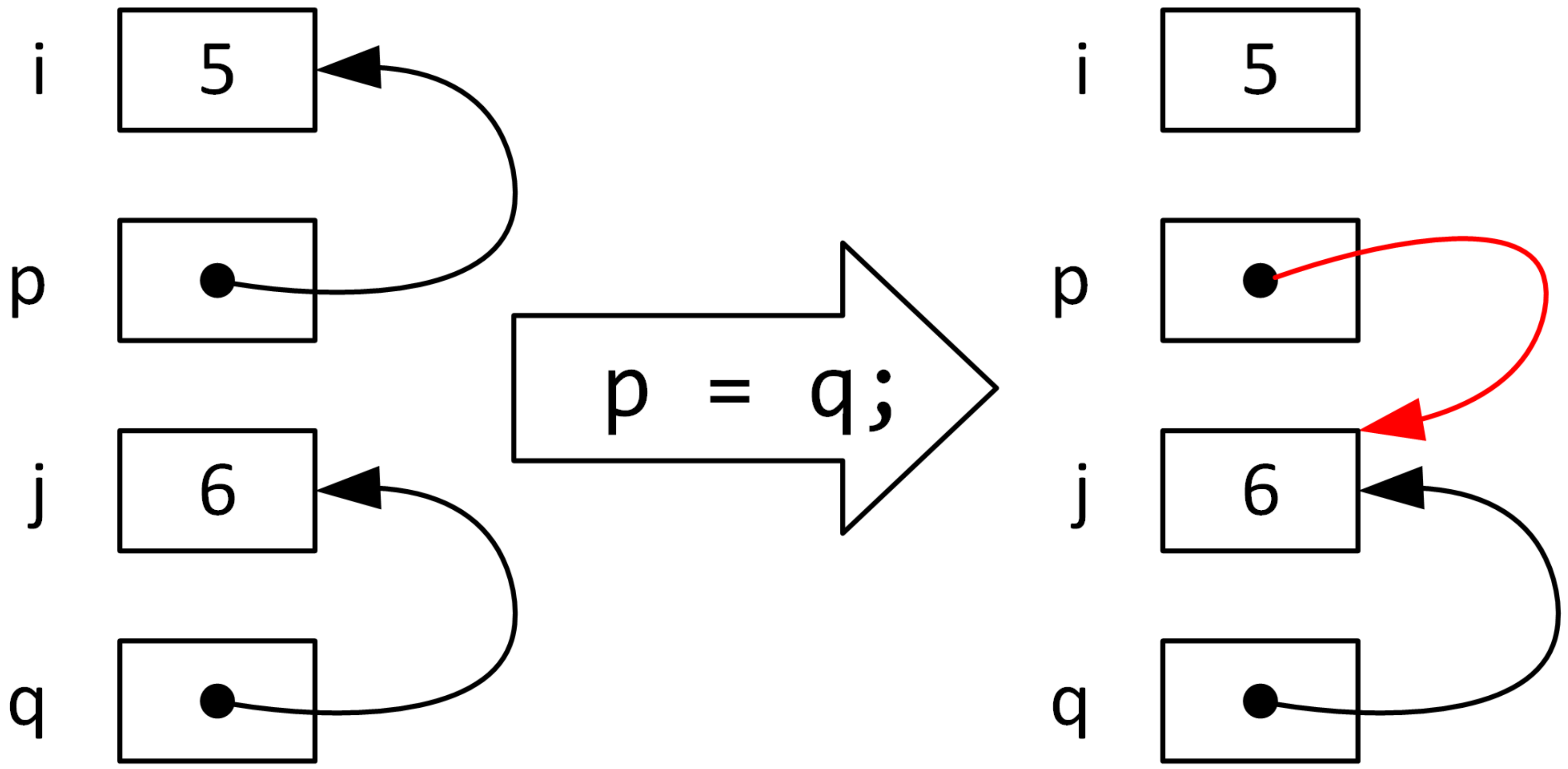
Pointer assignment

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 “change `p` to point at what `q` points at”. It changes the *value* of `p` to be the value of `q`. In this example, it assigns the *address* of `j` to `p`.

It **does not change the value of `i`**.


i
5
p
j
6
q
p = q;
i
5
p
j
6
q

Using the same initial values,

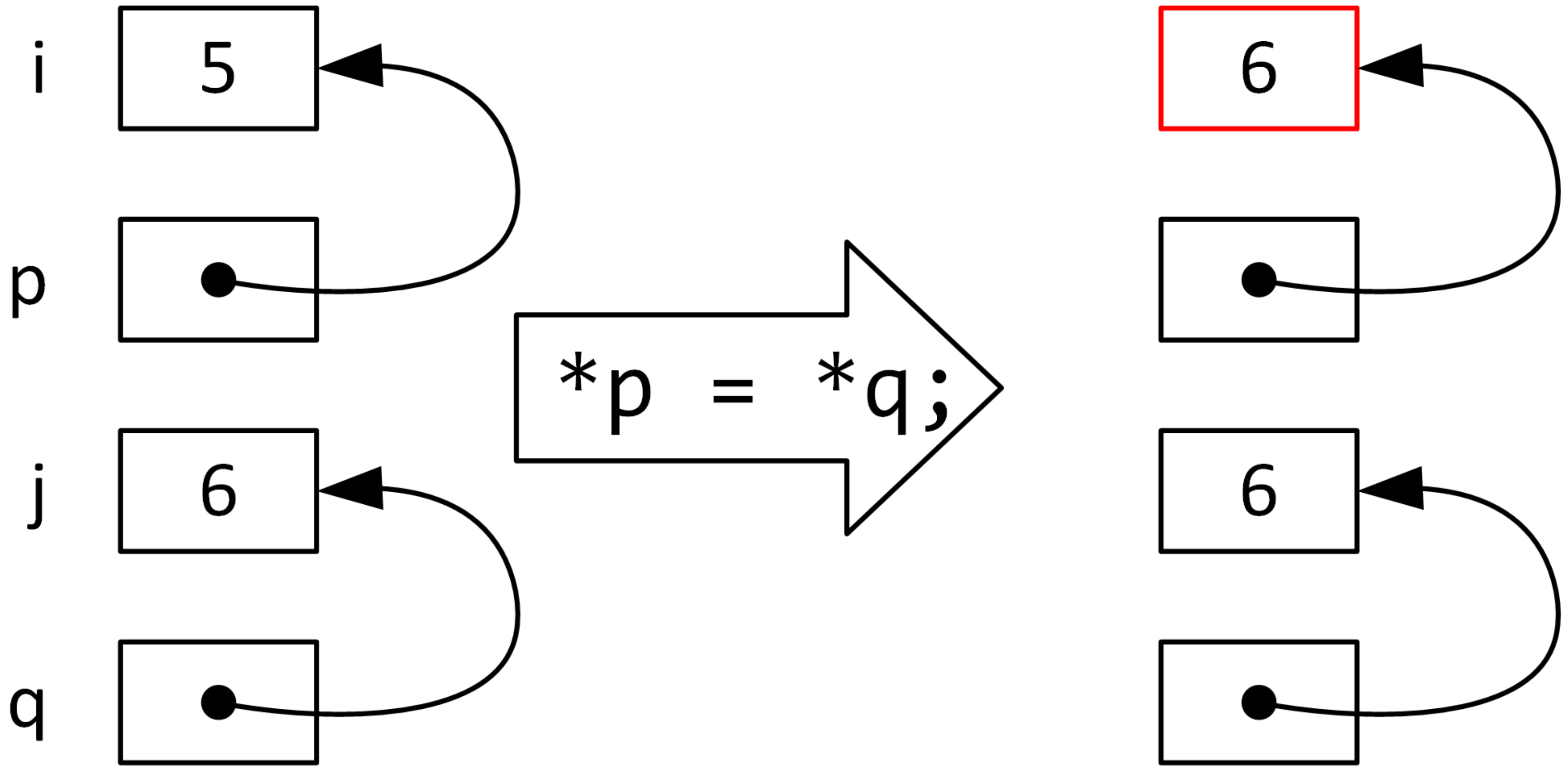
```
int i = 5;
int j = 6;

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.


i
5
p
j
6
q
*p = *q;
6
6

example: aliasing

```
int i = 1;
int *p1 = &i;
int *p2 = p1;
int **p3 = &p1;

trace_int(i);
*p1 = 10;                      // i changes...
trace_int(i);
*p2 = 100;                    // without being used directly
trace_int(i);
**p3 = 1000;
trace_int(i);
```

```
i => 1
i => 10
i => 100
i => 1000
```

Mutation & parameters

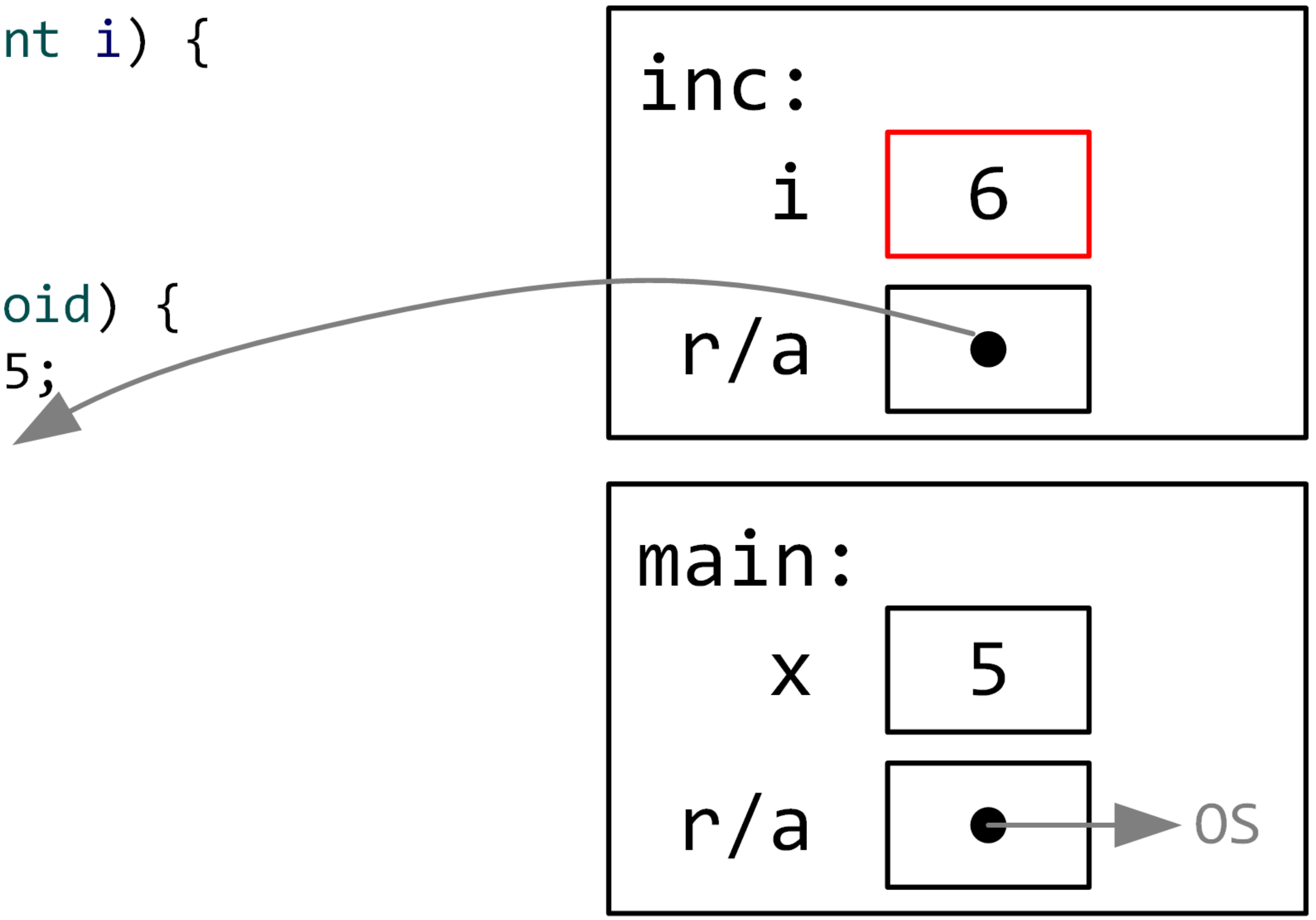
Consider the following C program:

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

int main(void) {
  int x = 5;
  inc(x);
  trace_int(x);     // 5 or 6 ?
}
```

It is important to remember that when `inc(x)` is called, a **copy** of `x` is placed in the stack frame, so `inc` cannot change `x`.

The `inc` function is free to change it's own copy of the argument (in the stack frame) without changing the original variable.

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

int main(void) {
  int x = 5;
  inc(x);
}
```



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 actually “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”.

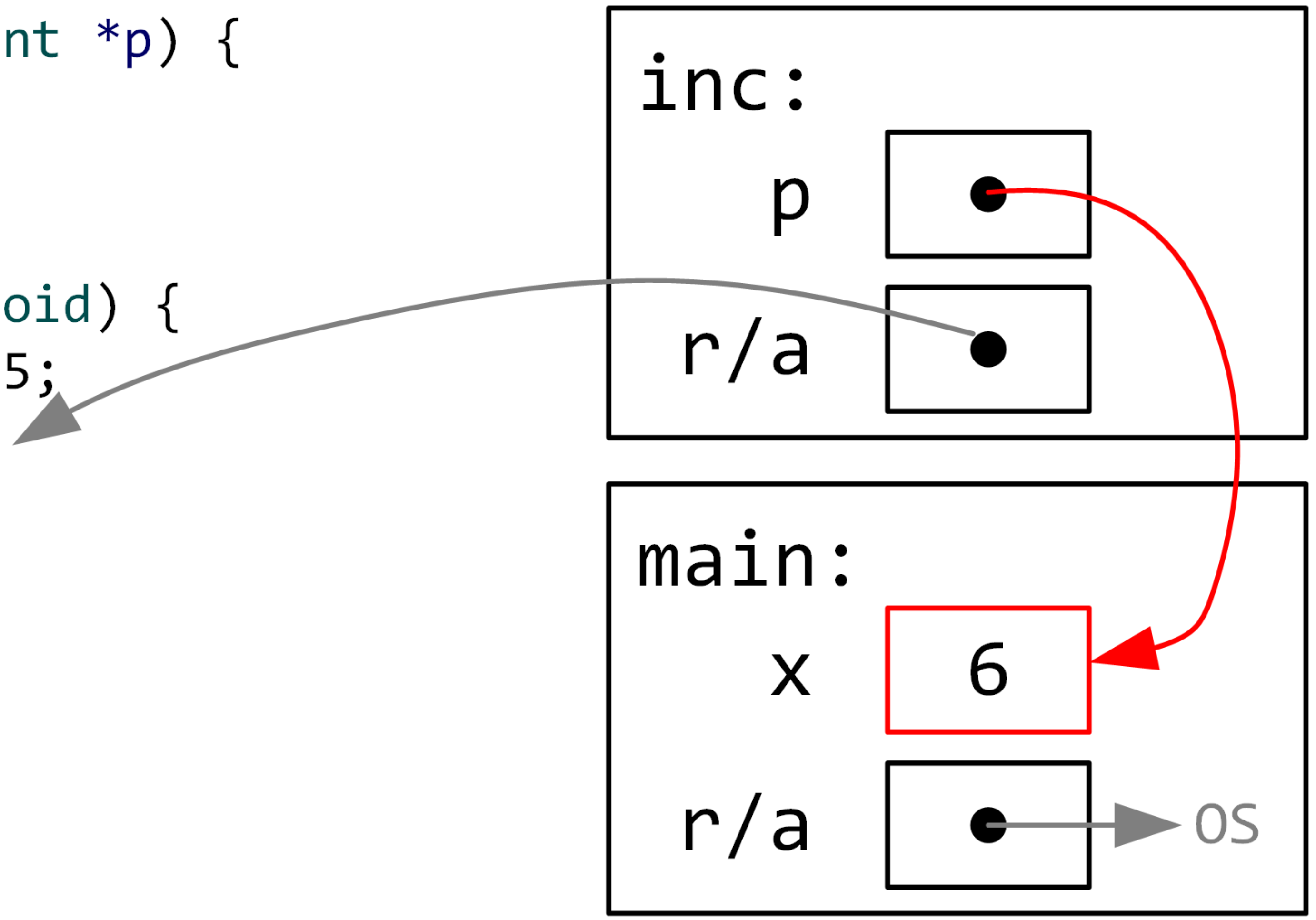
```
void inc(int *p) {
  *p += 1;
}

int main(void) {
  int x = 5;
  trace_int(x);
  inc(&x);                          // note the &
  trace_int(x);
}
```

```
x => 5
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;
}

int main(void) {
  int x = 5;
  inc(&x);
}
```



Whenever you have a parameter that is a pointer, you should **require** that it is a valid (*e.g.*, non-`NULL`) pointer.

```
// inc(p) increments the value of *p
// effects:  modifies *p
// requires: p is a valid pointer

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: `++` would have incremented the pointer `p`, not what it points at (`*p`).

example: mutation side effects

```c
//  effects: modifies *px and *py
void swap(int *px, int *py) {
  int temp = *px;
  *px = *py;
  *py = temp;
}

int main(void) {
  int a = 3;
  int b = 4;
  trace_int(a); trace_int(b);
  swap(&a, &b);                       // Note the &
  trace_int(a); trace_int(b);
}
```

```
a => 3
b => 4
a => 4
b => 3
```

Documenting side effects

We now have a fourth side effect that a function may have:

- produce output
- read input
- mutate a global variable
- **mutate a variable through a pointer parameter**

```
//  effects: modifies *px and *py
void swap(int *px, int *py) {
  int temp = *px;
  *px = *py;
  *py = temp;
}
```

In the *functional paradigm*, there is no observable difference between “pass by value” and “pass by reference”.

In Racket, simple values (*e.g.,* numbers) are passed by *value*, but structures are passed by *reference*.

C input: scanf

So far we have been using our tools (*e.g.*, `read_int`) to read input.

We are now capable of using the built-in `scanf` function.

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

`scanf` requires a **pointer** to a variable to **store** the value read in from input.

Just as with `printf`, you use multiple placeholders to read in more than one value.

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

This will help you debug your code and facilitate our testing.

The **return value** of `scanf` is the number (count) of values *successfully read*.

The return value can also be the special constant value `EOF` to indicate that the **E**nd **O**f **F**ile (EOF) has been reached.

A `Ctrl-D` (“Control D”) keyboard sequence sends an `EOF`.

Always check the return value of `scanf`: one is “success”.

(if you’re following our advice to read one value per `scanf`).

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

if (retval != 1) {
  printf("Fail! I could not read in an integer!\n");
}
```

As with our `read_int`, `scanf("%d", &i)` **ignores whitespace** (spaces and newlines) when reading the next integer.

If the next non-whitespace input to be read is not a valid integer (e.g., a letter), it stops reading and returns zero.

When reading in a `char`, you may or may not want to ignore whitespace.

```
// 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.

Returning more than one value

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 those 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;
}
```

Here is an example of how it can be used:

```
divide(13, 5, &q, &r);
trace_int(q);
trace_int(r);
```

```
q => 2
r => 3
```

This “multiple return” technique is also 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.

Returning an address

In Section 10, we use functions that return an address (pointer).

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 d; int e; ... 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 very common to pass the *address* of a structure to a function.

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

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

  trace_int(sqr_dist(&p1, &p2));      // note the &
}
```

```
sqr_dist(&p1, &p2) => 25
```

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

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 awkward.

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) {
  int xdist = p1->x - p2->x;
  int ydist = p1->y - p2->y;
  return xdist * xdist + ydist * ydist;
}
```

Passing the address of a structure to a function (instead of a copy) also allows the function to mutate the fields of the structure.

```
// scale(p, f) scales the posn p by f
// requires: p is not null
// effects:  modifies p

void scale(struct posn *p, int f) {
  p->x *= f;
  p->y *= f;
}
```

In the above documentation, we used p, where *p would be more correct. It is easily understood that p represents the structure.

```
// this is more correct, but unnecessary:

// scale(p, f) scales the posn *p by f
// effects:  modifies *p
```

We now have **two** different reasons for passing a structure pointer to a function:

- to avoid copying the structure
- to mutate the contents of the structure

It would be good to communicate whether or not there is a side effect (mutation).

However, documenting the **absence** of a side effect (“no side effect here”) is 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(const struct posn *p) {
  p->x = 5;    // INVALID
}
```

The `const` should be placed first, before the type.

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 point at any mutable int,
                        // you can modify the int (via *p)

const int *p;           // p can point at any int,
                        // you can NOT modify the int via *p

int * const p = &i;     // p always points at i, i must be
                        // mutable and can be modified via *p

const int * const p = &i;  // p must always point at i
                           // you can not modify i via *p
```

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”.

```
const int i = 42;        // these are equivalent
int const i = 42;        // but this form is discouraged
```

const parameters

As we just established, it is good style to use `const` with pointer parameters to communicate that the function does not (and can not) mutate the contents of the pointer.

```
void can_change(struct posn *p) {
  p->x = 5;   // VALID
}

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

What does it mean when `const` is used with simple (non-pointer) parameters?

For a simple value, the `const` keyword indicates that the parameter is immutable *within the function*.

```
int my_function(const int x) {
  // mutation of x here is invalid
  // ...
}
```

It does not require that the argument passed to the function is a constant.

Because a **copy** of the argument is made for the stack, it does not matter if the original argument value is constant or not.

A `const` parameter communicates (and enforces) that **the copy** of the argument will not be mutated.

Minimizing mutative side effects

In Section 03 we used *mutable* global variables to demonstrate mutation and how functions can have mutative side effects.

In practice, **mutable** global variables are **strongly discouraged** and considered “bad style”.

They make your code harder to understand, maintain and test.

On the other hand, global **constants** are “good style” and encouraged.

There are rare circumstances where global mutable variables are necessary.

Your preference for function design should be:

1. **No mutative side effects**[†]

   Functions are much easier to understand and test.

2. **Mutate data through pointer parameters**[†]

   When mutation of data is required, use a pointer parameter to access the data.

3. **Mutate global data**

   This should only be used when absolutely necessary.

[†] In addition, the behaviour of a function should not depend on a global mutable variable. Multiple calls to a function with identical arguments and data should have identical results.

Function pointers

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

For example, Racket functions are values that can be stored in variables and data structures, passed as arguments and returned by functions.

In C, functions are not first-class values, but ***function pointers*** are.

A significant difference is that **new** Racket functions can be created during program execution, while in C they cannot.

A function pointer can only point to a function that already exists.

A *function pointer* stores the (starting) address of a function, which is an address in the code section of memory.

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

The syntax to define a function pointer with name `fpname` is:

```
return_type (*fpname)(param1_type, param2_type, ...)
```

In an exam, we would not expect you to remember the syntax for defining a function pointer.

example: function pointer

```
int my_add(int x, int y) {
  return x + y;
}

int my_sub(int x, int y) {
  return x - y;
}

int main(void) {
  int (*fp)(int, int) = NULL;
  fp = my_add;
  trace_int(fp(7, 3));
  fp = my_sub;
  trace_int(fp(7, 3));
}
```

```
fp(7, 3) => 10
fp(7, 3) => 4
```

Goals of this Section

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

- define and dereference pointers
- use the new operators (&, `*`, `->`)
- describe aliasing
- use the `scanf` function to read input

- use pointers to structures as parameters and explain why parameters are often pointers to structures
- explain when a pointer parameter should be `const`
- use function pointers

Modularization & ADTs

Readings: CP:AMA 19.1, 10.2 – 10.5

The primary goal of this section is to be able to write a module.

Modularization

So far we have been designing programs with all of our definitions in a single source (`.c`) file.

For larger programs, keeping all of the code in one file is unwieldy.

Teamwork on a single file is awkward, and it is difficult to share or re-use code between programs.

A better strategy is to use ***modularization*** to divide programs into well defined **modules**.

The concept of modularization extends far beyond computer science. You can see examples of modularization in construction, automobiles, furniture, nature, etc.

A practical example of a good modular design is an “AA battery”.

We have already seen an elementary type of modularization in the form of *helper functions* that can “help” many other functions.

We will extend and formalize this notion of modularization.

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

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

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

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

Modules are also commonly known as **libraries**.

Modules *vs.* files

In this course, and in much of the "real world", it is considered **good style** to store **modules in separate files**.

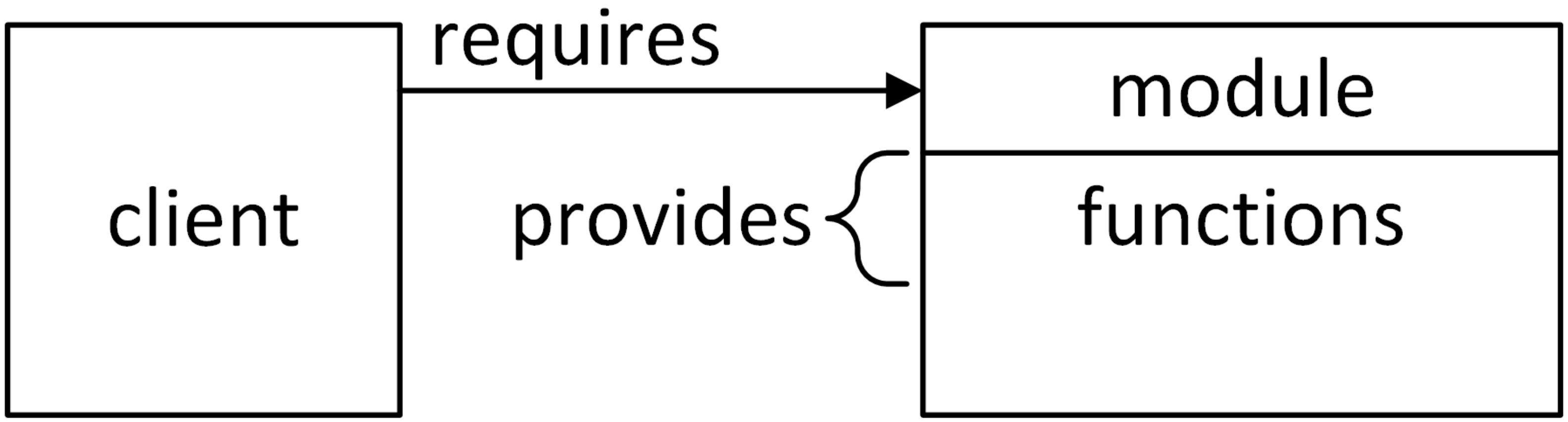
While the terms *file* and *module* are often used interchangeably, a file is only a module if it provides functions for use outside of the file.

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

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

Terminology

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

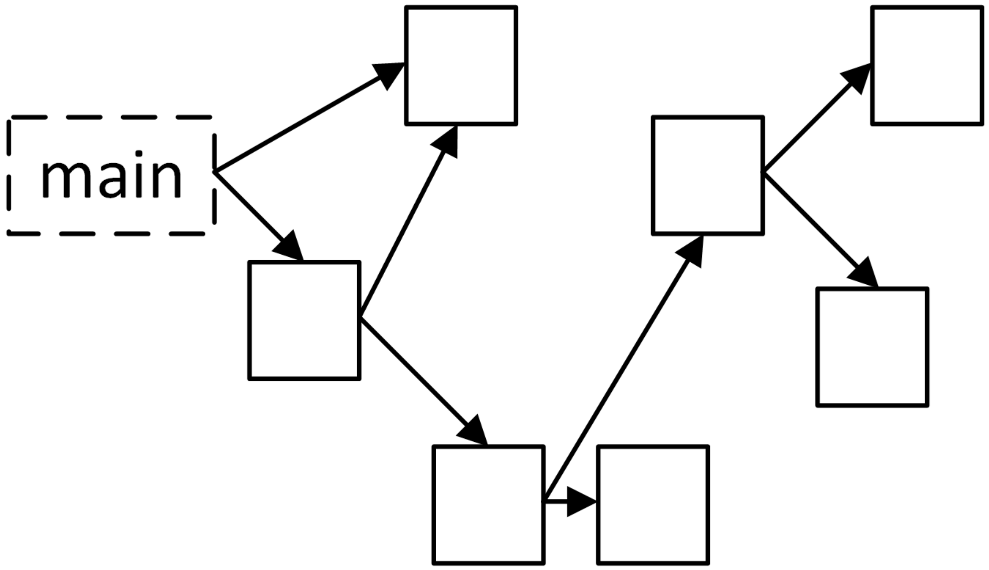


In practice, the client is a file that may be written by yourself, a co-worker or even a stranger.

Conceptually, it is helpful to imagine the client as a stranger.

Large programs can be built from many modules.

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



The *module dependency graph* cannot have any cycles.

There must be a "root" (or ***main file***) that acts only as a client.

This is the program file that defines `main` and is "run".

Building a program

In Section 04 we briefly discussed how we convert ("*compile*") a *source file* (`.c`) into machine code (`.o` or `.ll`) before it can be "run".

When building a program, we can combine (*"link"*) **multiple machine code files** together to form a single program.

In practice, that means we can "build" a program from multiple source code files and/or machine code files.

```
main.c  module1.c  module2.c  module3.ll
```

Modules do not contain a `main` function.

Only **one** `main` function can be defined in a program.

Motivation

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

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

Maintainability: It is much easier to test and debug a single module instead of a large program. If a bug is found, only the module that contains the bug needs to be fixed. We can even replace an entire module with a more efficient or more robust implementation.

Abstraction: To use a module, the client needs to understand **what** functionality it provides, but it does not need to understand **how** it is implemented. In other words, the client only needs an *"abstraction"* of how it works. This allows us to write large programs without having to understand how every piece works.

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

example: fun number module

Imagine that some integers are more “fun” than others, and we want to create a **fun** module that **provides** an `is_fun` function.

```
// fun.c [MODULE]

// is_fun(n) determines if n is fun or not

bool is_fun(int n) {
  return (n == -3   || n == 42   || n == 136 ||
          n == 1337 || n == 4010 || n == 8675309);
}
```

We have to learn a few more concepts before we can complete our module.

Our (yet to be completed) **fun** module illustrates the three key advantages of modularization.

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

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

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

Calling module functions

`is_fun` is defined in the **module** (`fun.c`) file.

How can we call `is_fun` from our **client** (*e.g.,* `main.c`)?

```
// main.c [CLIENT]

int main(void) {
  //...
  b = is_fun(k);  // ERROR
  //...
}
```

```
// fun.c [MODULE]

bool is_fun(int n) {
  //...
}
```

Because C is *statically typed*, this won't work.

C needs to know the return and parameter types of `is_fun`.

Declarations

In C, a function (or any *identifier*) must be ***declared*** before (or *“above”*) it can be referenced.

A ***declaration*** communicates to C the **type** of an identifier.

For example, a **function declaration** includes both the return type and the parameter type(s).

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

Declaration *vs.* definition

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

However, a definition *also* specifies the type of the identifier, so

a definition also includes a declaration.

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

Unfortunately, not all computer languages and reference manuals use these terms consistently.

A **function declaration** is simply the function header followed by a semicolon (`;`) instead of a code block.

example: function declaration

```c
// main.c [CLIENT]

// *** NEW declaration ***
bool is_fun(int n);


int main(void) {
  //...
  b = is_fun(k);   // OK
  //...
}
```

```c
// fun.c [MODULE]

bool is_fun(int n) {
  //...
}
```

By *declaring* `is_fun` at the top of our client (`main.c`), the `is_fun` function is now available in the client.

C ignores the parameter names in a function declaration (it is only interested in the parameter *types*).

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

```
// These are all equivalent:

bool is_fun(int n);
bool is_fun(int);
bool is_fun(int this_name_is_ignored);
```

It is good style to include the correct (and meaningful) parameter names in the declaration to aid communication.

Function ordering

Recall our very first C program. By adding a declaration, we can now place `main` “above” `my_sqr` in the code.

```
int my_sqr(int n);         // DECLARATION

int main(void) {
  trace_int(1 + 1);
  trace_int(my_sqr(7));  // OK
}

int my_sqr(int n) {        // DEFINITION
  return n * n;
}
```

A declaration is necessary to implement **mutual recursion**.

A **variable declaration** starts with the `extern` keyword and is not initialized.

example: variable declaration

```
extern int my_variable;         // variable DECLARATION

int f(int n) {
  return n + my_variable;       // this is now ok
}

int my_variable = 4010;         // variable DEFINITION
```

Variable declarations are uncommon.

Scope

Previously, we have seen two levels of identifier ***scope***:

- **local** identifiers are only visible inside of the function (or *block*) where it is defined
- **global** identifiers are defined at the *top level*, and are visible to all code following the definition

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

We now split global identifiers into two categories.

Global (top-level) identifiers can have either **program *or* module** scope.

- **module** identifiers are only available inside of the module (file) they are defined in
- **program** identifiers are available to any file in the program

We continue to use *global scope* to refer to identifiers that have *either* program or module scope.

By **default**, every C global identifier has **program scope**.

To declare a function or variable with **module** scope, the `static` keyword is used. `static` indicates that the identifier is only "accessible" from within to the current file.

```
int prog_scope_func(...) { ... }        // program scope
static int mod_scope_func(...) {...}    // module scope

int prog_scope_var = 42;                // program scope
static int mod_scope_var = 23;          // module scope
```

It is **good style** to make your module functions and variables `static` if they are not meant to be **provided**.

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

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

Module interface

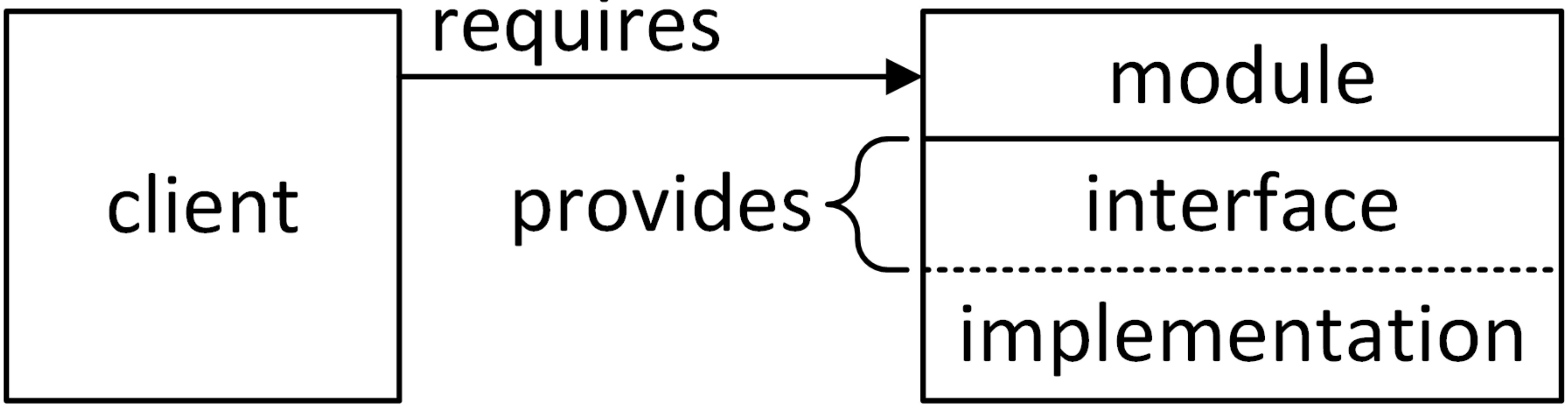
The module ***interface*** is the list of the functions that the module provides (including the documentation).

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

The interface is everything that a client would need to use the module.

The client does not need to see the implementation.

Terminology (revisited)



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

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

Interface contents

The contents of the interface include:

- an **overall description** of the module
- a **function declaration** for each provided function
- **documentation** (*e.g.,* a **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.

Interface (.h) files

For C modules, the **interface** is placed in a separate file with a `.h` file extension.

```
// fun.h [INTERFACE]

// This module is all about having fun

// is_fun(n) determines if n is a fun number
bool is_fun(int n);
```

The interface (`.h`) file has everything the client needs.

Interface files are also known as **h**eader files.

Clients can read the documentation in the interface (`.h`) file to understand how to use the provided functions.

The client can also “copy & paste” the function declarations from the interface file to make the module functions available.

```
// main.c [CLIENT]

// *** copied from .h ***
bool is_fun(int n);


int main(void) {
  //...
  b = is_fun(k);  // OK
  //...
}
```

But there is a much more elegant solution.

#include

A ***preprocessor directive*** **temporarily** “modifies” a source file *just before* it is run (but it does not *save* the modifications).

The *directive* `#include` *“cut & pastes”* or *“inserts”* the contents of another file directly into the current file.

```
  #include "fun.h"      // insert the contents of fun.h
```

For clients, this is perfect: it inserts the interface of the module containing all of the function **declarations**, making all of the provided functions available to the client.

Always put any `#include` *directives* at the top of your file.

example: fun module

```
// fun.h [INTERFACE]

// This module is all
// about having fun

// is_fun(n) determines if
//   n is a fun number
bool is_fun(int n);
```

```
// fun.c [IMPLEMENTATION]

#include "fun.h"

// see fun.h for details
bool is_fun(int n) {
  //...
}
```

```
//////////////////////////////////////////////////////////////
// main.c [CLIENT]

#include "fun.h"

int main(void) {
  //...
  b = is_fun(k);
  //...
}
```

Implementation notes

In the previous example, the fun *implementation* file (`fun.c`) included *its own interface* (`fun.h`).

This is good style as it ensures there are no discrepancies between the interface (declarations) and the implementation (definitions).

The function `is_fun` is fully documented in the *interface* file for the client, so in the *implementation* a simple comment referring the reader to the interface file is sufficient.

```
// see fun.h for details
bool is_fun(int n) {
  //...
}
```

For simple (non-pointer) parameters, `const` is meaningless in a function **declaration**.

The caller (client) does not need to know if the function mutates the **copy** of the argument value.

```
int my_function(int x);                   // DECLARATION
                                          // (no const)

int my_function(const int x) {            // DEFINITION
  // mutation of x here is invalid        // (with const)
  // ...
}
```

It is good style to use `const`ant parameters in **definitions** to improve communication.

In addition to `#include`, 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(int n) {
  return n + MY_NUMBER;
}
```

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

cs136.h

Since the beginning of this course, we have seen

```
#include "cs136.h"
```

in our programs. We now understand that we have been using support tools and functions (*e.g.*, `trace_int` and `read_int`) provided by a cs136 module.

The `cs136.h` interface file contains:

```
#include <assert.h>
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
```

So our cs136 module was requiring additional modules.

C standard modules

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

Fortunately, C provides several ***standard modules*** (also known as *libraries*) with many useful functions.

For example, the `stdio` module provides `printf` and `scanf`.

When using `#include` we use angle brackets (<>) to specify that the module is one of the *standard modules* and quotes (`""`) for “regular” modules (*i.e.*, ones we have written).

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

Here are some of the other modules that we have been using:

- `<assert.h>` provides the function `assert`
- `<limits.h>` provides the constants `INT_MAX` and `INT_MIN`
- `<stdbool.h>` provides the `bool` data type and the constants `true` and `false`
- `<stdlib.h>` provides the constant `NULL`

example: test client

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

```
// test-fun.c: testing client for the fun module

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

int main(void) {
  assert(is_fun(42));
  assert(!is_fun(13));
  //...
}
```

Do NOT add a `main` function to your implementation (*e.g.*,`fun.c`). Create a new test *client* with a `main` function.

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

In these circumstances, you can provide a `test_module_name` function that asserts your tests are successful.

Designing modules

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

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

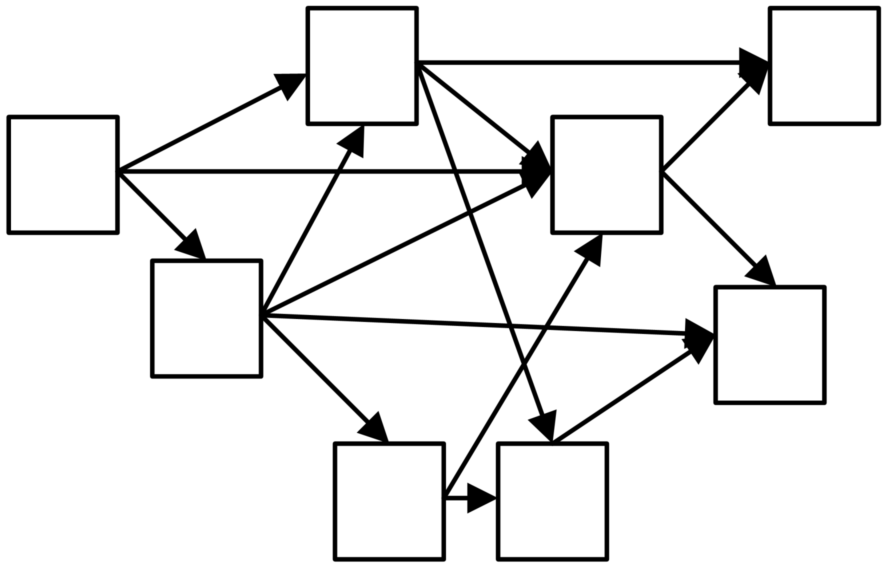
For now, we will have a brief discussion on what constitutes a **good** interface design.

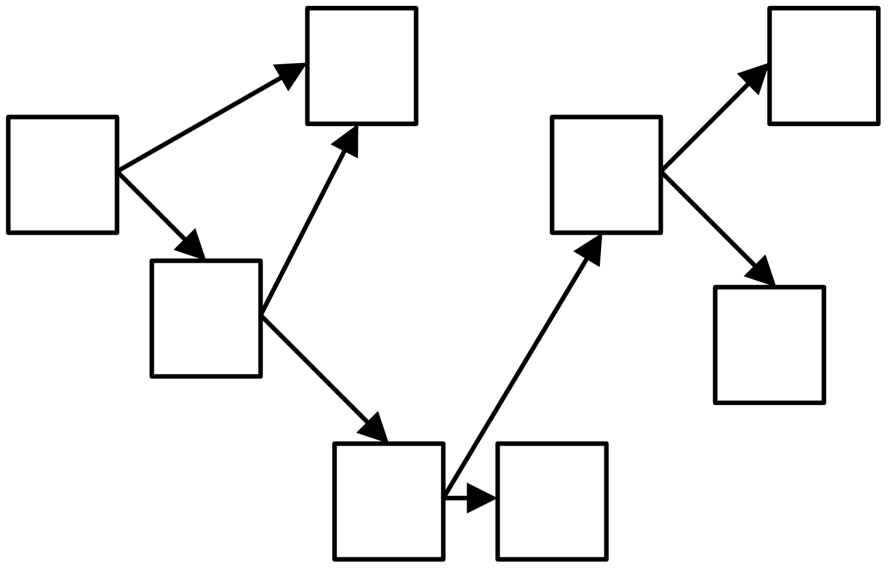
Cohesion and coupling

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

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

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


High coupling


Low coupling

Interface *vs.* implementation

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

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

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.

Information hiding in C

With C modules, it is easy to hide the implementation details from the client.

Instead of providing the client with the implementation source code (`.c` file) you can provide a machine code (*e.g.,* a `.ll` file). This is what we did with our `cs136` module.

While C is good at hiding the implementation code, it is not very good at hiding **data**.

If a **malicious** client obtains (or “guesses”) the memory address of some data they can access it directly.

Opaque structures in C

Fortunately, C supports **opaque structures**, which are “good enough” to hide data from a **friendly** client.

An *opaque structure* is like a “black box” that the client cannot “see” inside of.

They are implemented in C using ***incomplete declarations***, where a structure is *declared* without any fields.

```
  struct box;                     // INCOMPLETE DECLARATION
```

With an *incomplete declarations*, **only pointers to the structure can be defined**.

```
  struct box my_box;              // INVALID
  struct box *box_ptr;            // VALID
```

If a module only provides an *incomplete declaration* in the **interface**, the client can not create an instance of the `struct` or access any of the fields.

The module must provide a function to *create* (and *destroy*) an instance of the structure. This is explored more in Section 10.

Of course, if we want a **transparent** structure the client can use, we simply put the **complete definition** of the `struct` **in the interface** file (`.h` file).

example: stopwatch module

To illustrate the principles of information hiding, we will create a small `stopwatch` module.

For narrative purposes, we can imagine this module is for keeping track of race times and we do not want clients to be able to tamper with the times (this is an example of *security*).

In the **interface**, we have an incomplete declaration.

The client cannot define a `struct stopwatch`, so we need to provide *create* and *destroy* functions.

```
// stopwatch.h [INTERFACE]

struct stopwatch;

// stopwatch_create() creates a new stopwatch at time 0:00
// effects: allocates memory (client must call stopwatch_destroy)
struct stopwatch *stopwatch_create(void);

// stopwatch_destroy(sw) removes memory for sw
// effects: sw is no longer valid
void stopwatch_destroy(struct stopwatch *sw);
```

We learn how to write create/destroy functions in Section 10.

The rest of the interface provides some simple stopwatch functions.

```
// stopwatch.h [INTERFACE]

// stopwatch_get_seconds(sw) returns the number of seconds in sw
int stopwatch_get_seconds(const struct stopwatch *sw);

// stopwatch_get_minutes(sw) returns the number of minutes in sw
int stopwatch_get_minutes(const struct stopwatch *sw);

// stopwatch_add_time(sw, min, sec) adds min[utes] and
//   sec[onds] to sw
// requires: 0 <= minutes, seconds
// effects: modifies sw
void stopwatch_add_time(struct stopwatch *sw, int min, int sec);
```

The **implementation** fully defines a `struct stopwatch` and simple functions to access the fields of the structure.

```
// stopwatch.c [IMPLEMENTATION]

struct stopwatch {
  int min;
  int sec;
};
// requires: 0 <= min
//           0 <= sec <= 59

int stopwatch_get_seconds(const struct stopwatch *sw) {
  assert(sw);
  return sw->sec;
}

int stopwatch_get_minutes(const struct stopwatch *sw) {
  assert(sw);
  return sw->min;
}
```

The only non-trivial function is the stopwatch_add_time function.

```
// stopwatch.c [IMPLEMENTATION]

void stopwatch_add_time(struct stopwatch *sw, int min, int sec) {
  assert(sw);
  assert(sec >= 0);
  assert(min >= 0);
  sw->min += min;
  sw->sec += sec;
  while (sw->sec >= 60) {
    sw->min += 1;
    sw->sec -= 60;
  }
}
```

This simple client illustrates the use of the `stopwatch` module.

```
// client.c

#include "cs136.h"
#include "stopwatch.h"

int main(void) {
  struct stopwatch *sw = stopwatch_create();

  stopwatch_add_time(sw, 1, 59);
  stopwatch_add_time(sw, 3, 30);

  trace_int(stopwatch_get_minutes(sw));
  trace_int(stopwatch_get_seconds(sw));

  stopwatch_destroy(sw);
}
```

```
stopwatch_get_minutes(sw) => 5
stopwatch_get_seconds(sw) => 29
```

Maintainability

We can improve our `stopwatch` module without changing the client. This is an example of *maintainability*.

```
// stopwatch.c [IMPLEMENTATION]

static const int seconds_per_minute = 60;

void stopwatch_add_time(struct stopwatch *sw, int min, int sec) {
  assert(sw);
  assert(sec >= 0);
  assert(min >= 0);
  const int total_time = (min + sw->min) * seconds_per_minute +
                         sec + sw->sec;
  sw->min = total_time / seconds_per_minute;
  sw->sec = total_time % seconds_per_minute;
}
```

Flexibility

Because we have used **information hiding** to design our **interface**, our implementation is not only *maintainable*, but also **flexible**.

We can change the design of our `struct stopwatch` and the client will be completely unaware.

The client only accesses our structure through the provided functions. As long as we do not change the function behaviour, we have the *flexibility* to change our design.

For example, we can change our `struct stopwatch` to contain a single field (`seconds`) instead of storing the information in two fields (`sec` and `min`).

```
// stopwatch.c [IMPLEMENTATION] (asserts removed for conciseness)

struct stopwatch {
  int seconds;
};
// requires: 0 <= seconds

static const int seconds_per_minute = 60;

int stopwatch_get_seconds(const struct stopwatch *sw) {
  return sw->seconds % seconds_per_minute;
}

int stopwatch_get_minutes(const struct stopwatch *sw) {
  return sw->seconds / seconds_per_minute;
}

void stopwatch_add_time(struct stopwatch *sw, int min, int sec) {
  sw->seconds += min * seconds_per_minute + sec;
}
```

Data structures & abstract data types

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

- a `struct` with two fields (`sec` and `min`)
- a `struct` with one field (`seconds`)

For each *data structure* we knew how the data was “structured”.

However, the client doesn’t need to know how the data is structured.

The client only requires an **abstract** understanding that a `stopwatch` stores time information.

The `stopwatch` module is an implementation of a stopwatch **Abstract Data Type (ADT)**.

Formally, an 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*).

Data structures *vs.* ADTs

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

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

With an **ADT**, the client does not know how the data is structured and can only access the data through the interface functions (operations) provided by the ADT.

The terminology is especially confusing because ADTs are **implemented** with a data structure.

Collection ADTs

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

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

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

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

By some definitions, collection ADTs are the *only* type of ADT.

Dictionary (revisited)

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

Typical dictionary ADT operations:

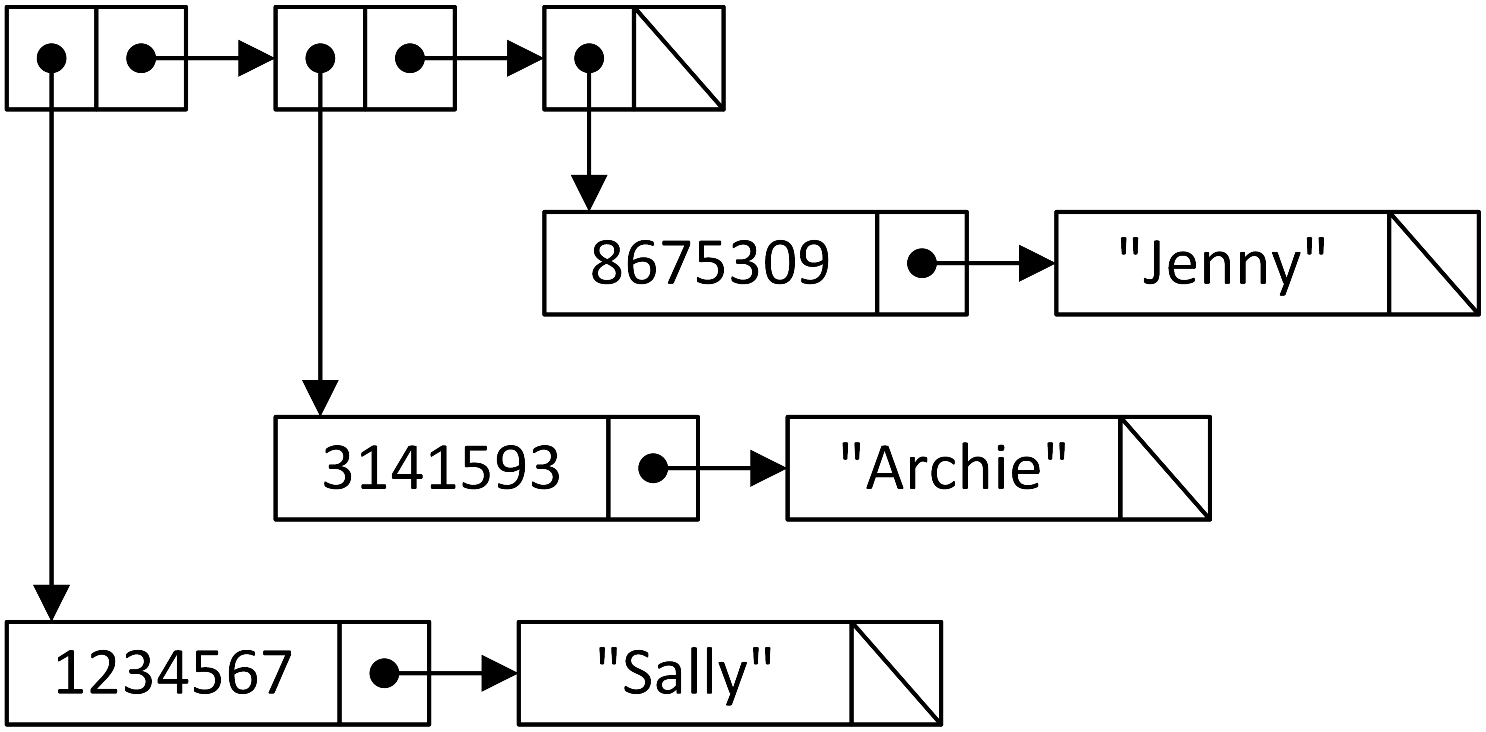
- **lookup:** for a given key, retrieve the corresponding value or “not found”
- **insert:** adds a new key/value pair (or replaces the value of an existing key)
- **remove:** *deletes* a key and its value

example: student numbers

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

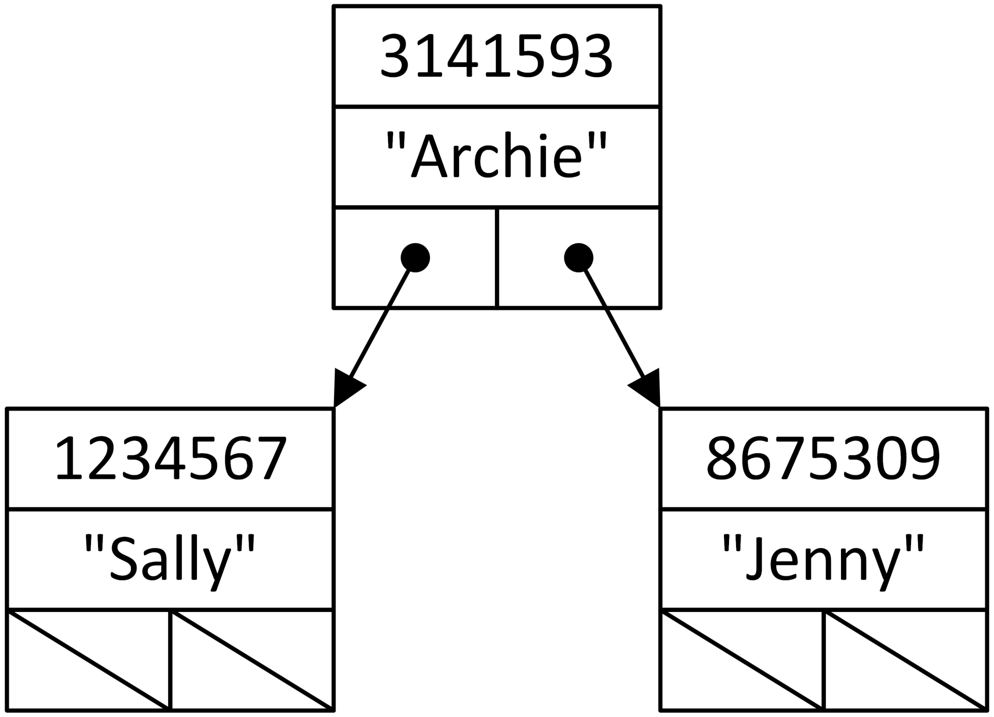
In CS 135 we implemented a dictionary with an ***association list data structure*** (a list of key/value pairs with each pair stored as a two-element list).

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



We also implemented 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)))
```



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.

Regardless, the **client** would never know which implementation was being used.

You likely have an intuition that BSTs are “more efficient” than association lists. In Section 08 we introduce a formal notation to describe the efficiency of an implementation.

More collection ADTs

Three additional collection ADTs that are explored in this course:

- stack
- queue
- sequence

Stack ADT

We have already been exposed to the idea of a stack when we learned about the **call stack**.

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

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

Typical stack ADT operations:

- **push:** adds an item to the top stack
- **pop:** removes the top item from the stack
- **top:** returns the top item on the stack
- **is_empty:** determines if the stack is empty

Queue ADT

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

Typical queue ADT operations:

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

Sequence ADT

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

Typical sequence ADT operations:

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

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

Goals of this Section

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

- explain and demonstrate the three core advantages of modular design: abstraction, re-usability and maintainability
- identify two characteristics of a good modular interface: high cohesion and low coupling
- explain and demonstrate information hiding and how it supports both security and flexibility
- explain what a modular interface is, the difference between an interface and an implementation, and the importance of a good interface design.

- explain the 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
- write good interface documentation
- use a module as a client (including writing a test client)

Arrays

Readings: CP:AMA 8.1, 9.3, 12.1, 12.2, 12.3

The primary goal of this section is to be able to use arrays.

Arrays

C only has two *built-in* types of “compound” data storage:

- `struct`ures
- ***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]` and the last is `my_array[5]`.

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

example: accessing array elements

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

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

int j = a[0];             // j is 4
int *p = &a[j - 1];       // p points at a[3]

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

example: arrays & iteration

Arrays and iteration are a powerful combination.

```
int a[6] = {4, 8, 15, 16, 23, 42};
int sum = 0;

for (int i = 0; i < 6; ++i) {
  printf("a[%d] = %d\n", i, a[i]);
  sum += a[i];
}
printf("sum = %d\n", sum);
```

```
a[0] = 4
a[1] = 8
a[2] = 15
a[3] = 16
a[4] = 23
a[5] = 42
sum = 108
```

Array initialization

Arrays can only be **initialized** with braces (`{}`).

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

a = {0, 0, 0, 0, 0, 0};     // INVALID
a = ??? ;                   // INVALID
```

Once defined, the entire array cannot be mutated at once. Only *individual elements* can be mutated.

If there are not enough elements in the initialization braces, the remaining values are initialized to zero.

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

Character arrays can be initialized with double quotes (") for convenience.

The following two definitions are equivalent:

```
char a[3] = {'c', 'a', 't'};
char b[3] = "cat";
```

In this example, a and b are character arrays and are not valid strings. This will be revisited in Section 09.

Like variables, the value of an uninitialized array depends on the scope of the array:

```
int a[5];  // uninitialized
```

- uninitialized *global* arrays are zero-filled.
- uninitialized *local* arrays are filled with arbitrary (“garbage”) values from the stack.

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.

To improve readability, the array length is often stored in a separate variable.

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

It might seem better to use a constant to specify the length of an array.

```
const int a_len = 6;
int a[a_len] = {4, 8, 15, 16, 23, 42};  // NOT IN CS136
```

This would appear to be a “better style”.

However, the syntax to do this properly is outside of the scope of this course (see following slide).

In this course, always define arrays using numbers. It is okay to have these “magic numbers” appear in your assignments.

```
int a[6] = ...;
```

A preferred syntax to specify the length of an array is to define a *macro*.

```
#define A_LEN 6

int main(void) {
  int a[A_LEN] = {4, 8, 15, 16, 23, 42};
  // ...
```

In this example, `A_LEN` is not a constant or even a variable.

`A_LEN` is a *preprocessor macro*. Every occurrence of `A_LEN` in the code is replaced with 6 before the program is run.

C99 supports *Variable Length Arrays* (VLAs), where the length of an **uninitialized** local array can be specified by a variable (or a function parameter) not known in advance. The size of the stack frame is increased accordingly.

```
int some_function(int n) {
  int m = n * 2;
  int a[m];     // length determined at run time
  // ...
```

This approach has many disadvantages and in more recent versions of C, this feature was removed (made optional). **You are not allowed to use VLAs in this course**. In Section 10 we see a better approach.

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 (and in this course) this should be avoided, as the `sizeof` operator only properly reports the array size in very specific circumstances.

Array size

The **length** of an array is the number of elements in the array.

The **size** of an array is the number of bytes it occupies in memory.

An array of k elements, each of size s, requires exactly $k \times s$ bytes.

In the C memory model, array elements are adjacent to each other. Each element of an array is placed in memory immediately after the previous element.

If `a` is an integer array with six elements (`int a[6]`) the size of `a` is: $(6 \times$ `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

The array identifier

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

```
int a[6] = {4, 8, 15, 16, 23, 42};
trace_ptr(a);
trace_ptr(&a);
trace_ptr(&a[0]);
```

```
a => 0x5000
&a => 0x5000
&a[0] => 0x5000
```

Even though a and &a have the same *value*, they have different *types*, and cannot be used interchangeably.

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

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

trace_int(a[0]);
trace_int(*a);
```

```
a[0] => 4
*a => 4
```

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 to the function, and is a separate parameter.

There is no method of “enforcing” that the length passed to a function is valid.

Functions should **require** that the length is valid, but there is no way for a function to `assert` that requirement.

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};
  trace_int(sum_array(my_array, 6));
}
```

```
sum_array(my_array, 6) => 108
```

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

It's good style to use the `const` keyword to both 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:

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

  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)`.

C does not perform any array “bounds checking”.

For a given array `a` of length l, C does not verify that `a[j]` is valid ($0 \leq j < l$).

C simply “translates” `a[j]` to `*(a + j)`, which may be outside the *bounds* of the array (*e.g.*, `a[1000000]` or `a[-1]`).

This is a common source of errors and bugs and a common criticism of C. Many modern languages have fixed this shortcoming and have “bounds checking” on arrays.

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 declarations:

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

In *most* contexts, there is no practical difference between an array identifier and an immutable pointer.

The subtle differences between an array and a pointer are discussed at the end of Section 09.

example: “pretty” print an array

```
// pretty prints an array with commas, ending with a period
// requires: len > 0

void print_array(int a[], int len) {
  assert(len > 0);
  for (int i = 0; i < len; ++i) {
    if (i) {
      printf(", ");
    }
    printf("%d", a[i]);
  }
  printf(".\n");
}

int main(void) {
  int a[6] = {4, 8, 15, 16, 23, 42};
  print_array(a, 6);
}
```

```
4, 8, 15, 16, 23, 42.
```

Array map

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

```
// array_map(f, a, len) replaces each element a[i]
//   with f(a[i])
// effects: modifies a

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

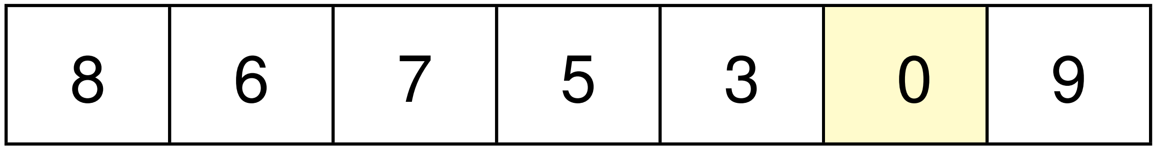
```c
#include "array_map.h"

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

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

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

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

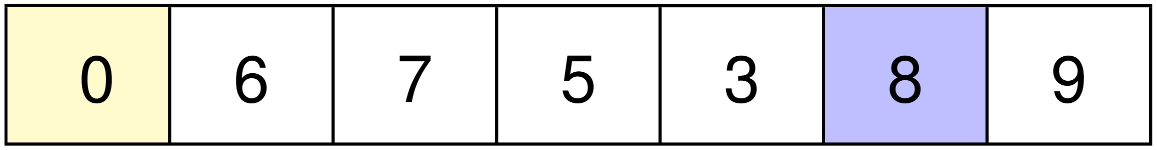
Selection sort

In ***selection sort***, the smallest element is *selected* to be the first element in the new sorted sequence, and then the next smallest element is selected to be the second element, and so on.

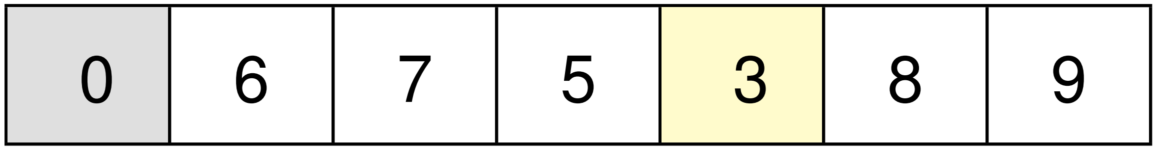
First, we find the position of the smallest element...



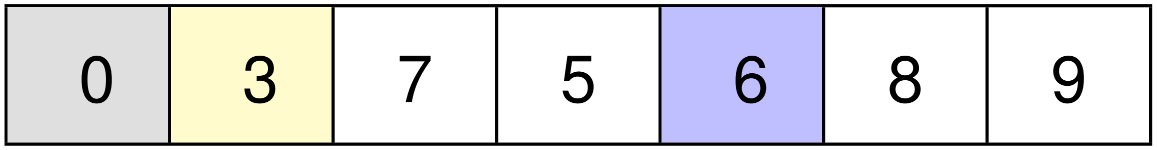
and then we *swap* the first element with the smallest.



Then, we find the next smallest element...



and then we *swap* that element with the second one, and so forth...



```
void selection_sort(int a[], int len) {
  int pos = 0;
  for (int i = 0; i < len - 1; ++i) {
    pos = i;
    for (int j = i + 1; j < len; ++j) {
      if (a[j] < a[pos]) {
        pos = j;
      }
    }
    swap(&a[i], &a[pos]);  // see Section 05
  }
}
// Notes:
//   i:   loops from 0 ... len-2 and represents the
//        "next" element to be replaced
//   j:   loops from i+1 ... len-1 and is "searching"
//        for the next smallest element
//   pos: position of the "next smallest"
```

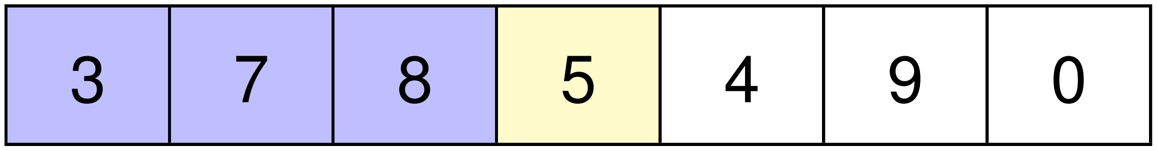
Insertion sort

In ***Insertion sort***, we consider the first element to be a sorted sequence (of length one).

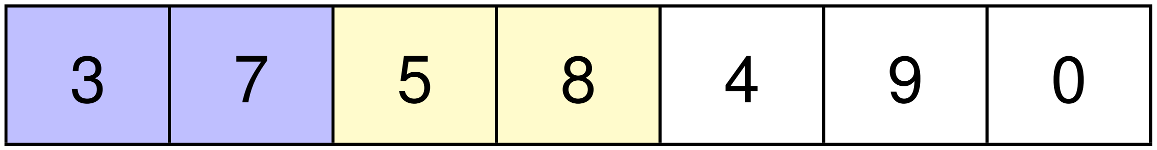
We then “insert” the second element into the existing sequence into the correct position, and then the third element, and so on.

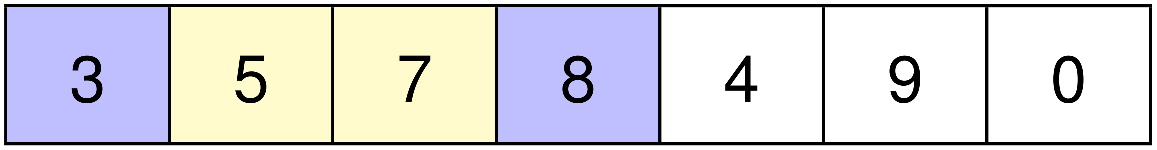
For each iteration of *Insertion sort*, the first `i` elements are sorted. We then “insert” the element `a[i]` into the correct position, moving all of the elements greater than `a[i]` one to the right to “make room” for `a[i]`.

Consider an iteration of insertion sort (`i = 3`), where the first `i` (3) elements have been sorted. We want to *insert* the element at `a[i]` into the correct position.

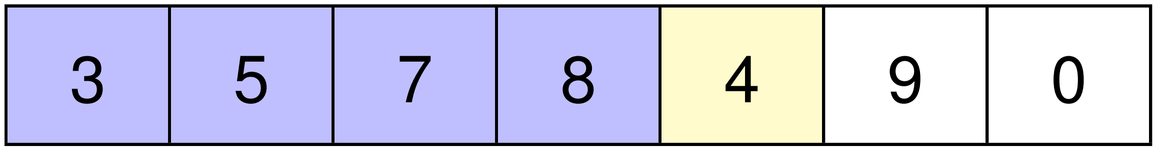


We continue to *swap* the element with the previous element until it reaches the correct position.





Once it is in the correct position, we start on the next element.



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

// Notes:
//   i:   loops from 1 ... len-1 and represents the
//        "next" element to be replaced
//   j:   loops from i ... 1 and is "inserting"
//        the element that was at a[i] until it
//        reaches the correct position
```

Quicksort

Quicksort is an example of a “divide & conquer“ algorithm.

First, an element is selected as a “pivot” element.

The list is then **partitioned** (*divided*) into two sub-groups: elements *less than* (or equal to) the pivot and those *greater than* the pivot.

Finally, each sub-group is then sorted (*conquered*).

Quicksort is also known as partition-exchange sort or Hoare’s quicksort (named after the author).

We have already seen the implementation of quick sort in racket.

```
(define (quick-sort lon)
  (cond [(empty? lon) empty]
    [else (define pivot (first lon))
          (define less (filter (lambda (x)
                               (<= x pivot)) (rest lon)))
          (define greater (filter (lambda (x)
                                  (> x pivot)) (rest lon)))
          (append (quick-sort less)
                  (list pivot)
                  (quick-sort greater))]))
```

For simplicity, we select the first element as the “pivot”. A more in-depth discussion of pivot selection occurs in CS 240.

In our C implementation of quick sort, we:

- select the first element of the array as our "pivot"
- move all elements that are larger than the pivot to the back of the array
- move ("swap") the pivot into the correct position
- recursively sort the "smaller than" sub-array and the "larger than" sub-array

The core quick sort function `quick_sort_range` has parameters for the range of elements (`first` and `last`) to be sorted, so a wrapper function is required.

```
void quick_sort_range(int a[], int first, int last) {

  if (last <= first) return;  // length is <= 1

  int pivot = a[first];       // first element is the pivot
  int pos = last;             // where to put next larger

  for (int i = last; i > first; --i) {
    if (a[i] > pivot) {
      swap(&a[pos], &a[i]);
      --pos;
    }
  }
  swap(&a[first], &a[pos]);   // put pivot in correct place
  quick_sort_range(a, first, pos - 1);
  quick_sort_range(a, pos + 1, last);
}

void quick_sort(int a[], int len) {
  quick_sort_range(a, 0, len - 1);
}
```

Binary search

In Racket, the built-in function `member` can be used to determine if a list contains an element.

We can write a similar function in C that finds the index of an element in an array:

```
// find(item, a, len) finds the index of item in a,
//   or returns -1 if it does not exist

int find(int item, const int a[], int len) {
  for (int i = 0; i < len; ++i) {
    if (a[i] == item) {
      return i;
    }
  }
  return -1;
}
```

But what if the array was previously *sorted*?

We can use **binary search** to find the element faster:

```
int find_sorted(int item, const int a[], int len) {
  int low = 0;
  int mid = 0;
  int high = len - 1;
  while (low <= high) {
    mid = (low + high) / 2;
    if (a[mid] == item) {
      return mid;
    } else if (a[mid] < item) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return -1;
}
```

Multi-dimensional data

All of the arrays seen so far have been one-dimensional (1D) arrays.

We can represent multi-dimensional data by “mapping” the higher dimensions down to one.

For example, consider a 2D array with 2 rows and 3 columns.

```
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.

Fixed-Length Arrays

A significant limitation of an array is that you need to know the length of the array **in advance**.

In Section 10 we introduce *dynamic memory* which can be used to circumvent this limitation, but first we explore a less sophisticated approach.

In some applications, it may be “appropriate” (or “easier”) to have a **maximum length** for an array.

In general, maximums should only be used when appropriate:

- They are wasteful if the maximum is excessively large.
- They are restrictive if the maximum is too small.

When working with maximum-length arrays, we need to keep track of

- the **“actual” length** of the array, and
- the **maximum possible length**.

To illustrate fixed-length arrays, we implement an integer **stack** structure with a maximum length of 100 elements.

The `len` field keeps track of the *actual* length of the stack.

```
struct stack {
  int len;
  int maxlen;
  int data[100];
};
```

We need to provide a `stack_init` function to initialize the structure:

```
void stack_init(struct stack *s) {
  assert(s);
  s->len = 0;
  s->maxlen = 100;
}
```

Ignoring the push operation for now, we can write the rest of the stack implementation:

```
bool stack_is_empty(const struct stack *s) {
  assert(s);
  return s->len == 0;
}

int stack_top(const struct stack *s) {
  assert(s);
  assert(s->len > 0);
  return s->data[s->len - 1];
}

// note: stack_pop returns the element popped
int stack_pop(struct stack *s) {
  assert(s);
  assert(s->len > 0);
  s->len -= 1;
  return s->data[s->len];
}
```

What happens if we exceed the maximum length when we try to `push` an element?

There are a few possibilities:

- the stack is not modified and an error message is displayed
- a special return value can be used
- an `assert`ion fails (terminating the program)
- the program explicitly **terminates** with an error message

Any approach may be appropriate as long as the contract properly documents the behaviour.

The `exit` function (part of <`stdlib.h`>) stops program execution. It is useful for “fatal” errors.

The argument passed to `exit` is equivalent to the `return` value of `main`.

For convenience, <`stdlib.h`> defines `EXIT_SUCCESS` which is `0` and `EXIT_FAILURE` which is non-zero.

```
if (something_bad) {
  printf("FATAL ERROR: Something bad happened!\n");
  exit(EXIT_FAILURE);
}
```

```
// stack_push(item, s) pushes item onto stack s
// requires: s is a valid stack
// effects:  modifies s
//           may display output and exit

void stack_push(int item, struct stack *s) {
  assert(s);
  if (s->len == s->maxlen) {
    printf("FATAL ERROR: stack capacity (%d) exceeded\n",
           s->maxlen);
    exit(EXIT_FAILURE);
  }
  s->data[s->len] = item;
  s->len += 1;
}
```

Goals of this Section

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

- define and initialize arrays
- use iteration to loop through arrays
- use pointer arithmetic
- explain how arrays are represented in the memory model, and how the array index operator (`[]`) uses pointer arithmetic to access array elements in constant time
- use both array index notation (`[]`) and array pointer notation and convert between the two

- use fixed-length arrays
- describe selection sort, insertion sort, quicksort and binary search on a sorted array
- represent multi-dimensional data in a single-dimensional array

Efficiency

Readings: None

The primary goal of this section is to be able to analyze the efficiency of an algorithm.

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.).

There are many objective and subjective methods for comparing algorithms:

- How easy is it to understand?
- How easy is it to implement?
- How accurate is it?
- How robust is it? (Can it handle errors well?)
- How adaptable is it? (Can it be used to solve similar problems?)
- **How fast (efficient) is it?**

In this course, we use ***efficiency*** to objectively compare algorithms.

Efficiency

The most common measure of efficiency is ***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 measurement? Is the 0.037 relevant?

Measuring *running times* in seconds can be problematic.

What are the alternatives?

Typically, we measure the number of **elementary operations** required to solve the problem.

- In C, we can count the number of operations, or in other words, the number of *operators* executed.
- In Racket, we can count the total number of (substitution) **steps** required, although that can be deceiving for built-in functions[†].

[†] We revisit the issue of built-in functions later.

You are not expected to count the exact number of operations.

We only count operations in these notes for illustrative purposes.

We introduce some simplification shortcuts soon.

Data size

What is the number of operations executed for this implementation?

```
int sum_array(const int a[], int len) {
  int sum = 0;
  int i = 0;
  while (i < len) {
    sum = sum + a[i];
    i = i + 1;
  }
  return sum;
}
```

The running time **depends on the length** of the array.

If there are n items in the array, it requires $7n + 3$ operations.

We are always interested in the running time *with respect to* the **size of the data**.

Traditionally, the variable n is used to represent the **size** (or **length**) of the data. m and k are also popular when there is more than one parameter.

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 data that is also important.

For example, with *trees*, we use n to represent the number of nodes in the tree and h to represent the *height* of the tree.

In advanced algorithm analysis, n may represent the number of *bits* required to represent the data, or the length of the *string* necessary to describe the data.

Algorithm Comparison

Problem: Write a function to determine if an array of positive integers contains at least `e` even numbers and `o` odd numbers.

```
// check_array(a, len, e, o) determines if array a
//   contains at least e even numbers and
//   at least o odd numbers
// requires: len > 0
//           elements of a > 0
//           e, o >= 0
```

Homer and Bart are debating the best algorithm (strategy) for implementing `check_array`.

Bart just wants to count the total number of odd numbers in the entire array.

```
bool bart(const int a[], int len, int e, int o) {
  int odd_count = 0;
  for (int i = 0; i < len; i = i + 1) {
    odd_count = odd_count + (a[i] % 2);
  }
  return (odd_count >= o) && (len - odd_count >= e);
}
```

If there are n elements in the array, $T(n) = 8n + 7$.

Remember, you are not expected to calculate this precisely.

Homer is lazy, and he doesn’t want to check all of the elements in the array if he doesn’t have to.

```
bool homer(const int a[], int len, int e, int o) {
  // only loop while it's still possible
  while (len > 0 && e + o <= len) {
    if (a[len - 1] % 2 == 0) {  // even case:
      if (e > 0) {
        e = e - 1;                // only decrement e if e > 0
      }
    } else if (o > 0) {
      o = o - 1;
    }
    if (e == 0 && o == 0) {
      return true;
    }
    len = len - 1;
  }
  return false;
}
```

The problem with analyzing Homer's code is that it depends not just on the length of the array, but on the contents of the array and the parameters e and o.

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// these are fast:
bool fast1 = homer(a, 10, 0, 11);    // false;
bool fast2 = homer(a, 10, 1, 0);     // true;

// these are slower:
bool slow1 = homer(a, 10, 5, 5);     // true;
bool slow2 = homer(a, 10, 6, 4);     // false;
```

For Homer's code, the ***best case*** is when it can `return` immediately, and the ***worst case*** is when *all* of the array elements are visited.

For Bart's code, the best case is the same as the worst case.

`homer`	$T(n) = 4$	(best case)
	$T(n) = 17n + 1$	(worst case)
`bart`	$T(n) = 8n + 7$	(all cases)

Which implementation is more efficient?

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

Worst case running time

Typically, we want to be conservative (*pessimistic*) and use the *worst case*.

Unless otherwise specified, the running time of an algorithm is the **worst case running time**.

Comparing the worst case, Bart's implementation ($8n + 7$) is more efficient than Homer's ($17n + 1$).

We may also be interested in the *average* case running time, but that analysis is typically much more complicated.

Big O notation

In practice, we are not concerned with the difference between the running times $(8n + 7)$ and $(17n + 1)$.

We are interested in the ***order*** of a running time. The order is the **"dominant" term** in the running time **without any constant coefficients**.

The dominant term in both $(8n + 7)$ and $(17n + 1)$ 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

- $2016 = O(1)$
- $100000 + n = O(n)$
- $n + n \log n = O(n \log n)$
- $999n + 0.01n^2 = O(n^2)$
- $\frac{n(n+1)(2n+1)}{6} = O(n^3)$
- $n^3 + 2^n = O(2^n)$

When comparing algorithms, the most efficient algorithm is the one with the lowest *order*.

For example, an $O(n \log n)$ algorithm is more efficient than an $O(n^2)$ algorithm.

If two algorithms have the same *order*, they are considered **equivalent**.

Both Homer's and Bart's implementations are $O(n)$, so they are equivalent.

Big O arithmetic

When *adding* two orders, the result is the largest of the two orders.

- $O(\log n) + O(n) = O(n)$
- $O(1) + O(1) = O(1)$

When *multiplying* two orders, the result is the product of the two orders.

- $O(\log n) \times O(n) = O(n \log n)$
- $O(1) \times O(n) = O(n)$

There is no “universally accepted” Big O notation.

In many textbooks, **and in this introductory course**, the notation

$$T(n) = 1 + 2n + 3n^2 = O(1) + O(n) + O(n^2) = O(n^2)$$

is acceptable.

In other textbooks, and in other courses, this notation may be too informal.

In CS 240 and CS 341 you will study orders and Big O notation much more rigourously.

Algorithm analysis

An important skill in Computer Science is the ability to ***analyze*** a function and determine the *order* of the running time.

In this course, our goal is to give you experience and work toward building your intuition:

```
int sum_array(const int a[], int len) {
  int sum = 0;
  for (int i = 0; i < len; ++i) {
    sum += a[i];
  }
  return sum;
}
```

"Clearly, each element is visited once, so the running time of `sum_array` is $O(n)$".

Contract update

You should include the `time` (efficiency) of each function that is not $O(1)$ and is not *obviously* $O(1)$.

If there is any ambiguity as to how n is measured, it should be specified.

```
// sum_array(const int a[], int len) sums the elements
//   of array a
// time: O(n), n is the len of a
```

Analyzing simple functions

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

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

If no other functions are called, there must be a fixed number of operators. Each operator is $O(1)$, so the running time is:

$$O(1) + O(1) + \overset{\text{[fixed \# of times]}}{\cdots} + O(1) = O(1)$$

If a simple function calls other functions, its running time depends on those functions.

Built-in functions

Consider the following two implementations.

```
(define (a-length-two? lst)
  (= 2 (length lst)))
```

```
(define (b-length-two? lst)
  (and (cons? lst) (cons? (rest lst))
       (empty? (rest (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 (numeric)

When working with *small* integers (*i.e.,* valid C integers), the Racket numeric functions are $O(1)$.

However, because Racket can handle arbitrarily large numbers it is more complicated.

For example, the running time to add two *large* positive integers is $O(\log n)$, where n is the largest number.

Racket running times (lists)

Elementary list functions are $O(1)$:

`cons cons? empty empty? rest first second tenth`

List functions that process the full list are typically $O(n)$:

`length last reverse append`

Abstract list functions (*e.g.*, `map`, `filter`) depend on the consumed function, but are $O(n)$ for straightforward $O(1)$ functions.

The exception is Racket's `sort`, which is $O(n \log n)$.

Racket running times (equality)

We can assume `=` (numeric equality) is $O(1)$.

`symbol=?` is $O(1)$, but

`string=?` is $O(n)$, where n is the length of the smallest string[†].

Racket's generic `equal?` is deceiving: its running time is $O(n)$, where n is the "size" of the smallest argument.

Because `(member e lst)` depends on `equal?`, its running time is $O(nm)$ where n is the length of the `lst` and m is the size of `e`.

[†] This highlights another difference between symbols & strings.

Array efficiency

One of the significant differences between arrays and lists is that any element of an array can be accessed in constant time regardless of the index or the length of the array.

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

To access the i-th element in a **list** (*e.g.*, `list-ref`) is $O(i)$.

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.

Iterative analysis

Iterative analysis uses **summations**.

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

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

Because we are primarily interested in *orders*,

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

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

Procedure for iteration

1. Work from the *innermost* loop to the *outermost*
2. Determine the number of iterations in the loop (in the worst case) in relation to the size of the data (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 data* and does not necessarily reflect the actual loop counter values.

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

There are $n/10$ iterations. Because we are only interested in the *order*, $n/10$ and n are equivalent.

$$\sum_{i=1}^{n/10} O(1) = O(n)$$

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

```
k = n;                  // n is size of the data
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 (i = 0; i < n; ++i) {
  for (j = 0; j < i; ++j) {
    printf("*");
  }
  printf("\n");
}
```

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

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

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 data 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

```
int sum_first(int n) {
  if (n == 0) return 0;
  return n + sum_first(n - 1);
}
```

1. All non-recursive operations: $O(1)$ `+`, `-`, `==`
2. size of the recursion: $n - 1$
3. $T(n) = O(1) + T(n - 1)$ (combine 1 & 2)
4. $T(n) = O(n)$ (table lookup)

Revisiting sorting algorithms

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

For simplicity, we only consider sorting **numbers**.

When sorting strings or large data structures, you must also include the time to compare each element.

When analyzing sorting algorithms, one measure of running time is the number of comparisons.

Selection sort

Recall our C implementation of selection sort:

```
void selection_sort(int a[], int len) {
  int pos = 0;
  for (int i = 0; i < len - 1; ++i) {
    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
  }
}
```

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

Insertion sort

The analysis for the worst case of insertion sort is also $O(n^2)$.

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

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

However, in the *best case*, the array is already sorted, and the inner loop terminates immediately. This best case running time is $O(n)$.

Quick sort

In our C implementation of quick sort, we:

1. select the first element of the array as our “pivot”. $O(1)$
2. move all elements that are larger than the pivot to the back of the array. $O(n)$.
3. move (“swap”) the pivot into the correct position. $O(1)$.
4. recursively sort the “smaller than” sub-array and the “larger than” sub-array. $T(?)$

The analysis of step 4 is a little trickier.

When the pivot is in “the middle” it splits the sublists equally, so

$$T(n) = O(n) + 2T(n/2) = O(n \log n)$$

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
selection sort	$O(n^2)$	$O(n^2)$
insertion sort	$O(n)$	$O(n^2)$
quick sort	$O(n \log n)$	$O(n^2)$

From this table, it might appear that insertion sort is the best choice.

However, as mentioned with quick sort, the “typical” or “average” case for quick sort is much better than insertion sort.

In Section 10, we will see merge sort, which is $O(n \log n)$ in the worst case.

Binary search

In Section 07, we implemented binary search on a sorted array.

```
int find_sorted(int item, const int a[], int len) {
  // ...
  while (low <= high) {
    mid = (low + high) / 2;
    // ...
    if (a[mid] < item) {
      low = mid + 1;
    } else {
      high = mid - 1;
  //...
```

In each iteration, the size of the search range (n = `high-low`) was halved, so the running time is:

$$T(n) = \sum_{i=1}^{\log_2 n} O(1) = O(\log n)$$

Algorithm Design

In this introductory course, the algorithms we develop are mostly straightforward.

To provide some insight into *algorithm design*, we introduce a problem that is simple to describe, but hard to solve efficiently.

We present four different algorithms to solve this problem, each with a different running time.

The maximum subarray problem

Problem: Given an array of integers, find the **maximum sum** of any *contiguous* sequence (subarray) of elements.

For example, for the following array:

31	-41	59	26	-53	58	97	-93	-23	84

the maximum sum is 187:

31	-41	59	26	-53	58	97	-93	-23	84

This problem has many applications, including *pattern recognition* in *artificial intelligence*.

Solution A: $O(n^3)$

```
// for every start position i and ending position j
// loop between them (k) summing elements
int max_subarray(const int a[], int len) {
  int maxsofar = 0;
  int sum = 0;
  for (i = 0; i < len; ++i) {
    for (j = i; j < len; ++j) {
      sum = 0;
      for (k = i; k <= j; ++k) {
        sum += a[k];
      }
      maxsofar = max(maxsofar, sum);
    }
  }
  return maxsofar;
}
```

$$T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} O(1) = O(n^3)$$

Solution B: $O(n^2)$

```
// for every start position i,
// check if the sum from i...j is the max

int max_subarray(const int a[], int len) {
  int maxsofar = 0;
  int sum = 0;
  for (i = 0; i < len; ++i) {
    sum = 0;
    for (j = i; j < len; ++j) {
      sum += a[j];
      maxsofar = max(maxsofar, sum);
    }
  }
  return maxsofar;
}
```

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

Solution C: $O(n \log n)$

We only describe this recursive *divide and conquer* approach.

1. Find the midpoint position m. $O(1)$
2. Find (a) the maximum subarray from `(0...m-1)`, and
   (b) the maximum subarray from `(m+1...len-1)`. $2T(n/2)$
3. Find (c) the maximum subarray that includes m. $O(n)$
4. Find the maximum of (a), (b) and (c). $O(1)$

$$T(n) = O(n) + 2T(n/2) = O(n \log n)$$

Solution D: $O(n)$

```
// for each position i, keep track of
// the maximum subarray ending at i

int max_subarray(const int a[], int len) {
  int maxsofar = 0;
  int maxendhere = 0;
  for (i = 0; i < len; ++i) {
    maxendhere = max(maxendhere + a[i], 0);
    maxsofar = max(maxsofar, maxendhere);
  }
  return maxsofar;
}
```

In this introductory course, you are not expected to be able to come up with this solution yourself.

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 return 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 2 3))
=> (+ 1 (sum '(2 3)))
=> (+ 1 (+ 2 (sum '(3))))
=> (+ 1 (+ 2 (+ 3 (sum empty))))
=> (+ 1 (+ 2 (+ 3 0)))
=> (+ 1 (+ 2 3))
=> (+ 1 5)
=> 6

(asum '(1 2 3))
=> (asum/acc '(1 2 3) 0)
=> (asum/acc '(2 3) 1)
=> (asum/acc '(3) 3)
=> (asum/acc empty 6)
=> 6
```

The `sum` expression “grows” to $O(n)$ +’s, but the `asum` expression does not use any additional space.

The measured run-time of `asum` is *significantly* faster than `sum` (in an experiment with a list of one million `1`'s, over `40` times faster).

`sum` uses $O(n)$ space, whereas `asum` uses $O(1)$ space.

But **both** functions make the **same** number of recursive calls, how is this explained?

The difference is that `asum` uses **tail recursion**.

A function is ***tail recursive*** if the recursive call is always the **last expression** to be evaluated (the “tail”).

Typically, this is achieved by using accumulative recursion and providing a partial result as one of the parameters.

With tail recursion, the previous stack frame can be **reused** for the next recursion (or the previous frame can be discarded before the new stack frame is created).

Tail recursion is more space efficient and avoids stack overflow.

Many modern C compilers detect and take advantage of tail recursion.

Big O revisited

We now revisit *Big O notation* and define it more formally.

$O(g(n))$ is the **set** of all functions whose "order" is **less than or equal** to $g(n)$.

$$n^2 \in O(n^{100})$$
$$n^3 \in O(2^n)$$

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.

Big O describes the *asymptotic* behaviour of a function.

This is **different** than describing the **worst case** behaviour of an algorithm.

Many confuse these two topics but they are completely **separate concepts**. You can asymptotically define the best case and the worst case behaviour of an algorithm.

For example, the best case insertion sort is $O(n)$, while the worst case is $O(n^2)$.

A slightly more formal definition of Big O is

$$\boxed{\begin{array}{l} f(n) \in O(g(n)) \Leftrightarrow f(n) \leq c \cdot g(n) \\ \text{for large } n \text{ and some positive number } c \end{array}}$$

This definition makes it clear why we *"ignore"* constant coefficients.

For example,

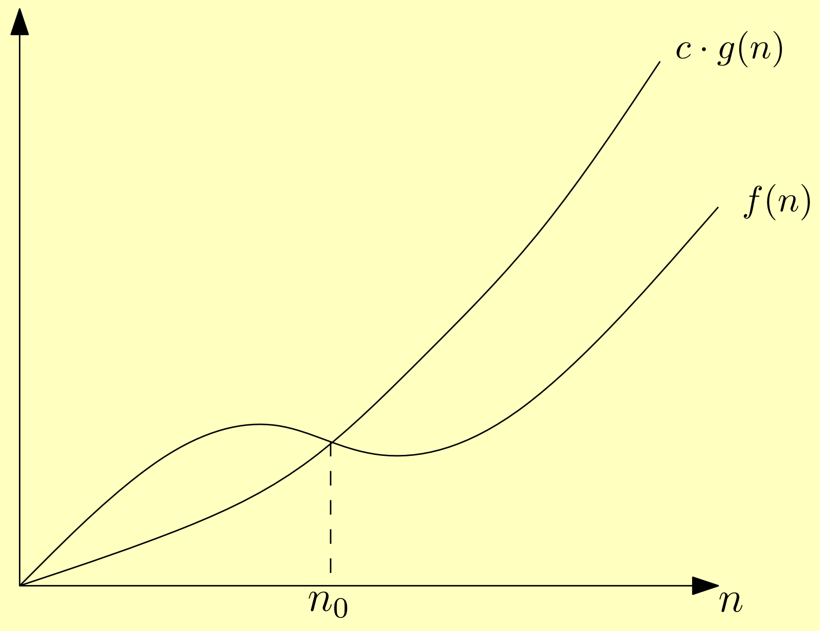
$9n \in O(n)$ for $c = 10,$ $9n \leq 10n$, and

$0.01n^3 + 1000n^2 \in O(n^3)$
for $c = 1001,$ $0.01n^3 + 1000n^2 \leq 1001n^3$

The full definition of Big O is

$$f(n) \in O(g(n)) \Leftrightarrow \exists c, n_0 > 0, \forall n \geq n_0, f(n) \leq c \cdot g(n)$$

$f(n)$ is in $O(g(n))$ if there exists a positive c and n_0 such that for any value of $n \geq n_0$, $f(n) \leq c \cdot g(n)$.



In later CS courses, you will use the formal definition of Big O to *prove* algorithm behaviour more rigourously.

There are other asymptotic functions in addition to Big O.

(for each of the following, $\exists n_0 > 0, \forall n \geq n_0 \ldots$)

$$f(n) \in \omega(g(n)) \Leftrightarrow \forall c > 0, c \cdot g(n) \leq f(n)$$
$$f(n) \in \Omega(g(n)) \Leftrightarrow \exists c > 0, c \cdot g(n) \leq f(n)$$
$$f(n) \in \Theta(g(n)) \Leftrightarrow \exists c_1, c_2 > 0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$
$$f(n) \in O(g(n)) \Leftrightarrow \exists c > 0, f(n) \leq c \cdot g(n)$$
$$f(n) \in o(g(n)) \Leftrightarrow \forall c > 0, f(n) \leq c \cdot g(n)$$

$O(g(n))$ is often used when $\Theta(g(n))$ is more appropriate.

Goals of this Section

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

- use the new terminology introduced (*e.g.,* algorithm, time efficiency, running time, order)
- compute the order of an expression
- explain and demonstrate the use of Big O notation and how n is used to represent the size of the data
- determine the “worst case” running time for a given implementation

- deduce the running time for many built-in functions
- 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

Strings

Readings: CP:AMA 13

The primary goal of this section is to be able to use strings.

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'` is equivalent to `0`. That is different from `'0'`, which is equivalent to 48 (the ASCII character for the symbol zero).

String initialization

The following definitions create equivalent 4-character arrays:

```
char a[4] = {'c', 'a', 't', '\0'};
char b[4] = {'c', 'a', 't', 0};
char c[4] = {'c', 'a', 't'};
char d[4] = { 99,  97, 116, 0};
char e[4] = "cat";
char f[4] = "cat\0";
```

Because they all have a null terminator, they are also strings.

C supports an *automatic* length declaration ([ ]), where the length is determined by the initialization.

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

If you combine the automatic length declaration with double quote("), initialization, it adds the null terminator for you.

```
// these are equivalent
char a[4] = {'c', 'a', 't', '\0'};
char b[] =  "cat";
```

As we will explain letter, the double quotes used in array **initialization** is **different** than the quotes used in expressions (*e.g.*, in `printf("string")`).

Null termination

With null terminated strings, we do not need to pass the *length* to functions. It is determined by the location of the `'\0'`.

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

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

It is good style to have `const` parameters to communicate that no changes (mutation) occurs to the string.

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);
}
```

In practice, pointer notation is often used with strings as it is slightly faster. Using array index notation (`s[i]`) performs an extra addition in the loop.

Lexicographical order

Characters can be easily compared (`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 *addresses* (pointers), which is not helpful.

To compare strings we are typically interested in using a **lexicographical order**.

Strings require us to be more careful with our terminology, as “smaller than” and “greater than” are ambiguous: are we considering just the **length** of the string? To avoid this problem we use **precedes** (“before”) and **follows** (“after”).

To compare two strings using a **lexicographical order**, we first compare the first character of each string. If they are different, the string with the smaller first character *precedes* the other string. Otherwise (the first characters are the same), the second characters are compared, and so on.

If the end of one string is encountered, it *precedes* the other string. Two strings are equal (the same) if they are the same length and all of their characters are identical.

The following strings are in lexicographical order:

```
  "" "a" "az" "c" "cab" "cabin" "cat" "catastrophe"
```

The <`string.h`> library function `strcmp` uses lexicographical ordering.

`strcmp(s1, s2)` returns zero if the strings are identical. If `s1` precedes `s2`, it returns a negative integer. Otherwise (`s1` follows `s2`) it returns a positive integer.

```
int my_strcmp(const char s1[], const char s2[]) {
  int i = 0;
  while (s1[i] == s2[i]) {
    if ((s1[i] == '\0') && (s2[i] == '\0')) return 0;
    ++i;
  }
  if (s1[i] < s2[i]) return -1;
  return 1;
}
```

To compare if two strings are *equal* (identical), use the `strcmp` function and check for **zero (false)**.

```
char a[] = "the same?";
char b[] = "the same?";
char c[] = "different";

trace_bool(strcmp(a, b) == 0);
trace_bool(!strcmp(a, b));
trace_bool(!strcmp(a, c));
```

```
strcmp(a, b) == 0 => true
!strcmp(a, b) => true
!strcmp(a, c) => false
```

Never use the equality operator (==) to compare strings. It compares the *addresses* of the strings, not their contents.

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.

`printf` does not print out the null character.

When using `%s` with `scanf`, it stops reading the string when a whitespace 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?\n");
  scanf("%s", name);
```

You must be very careful to reserve enough space for the string to be read in, and **do not forget the null character**.

`scanf("%s", ...)` automatically adds the null character.

example: understanding scanf

```
char name[10] = {0};
while (scanf("%s", name) == 1) {
  printf("Hello, %s!\n", name);
}
```

The input:

```
Samantha Bob [EOF]
```

Produces the following output:

```
Hello, Samantha!
Hello, Bob!
```

Afterward, what is stored in the `name` array?

B	o	b	\0	n	t	h	a	\0	\0

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

```
char name[81];
printf("What is your first name?\n");
scanf("%s", name);
```

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

example 1: scanf and buffers

```c
int main(void) {
  char name[8];
  char message[] = "Hello.";
  char prompt[] = "What is your name?";
  while (1) {
    printf("message: %s\n", message);
    printf("prompt:  %s\n", prompt);
    if (scanf("%s", name) != 1) break;
    printf("Welcome, %s!\n", name);
  }
}
```

In this example, entering a long name causes C to write characters beyond the length of the `name` array. Eventually, it overwrites the memory where `message` is stored, and if long enough, where `prompt` is stored.

This is known as a ***buffer overrun*** (or *buffer overflow*). The C language is especially susceptible to *buffer overruns*, which can cause serious stability and security problems.

In this introductory course, having an array with an appropriate length and using `scanf` is “good enough”.

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

example 2: scanf and buffers

```
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")) {
       printf("Your balance is: %d\n", balance);
     } else if (!strcmp(command, "deposit")) {
       printf("Enter your deposit amount: ");
       int dep;
       scanf("%d", &dep);
       balance += dep;
     } else if (!strcmp(command, "q")) {
       printf("Bye!\n"); break;
     } else {
       printf("Invalid command. Please try again.\n");
     }
  }
}
```

In this banking example, entering a long command causes C to write characters beyond the length of the `command` array. Eventually, it overwrites the memory where `balance` is stored.

It writes four `char`s into the four bytes where `balance` is stored. The value of `balance` is a “re-interpretation” of those four bytes as an `int`, instead of four `char`s.

If you need to read in a string that includes whitespace until a newline (`\n`) is encountered, the `gets` function can be used (CP:AMA 13.3).

It is also very susceptible to overruns, but is convenient to use in this course.

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

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

One popular strategy to avoid overruns is to only read in one character at a time (*e.g.*, with `scanf("%c")` or `getchar`). For an example of using `getchar` to avoid overruns, see CP:AMA 13.3.

Two additional <`string.h`> library functions that are useful, but susceptible to buffer overruns are:

`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).

Consider this simple implementation of `my_strcpy`:

```
char *my_strcpy(char *dst, const char *src) {
   char *d = dst;
   while (*src) {
      *d = *src;
      ++d; ++src;
   }
   *d = '\0';
   return dst;
}
```

with the following function call:

```
char s[9] = "spam";
my_strcpy(s + 4, s);
```

The null terminator of `src` is overwritten, so it will continue to fill up memory with `spamspamspam...` until a crash occurs.

While *writing* to a buffer can cause dangerous buffer overruns, *reading* an improperly terminated string can also cause problems.

```
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`.

String literals

C strings in quotations (*e.g.*, `"string"`) that are in an **expression** (*i.e.*, *not* part of an *array initialization*) are known as ***string literals***.

```
printf("literal\n");

printf("literal %s\n", "another literal");

if (!strcmp(s, "literal")) ...

strcpy(dst, "literal");

int i = strlen("literal");

scanf("%d", &i);
```

String literal storage

Where are string literals stored?

For each *string literal*, a null-terminated `const char` array is created in the **read-only data** section.

In the code, the occurrence of the *string literal* is replaced with the 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 name is actually given to each literal, it is helpful to imagine that one is:

```
const char string_literal_1[] = "i = %d\n";
const char string_literal_2[] = "the value of j is %d\n";

void foo(int i, int j) {
  printf(string_literal_1, i);
  printf(string_literal_2, j);
}
```

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

Note the subtle difference between the following two definitions:

```
int main(void) {
  char a[] = "mutable char array";
  char *p  = "constant string literal";
  //...
}
```

Once again, it is helpful to think of the string literal as a separately defined `const char` array.

```
const char string_literal_1[] = "constant string literal";

int main(void) {
  char a[] = "mutable char array";
  char *p  = string_literal_1;
  //...
}
```

Arrays vs. pointers

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

Consider again two similar string definitions:

```
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 more similar to a **constant** pointer (that cannot change what it “points at”).

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

In most practical expressions a and p would be equivalent. The only significant differences between them are:

- a has the same value as &a, while p and &p have different values
- The size of a is 24 bytes, while `sizeof(p)` is 8

C running times (strings & I/O)

<`string.h`> functions (*e.g.*, `strlen`, `strcpy`) are $O(n)$, where n is the length of the string. For `strcmp`, n is the length of the smallest string.

<`stdio.h`> functions `printf` and `scanf` are $O(1)$, except when working with strings (`"%s"`), which are $O(n)$, where n is the length of the string.

Note that the string literal used with `printf` must always be constant length (*i.e.*, `printf("literal")`).

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

```
int char_count(char c, char *s) {
  int count = 0;
  for (int i = 0; i < strlen(s); ++i) {    // BAD !!!!
    if (s[i] == c) ++count;
  }
  return count;
}
```

By using an $O(n)$ function (`strlen`) inside of the loop, the function becomes $O(n^2)$ instead of $O(n)$.

Unfortunately, this mistake is common amongst beginners.

This will be harshly penalized on assignments & exams.

Arrays of Strings

An array of strings can be defined as a 2D array of `char`s, but it is awkward and rarely used.

Instead, an **array of pointers** is more common.

```
  char *aos[] = {"my awesome array", "of string", "literals"};
```

In the above example, `aos` is an array of pointers, with each pointer pointing to a string literal.

Despite being a “proper” 2D array, you can access any `char` as if it was in a 2D array of `chars`.

For example, `aos[0][1]` is `(aos[0])[1]`, which is `'y'`.

```
// equivalent definition

const char str_lit_0[] = "my awesome array";
const char str_lit_1[] = "of string";
const char str_lit_2[] = "literals";

char *aos[] = {str_lit_0, str_lit_1, str_lit_2};
```

This array of pointers can be passed to a function, but as with all arrays, you must still pass the array length:

```
void aos_function(char *aos[], int num_strings) { ... }
// OR
void aos_function(char **aos,  int num_strings) { ... }
```

For complicated technical reasons, do not worry about adding `const` to parameters/definitions that are arrays of pointers.

Until we learn how to use dynamic memory, defining an array of *mutable* strings is a little more awkward.

You must define each mutable string separately.

```
  char s0[] = "my mutable array";
  char s1[] = "of strings";
  char *aos[] = {s0, s1};
```

A 2D array of `chars` requires that each string is allocated the same fixed number of `chars` (regardless of the actual string length).

```
  char aos2d[3][21] = {"my", "two dimensional", "char array"};
```

This is awkward because a function would need to know the fixed length in advance.

```
  void aos_function(char aos2d[][21], int num_strings) { ... }
```

If necessary, the array could be “re-interpreted” (cast) as a 1D array, and the fixed lengths could be passed as parameters.

Goals of this Section

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

- define and initialize strings
- 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)

Dynamic Memory & ADTs in C

Readings: CP:AMA 17.1, 17.2, 17.3, 17.4

The primary goal of this section is to be able to use dynamic memory.

The heap

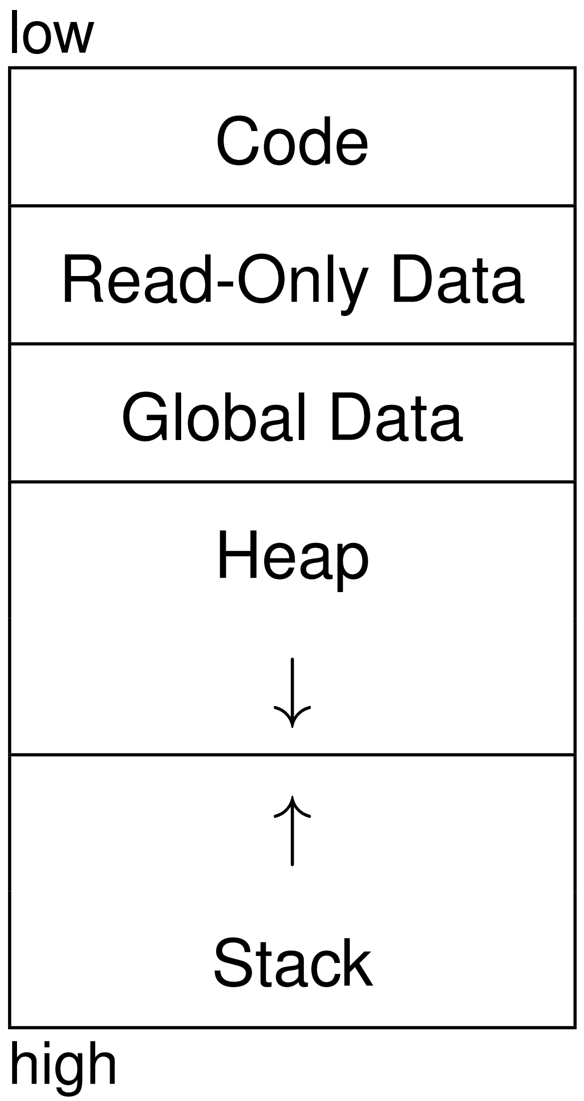
The ***heap*** is the final section in the C memory model.

It can be thought of as a big “pile” (or “pool”) of memory that is available to your program.

Memory is **dynamically** *“borrowed”* from the heap. We call this ***allocation***.

When the borrowed memory is no longer needed, it can be *“returned”* and possibly **reused**. We call this ***deallocation***.

If too much memory has already been allocated, attempts to borrow additional memory fail.


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” has stuck.

A similar problem arises with “the stack” region of memory because there is also a Stack ADT. However, their behaviour is very similar so it is far less confusing.

malloc

The `malloc` (**m**emory **alloc**ation) function obtains memory from the heap *dynamically*. It is provided in <`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
// time: O(1) [close enough for this course]
```

For example, if you want enough space for an array of 100 `int`s:

```
int *my_array = malloc(100 * sizeof(int));
```

or a single `struct posn`:

```
struct posn *my_posn = malloc(sizeof(struct posn));
```

These two examples illustrate the most common use of dynamic memory: allocating space for **arrays and structures**.

You should always use `sizeof` with `malloc` to improve portability and to improve communication.

`Seashell` allows

```
  int *my_array = malloc(400);
```

instead of

```
  int *my_array = malloc(100 * sizeof(int));
```

but the latter is much better style and is more portable.

Strictly speaking, the type of the `malloc` parameter is `size_t`, which is a special type produced by the `sizeof` operator.

`size_t` and `int` are different types of integers.

`Seashell` is mostly forgiving, but in other C environments using an `int` when C expects a `size_t` may generate a warning.

The proper `printf` placeholder to print a `size_t` is `%zd`.

The declaration for the `malloc` function is:

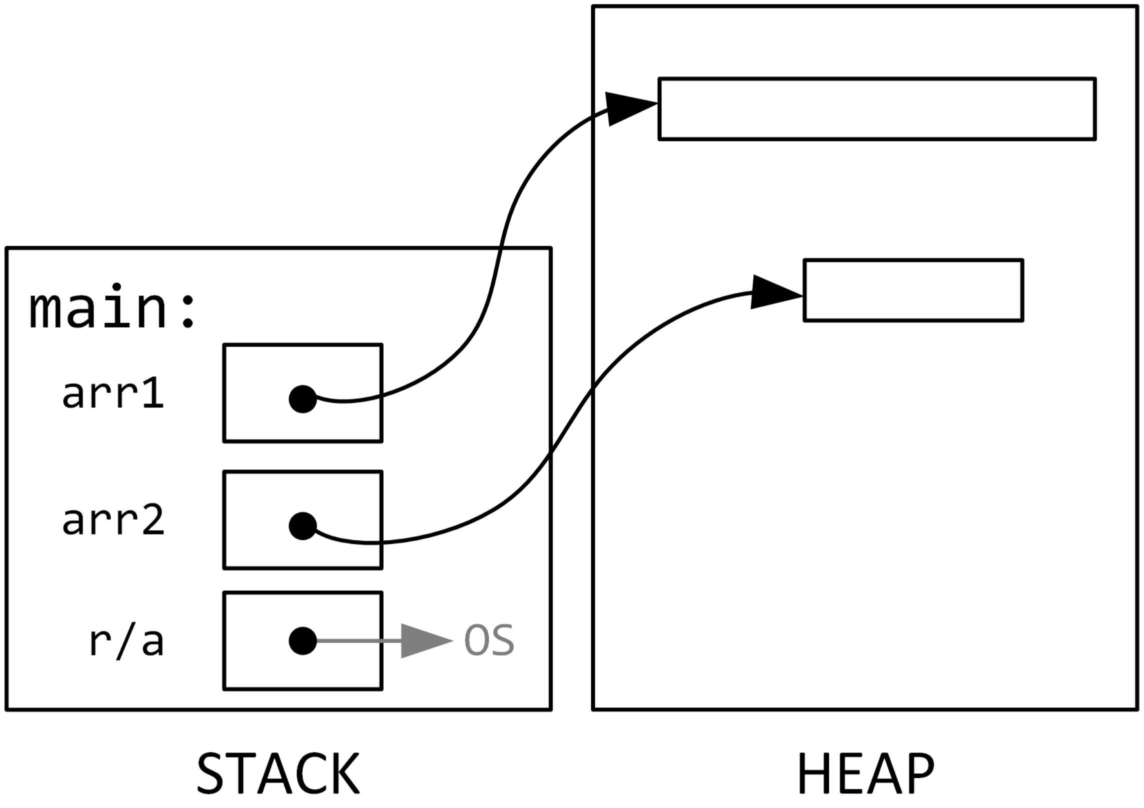
```
void *malloc(size_t s);
```

The return type is a (`void *`) (*void pointer*), a special pointer that can point at *any* type.

```
int *my_array = malloc(10 * sizeof(int));

struct posn *my_posn = malloc(sizeof(struct posn));
```

example: visualizing the heap

```
int main(void) {
  int *arr1 = malloc(10 * sizeof(int));
  int *arr2 = malloc(5 * sizeof(int));
  //...
}
```



An unsuccessful call to `malloc` returns `NULL`.

In practice it's good style to check every `malloc` return value and gracefully handle a `NULL` instead of crashing.

```
int *my_array = malloc(n * sizeof(int));
if (my_array == NULL) {
  printf("Sorry, I'm out of memory! I'm exiting....\n");
  exit(EXIT_FAILURE);
}
```

In the "real world" you should always perform this check, but in this course, you do **not** have to check for a `NULL` return value unless instructed otherwise.

In these notes, we omit this check to save space.

The heap memory provided by `malloc` is **uninitialized**.

```
int *a = malloc(10 * sizeof(int));
printf("the mystery value is: %d\n", a[0]);
```

Although `malloc` is very complicated, for the purposes of this course, you can assume that `malloc` is $O(1)$.

There is also a `calloc` function which essentially calls `malloc` and then “initializes” the memory by filling it with zeros. `calloc` is $O(n)$, where n is the size of the block.

free

For every block of memory obtained through `malloc`, you must eventually `free` the memory (when the memory is no longer in use).

```
// free(p) returns memory at p back to the heap
// requires: p must be from a previous malloc
// effects: the memory at p is invalid
// time: O(1)
```

In the `Seashell` environment, you **must** `free` every block.

```
int *my_array = malloc(n * sizeof(int));
// ...
// ...
free(my_array);
```

Invalid after free

Once a block of memory is `free`d, reading from or writing to that memory is invalid and may cause errors (or unpredictable results).

Similarly, it is invalid to `free` memory that was not returned by a `malloc` or that has already been `free`d.

```
int *a = malloc(10 * sizeof(int));
free(a);
int k = a[0];    // INVALID
a[0] = 42;       // INVALID
free(a);         // INVALID
a = NULL;        // GOOD STYLE
```

Pointer variables may still contain the address of the memory that was `free`d, so it is often good style to assign `NULL` to a `free`d pointer variable.

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 *my_array;
  my_array = malloc(10 * sizeof(int));
  my_array = malloc(10 * sizeof(int)); // Memory Leak!
```

In this example, the address from the original `malloc` has been overwritten.

That memory is now *“lost”* (or *leaked*) and so it can never be `free`d.

Garbage collection

Many modern languages (including Racket) have a ***garbage collector***.

A garbage collector **detects** when memory is no longer in use and **automatically** frees memory and returns it to the heap.

One disadvantage of a garbage collector is that it can be slow and affect performance, which is a concern in high performance computing.

Merge Sort

In ***merge sort***, the array is split (in half) into two separate arrays. The two arrays are sorted and then they are `merge`d back together into the original array.

This is another example of a *divide and conquer* algorithm.

The arrays are *divided* into two smaller problems, which are then sorted (*conquered*). The results are combined to solve the original problem.

To simplify our implementation, we use a `merge` helper function.

```
// merge(dest, src1, len1, src2, len2) modifies dest to contain
//   the elements from both src1 and src2 in sorted order
// requires: length of dest is at least (len1 + len2)
//           src1 and src2 are sorted
// effects: modifies dest
// time: O(n), where n is len1 + len2

void merge(int dest[], const int src1[], int len1,
                       const int src2[], int len2) {
  int pos1 = 0;
  int pos2 = 0;
  for (int i = 0; i < len1 + len2; ++i) {
    if (pos1 == len1 || (pos2 < len2 && src2[pos2] < src1[pos1])) {
      dest[i] = src2[pos2];
      ++pos2;
    } else {
      dest[i] = src1[pos1];
      ++pos1;
    }
  }
}
```

```
void merge_sort(int a[], int len) {
  if (len <= 1) return;
  int llen = len / 2;
  int rlen = len - llen;

  int *left = malloc(llen * sizeof(int));
  int *right = malloc(rlen * sizeof(int));

  for (int i = 0; i < llen; ++i) left[i] = a[i];
  for (int i = 0; i < rlen; ++i) right[i] = a[i + llen];

  merge_sort(left, llen);
  merge_sort(right, rlen);

  merge(a, left, llen, right, rlen);

  free(left);
  free(right);
}
```

Merge sort is $O(n \log n)$, even in the *worst case*.

Duration

Using dynamic (heap) memory, a function can obtain memory that **persists after** the function has `return`ed.

```
// build_array(n) returns a new array initialized with
//   values a[0] = 0, a[1] = 1, ... a[n-1] = n-1
// effects: allocates a heap array (caller must free)

int *build_array(int len) {
  assert(len > 0);
  int *a = malloc(len * sizeof(int));
  for (int i = 0; i < len; ++i) {
    a[i] = i;
  }
  return a;     // array exists beyond function return
}
```

The caller (client) is responsible for `free`ing the memory (the contract should communicate this).

The <string.h> function strdup makes a duplicate of a string.

```
// my_strdup(s) makes a duplicate of s
// effects: allocates memory (caller must free)

char *my_strdup(const char *s) {
  char *newstr = malloc((strlen(s) + 1) * sizeof(char));
  strcpy(newstr, s);
  return newstr;
}
```

Recall that the strcpy(dest, src) copies the characters from src to dest, and that the dest array must be large enough.

When allocating memory for strings, don't forget to include space for the null terminator.

strdup is not officially part of the C standard, but common.

Resizing arrays

Because `malloc` requires the size of the block of memory to be allocated, it does not seem to solve the problem:

“What if we do not know the length of an array in advance?”

To solve this problem, we can ***resize*** an array by:

- creating a new array
- copying the items from the old to the new array
- `free`ing the old array

example: resizing an array

As we will see shortly, this is not how it is done in practice, but this is an illustrative example.

```
// my_array has a length of 100
int *my_array = malloc(100 * sizeof(int));

// stuff happens...

// oops, my_array now needs to have a length of 101
int *old = my_array;
my_array = malloc(101 * sizeof(int));
for (int i = 0; i < 100; ++i) {
  my_array[i] = old[i];
}
free(old);
```

realloc

To make resizing arrays easier, there is a `realloc` function.

```
// realloc(p, newsize) resizes the memory block at p
//   to be newsize and returns a pointer to the
//   new location, or NULL if unsuccessful
// requires: p must be from a previous malloc/realloc
// effects: the memory at p is invalid (freed)
// time: O(n), where n is min(newsize, oldsize)
```

Similar to our previous example, `realloc` preserves the contents from the old array location.

```
int *my_array = malloc(100 * sizeof(int));
// stuff happens...
my_array = realloc(my_array, 101 * sizeof(int));
```

The pointer returned by `realloc` may actually be the *original* pointer, depending on the circumstances.

Regardless, after `realloc` **only the new returned pointer can be used**. You should assume that the address passed to `realloc` was `free`d and is now **invalid**.

Typically, `realloc` is used to request a larger size and the additional memory is *uninitialized*.

If the size is smaller, the extraneous memory is discarded.

Although rare, in practice,

```
  my_array = realloc(my_array, newsize);
```

could possibly cause a memory leak if an "out of memory" condition occurs.

In C99, an unsuccessful `realloc` returns `NULL` and the original memory block is not `free`d.

```
// safer use of realloc
int *tmp = realloc(my_array, newsize);
if (tmp) {
  my_array = tmp;
} else {
  // handle out of memory condition
}
```

String I/O: strings of unknown length

In Section 07 we saw how reading in strings can be susceptible to buffer overruns.

```
char str[81];
int retval = scanf("%s", str);
```

The target array is often oversized to ensure there is capacity to store the string. Unfortunately, regardless of the length of the array, a buffer overrun may occur.

To solve this problem we can continuously resize (`realloc`) an array while reading in only one `char`acter at a time.

```
// readstr() reads in a non-whitespace string from I/O
//   or returns NULL if unsuccessful
// effects: allocates memory (caller must free)

char *readstr(void) {
  char c;
  if (scanf(" %c", &c) != 1) return NULL; // ignore initial WS
  int len = 1;
  char *str = malloc(len * sizeof(char));
  str[0] = c;
  while (1) {
    if (scanf("%c", &c) != 1) break;
    if (c == ' ' || c == '\n') break;
    ++len;
    str = realloc(str, len * sizeof(char));
    str[len - 1] = c;
  }
  str = realloc(str, (len + 1) * sizeof(char));
  str[len] = '\0';
  return str;
}
```

Improving readstr

Unfortunately, the running time of `readstr` is $O(n^2)$, where n is the length of the string.

This is because `realloc` is $O(n)$ and occurs inside of the loop.

A better approach might be to allocate **more memory than necessary** and only call `realloc` when the array is “full”.

A popular strategy is to **double** the length of the array when it is full.

Similar to working with *maximum-length arrays*, we need to keep track of the *“actual”* length in addition to the *allocated* length.

```
char *readstr(void) {
  char c;
  if (scanf(" %c", &c) != 1) return NULL; // ignore initial WS
  int maxlen = 1;
  int len = 1;
  char *str = malloc(maxlen * sizeof(char));
  str[0] = c;
  while (1) {
    if (scanf("%c", &c) != 1) break;
    if (c == ' ' || c == '\n') break;
    if (len == maxlen) {
      maxlen *= 2;
      str = realloc(str, maxlen * sizeof(char));
    }
    ++len;
    str[len - 1] = c;
  }
  str = realloc(str, (len + 1) * sizeof(char));
  str[len] = '\0';
  return str;
}
```

With our “doubling” strategy, most iterations are $O(1)$, unless it is necessary to resize (`realloc`) the array.

The resizing time for the first 32 iterations would be:

2,4,0,8,0,0,0,16,0,0,0,0,0,0,0,32,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,64

For n iterations, the total resizing time is at most:

$$2 + 4 + 8 + \ldots + \frac{n}{4} + \frac{n}{2} + n + 2n = 4n - 2 = O(n).$$

By using this doubling strategy, the **total** run time for `readstr` is now only $O(n)$.

ADTs in C

With dynamic memory, we now have the ability to implement an *Abstract Data Type (ADT)* in C.

In Section 06, the first ADT we saw was a simple *stopwatch ADT*. It demonstrated **information hiding**, which provides both *security* and *flexibility*.

It used an **opaque** structure, which meant that the client could not **create** a `stopwatch`.

example: stopwatch ADT

This is the **interface** we used in Section 06.

```
// stopwatch.h [INTERFACE]

struct stopwatch;

// stopwatch_create() creates a new stopwatch at time 0:00
// effects: allocates memory (client must call stopwatch_destroy)
struct stopwatch *stopwatch_create(void);

// stopwatch_destroy(sw) frees memory for sw
// effects: sw is no longer valid
void stopwatch_destroy(struct stopwatch *sw);
```

We can now *complete* our **implementation**.

```
// stopwatch.c [IMPLEMENTATION]

struct stopwatch {
  int seconds;
};
// requires: 0 <= seconds

struct stopwatch *stopwatch_create(void) {
  struct stopwatch *sw = malloc(sizeof(struct stopwatch));
  sw->seconds = 0;
  return sw;
}

void stopwatch_destroy(struct stopwatch *sw) {
  assert(sw);
  free(sw);
}
```

Implementing a Stack ADT

As discussed in Section 06, the stopwatch ADT illustrates the principles of an ADT, but it is not a typical ADT.

The **Stack ADT** (one of the *Collection ADTs*) is more representative.

The interface is nearly identical to the stack implementation from Section 07 that demonstrated *maximum-length arrays*.

The only differences are: it uses an opaque structure, it provides `create` and `destroy` functions, and there is no maximum: it can store an arbitrary number of integers.

```c
// stack.h (INTERFACE)

struct stack;

struct stack *stack_create(void);

bool stack_is_empty(const struct stack *s);

int stack_top(const struct stack *s);

int stack_pop(struct stack *s);

void stack_push(int item, struct stack *s);

void stack_destroy(struct stack *s);
```

The Stack ADT uses the “doubling” strategy.

```
// stack.c (IMPLEMENTATION)

struct stack {
  int len;
  int maxlen;
  int *data;
};

struct stack *stack_create(void) {
  struct stack *s = malloc(sizeof(struct stack));
  s->len = 0;
  s->maxlen = 1;
  s->data = malloc(s->maxlen * sizeof(int));
  return s;
}

void stack_destroy(struct stack *s) {
  free(s->data);
  free(s);
}
```

Most of the operations are identical to the maximum-length implementation.

```
bool stack_is_empty(const struct stack *s) {
  assert(s);
  return s->len == 0;
}

int stack_top(const struct stack *s) {
  assert(s);
  assert(s->len);
  return s->data[s->len - 1];
}

int stack_pop(struct stack *s) {
  assert(s);
  assert(s->len);
  s->len -= 1;
  return s->data[s->len];
}
```

The doubling strategy is implemented in `push`.

```
void stack_push(int item, struct stack *s) {
  assert(s);
  if (s->len == s->maxlen) {
    s->maxlen *= 2;
    s->data = realloc(s->data, s->maxlen * sizeof(int));
  }
  s->data[s->len] = item;
  s->len += 1;
}
```

What is the running time of a single call to `stack_push`?

- $O(n)$ when doubling occurs
- $O(1)$ otherwise (most of the time)

Amortized analysis

To understand ***amortized analysis***, we first consider a more abstract example than `stack_push`.

Homer wants to do some “push-ups” to get some exercise.

His strategy is that on day k, when k is a power of 2, he will do k push-ups. He will then skip $(k - 1)$ days until it is another power of 2.

So the number of push-ups Homer does on the first 31 days is:

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

After 31 days, he has done exactly 31 push-ups. So, **on average**, he is doing one push-up per day.

The analysis for `stack_push` is very similar.

Ignoring any `pop` operations, the **total time** for n calls to `stack_push` is $O(n)$.

The ***amortized*** ("average") time for **each call** is:
$O(n)/n = O(1)$.

In other words, we can say that the *amortized* running time of `stack_push` is $O(1)$.

```
// stack_push(item, s) pushes item onto stack s
// requires: s is a valid stack
// effects:  modifies s
// time:     O(1) [amortized]
```

You will use *amortized* analysis in CS 240 and in CS 341.

In this implementation, we never *"shrink"* the array when items are popped.

A popular strategy is to shrink when the length reaches $\frac{1}{4}$ of the maximum capacity. Although more complicated, this also has an *amortized* run-time of $O(1)$ for an arbitrary sequence of `push`es and `pop`s.

Languages that have a built-in resizable array (*e.g.,* C++'s vector) often use a similar "doubling" strategy.

Goals of this Section

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

- describe the heap
- use the functions `malloc`, `realloc` and `free` to interact with the heap
- explain that the heap is finite, and demonstrate how to check `malloc` for success
- describe memory leaks, how they occur, and how to prevent them

- describe the doubling strategy, and how it can be used to manage dynamic arrays to achieve an amortized $O(1)$ run-time for additions
- create dynamic resizable arrays in the heap
- write functions that create and return a new `struct`
- document dynamic memory side-effects in contracts

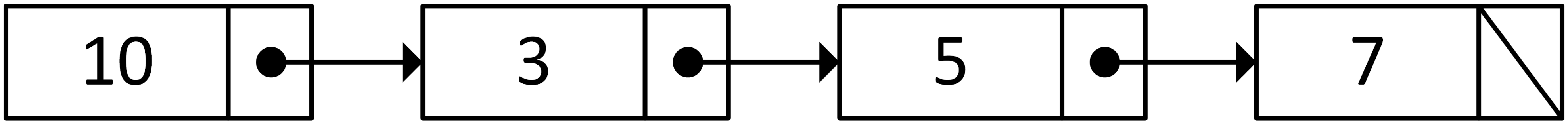
Linked Data Structures

Readings: CP:AMA 17.5

The primary goal of this section is to be able to use linked lists and trees.

Linked lists

Racket's list type is more commonly known as a ***linked list***.

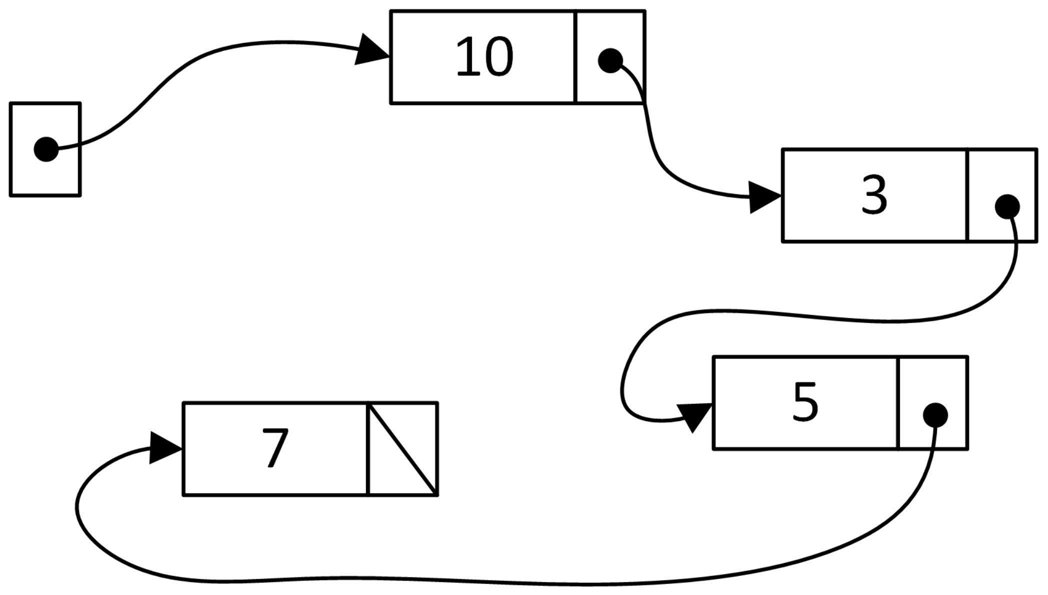


Each ***node*** contains an ***item*** and a **link** (*pointer*) to the ***next*** node in the list.

The *link* in the *last node* is a **sentinel value**.

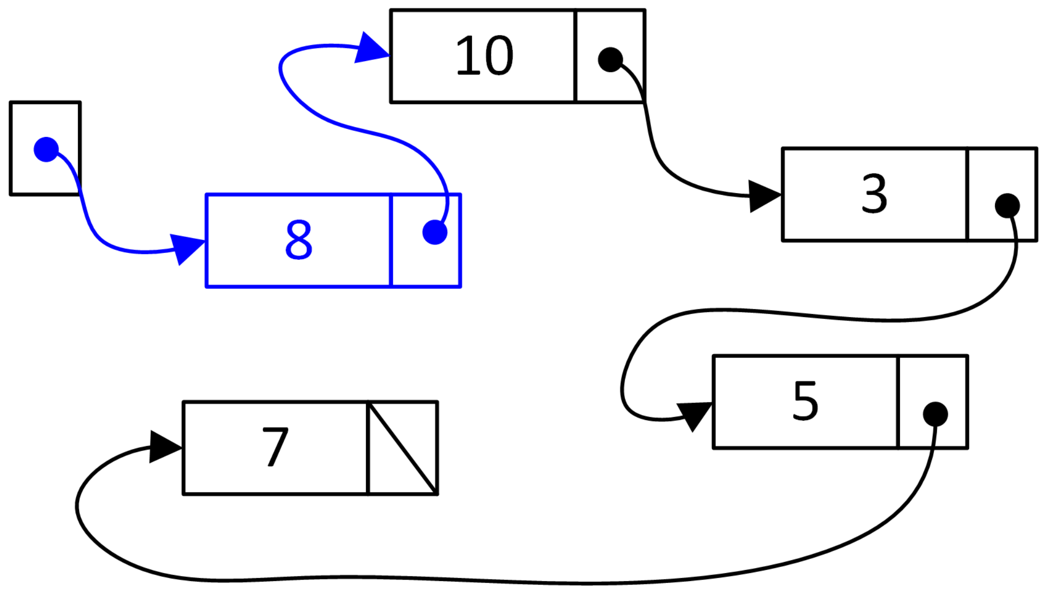
In Racket we use `empty`, and in C we use a `NULL` pointer.

Linked lists are usually represented as a link (pointer) to the ***front***.



Unlike arrays, linked list nodes are **not** arranged sequentially in memory. There is no fast and convenient way to "jump" to the i-th element. The list must be **traversed** from the *front*. Traversing a linked list is $O(n)$.

A significant advantage of a linked list is that its length can easily change, and the length does not need to be known in advance.



The memory for each node is allocated dynamically (*i.e.*, using *dynamic memory*).

Functional *vs.* Imperative approach

In Section 03, we discussed some of the differences between the **functional** and **imperative** programming paradigms.

The core concept of a linked list data structure is independent of any single paradigm.

However, the approach used to **implement** linked list functions are often very different.

Programming with linked lists further illustrates the differences between the two paradigms.

Dynamic memory in Racket

Dynamic memory in Racket is mostly *"hidden"*.

The `cons` function dynamically creates a **new** linked list **node**.

In other words, inside of every `cons` is a hidden `malloc`.

Structure constructors also use dynamic memory

(*e.g.*, `make-posn` or simply `posn` in full Racket).

`list` and quote list notation `'(1 2 3)` *implicitly* use `cons`.

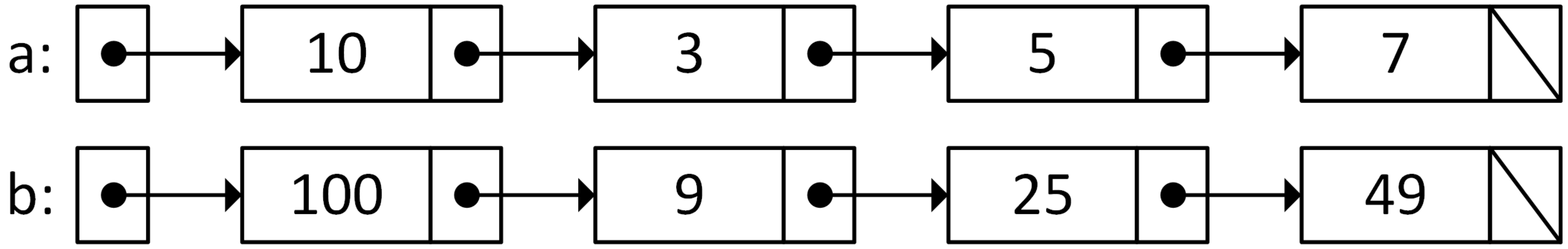
In the functional programming paradigm, functions always generate **new** values rather than changing existing ones.

Consider a function that “squares” a list of numbers.

- In the *functional* paradigm, the function **must** generate a **new** list, because there is no *mutation*.
- in the *imperative* paradigm, the function is more likely to **mutate** an existing list instead of generating a new list.

`sqr-list` uses `cons` to `cons`truct a **new list**:

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

(define a '(10 3 5 7))
(define b (sqr-list a))
```



Of course, in an imperative language (*e.g.,* C) it is *also* possible to write a “square list” function that follows the functional paradigm and generates a new list.

This is another example of why clear communication (purposes and contracts) is so important.

In practice, most imperative list functions perform mutation. If the caller wants a new list (instead of mutating an existing one), they can first make a *copy* of the original list and then mutate the new copy.

Mixing paradigms

Problems may arise if we naïvely use the functional paradigm in an imperative environment without considering the consequences.

This is especially important in C, where there is no garbage collector.

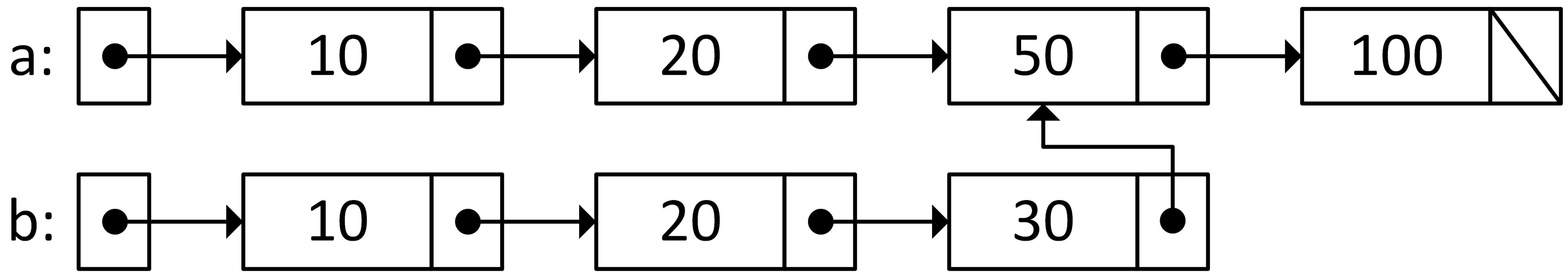
Functional (Racket)	Imperative (C)
no mutation	mutation
garbage collector	no garbage collector
hidden pointers	explicit pointers

The following example highlights the potential problems.

Consider an `insert` function (used in `insertion sort`).

```
;; (insert n slon) inserts n into a sorted list of numbers

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

(define a '(10 20 50 100))
(define b (insert 30 a))
```

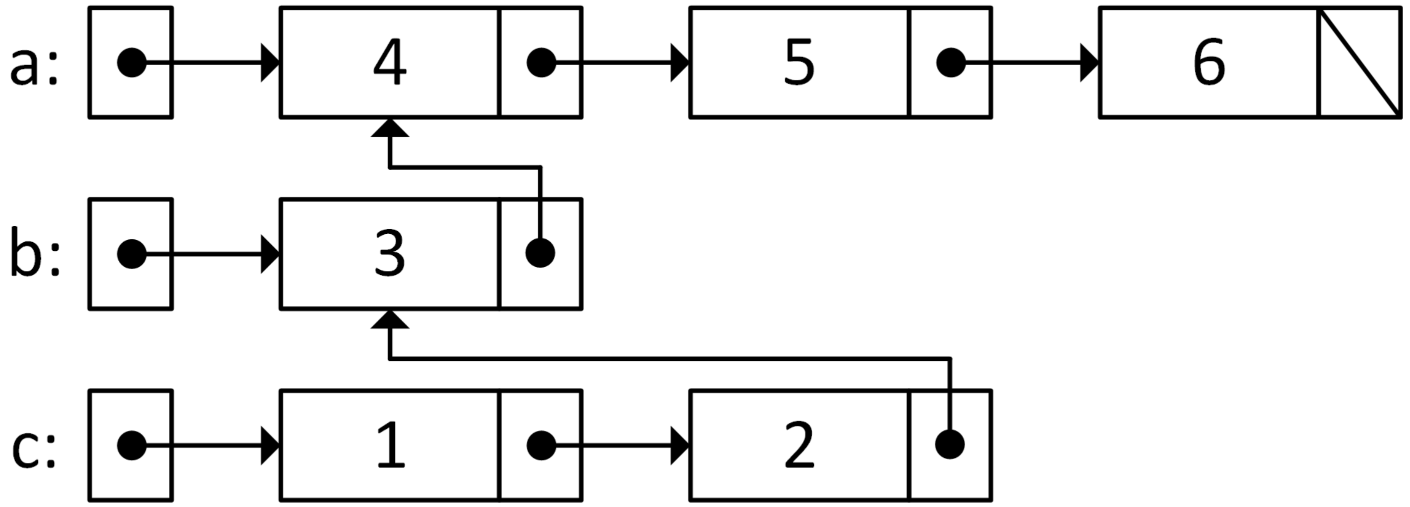
What will the memory diagram look like for a and b?

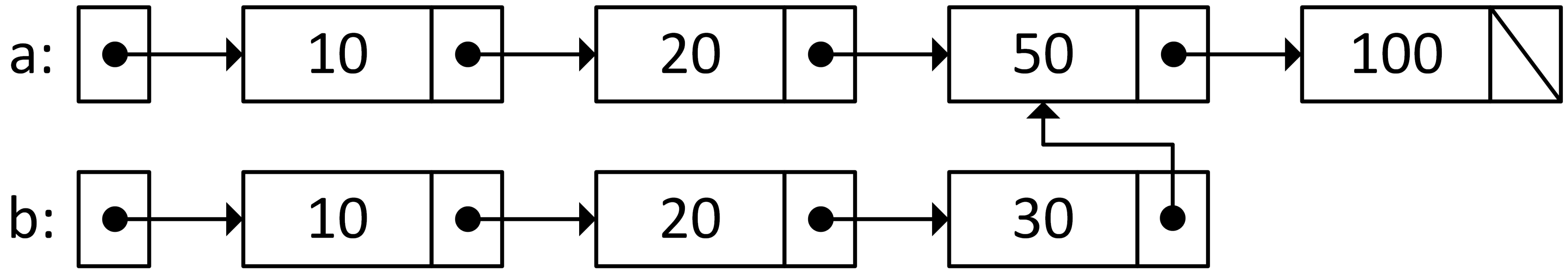


The lists **share the last two nodes**.

example: more node sharing in Racket

```
(define a '(4 5 6))
(define b (cons 3 a))
(define c (append '(1 2) b))
```





In Racket, lists can share nodes with no negative consequences.

It is *transparent* because there is **no mutation**, and there is a **garbage collector**.

In an imperative language like C, this configuration is problematic.

- If we call a mutative function such as “square list” on `a`, then some of the elements of `b` will unexpectedly change.
- If we explicitly `free` all of the memory for list `a`, then list `b` will become invalid.

To avoid mixing paradigms, we use the following *guidelines* when implementing linked lists in C:

- lists shall not **share nodes**
- new nodes shall only be created (`malloc`'d) when necessary (inserting nodes and creating new lists).

In Racket, lists generated with `cons` are immutable.

There is a special `mcons` function to generate a mutable list.

This is one of the significant differences between the Racket language and the Scheme language.

In the Scheme language, lists generated with `cons` are mutable.

Linked lists in C

We declare a *linked list node* (`llnode`) that stores an “item” and a link (pointer) to the next node. For the *last node*, `next` is `NULL`.

```
struct llnode {
  int item;
  struct llnode *next;
};
```

A C structure can contain a *pointer* to its own structure type. This is the first **recursive data structure** we have seen in C.

There is no “official” way of implementing a linked list in C. The CP:AMA textbook and other sources use slightly different conventions.

In this Section we use a ***wrapper strategy***, where we wrap the link to the first node of a list inside of another structure (`llist`).

```
struct llist {
  struct llnode *front;
};
```

This wrapper strategy makes some of the following code more straightforward.

It also makes it easier to avoid having lists share nodes (mixing paradigms).

In an Appendix we present examples that do not use a wrapper and briefly discuss the advantages and disadvantages.

To dynamically create a list, we define a `list_create` function.

```
struct llist *list_create(void) {
  struct llist *lst = malloc(sizeof(struct llist));
  lst->front = NULL;
  return lst;
}



int main(void) {

  struct llist *lst = list_create();
  // ...
}
```

We need to add items to our linked list.

The following code creates a **new** node

```
struct llnode *new_node(int i, struct llnode *pnext) {
  struct llnode *node = malloc(sizeof(struct llnode));
  node->item = i;
  node->next = pnext;
  return node;
}
```

And the following code inserts a new node to the **front** of the list.

```
void add_front(int i, struct llist *lst) {
  lst->front = new_node(i, lst->front);
}
```

Traversing a list

We can *traverse* a list **iteratively** or **recursively**.

When iterating through a list, we typically use a (`llnode`) pointer to keep track of the "current" node.

```
int length(const struct llist *lst) {
  int len = 0;
  struct llnode *node = lst->front;
  while (node) {
    ++len;
    node = node->next;
  }
  return len;
}
```

Remember `(node)` will be false at the end of the list (`NULL`).

When using **recursion**, remember to recurse on a node (`llnode`) not the wrapper list itself (`llist`).

```
int length_nodes(struct llnode *node, int sofar) {
  if (node == NULL) {
    return sofar;
  }
  return length_nodes(node->next, sofar + 1);
}
```

You can write a corresponding wrapper function:

```
int list_length(struct llist *lst) {
  return length_nodes(lst->front, 0);
}
```

Destroying a list

In C, we don't have a *garbage collector*, so we must be able to `free` our linked list. We need to free every node and the list wrapper.

When using an iterative approach, we are going to need *two* node pointers to ensure that the nodes are `free`d in a safe way.

```
void list_destroy(struct llist *lst) {
  struct llnode *curnode = lst->front;
  struct llnode *nextnode = NULL;
  while (curnode) {
    nextnode = curnode->next;
    free(curnode);
    curnode = nextnode;
  }
  free(lst);
}
```

With a recursive approach, it is more convenient to free the *rest* of the list before we `free` the first node.

```
void free_nodes(struct llnode *node) {
  if (node) {
    free_nodes(node->next);
    free(node);
  }
}

void list_destroy(struct llist *lst) {
  free_nodes(lst->front);
  free(lst);
}
```

Duplicating a list

Previously, we used the “square list” function to illustrate the differences between the functional and imperative paradigms.

```
// list_sqr(lst) squares each item in lst
// effects: modifies lst

void list_sqr(struct llist *lst) {
  struct llnode *node = lst->front;
  while (node) {
    node->item *= node->item;
    node = node->next;
  }
}
```

But what if we do want a **new** list that is squared instead of mutating an existing one?

One solution is to provide a `list_dup` function, that makes a *duplicate* of an existing list.

The recursive function is the most straightforward.

```
struct llnode *dup_nodes(struct llnode *oldnode) {
  if (oldnode == NULL) {
    return NULL;
  }
  return new_node(oldnode->item, dup_nodes(oldnode->next));
}

struct llist *list_dup(struct llist *oldlist) {
  struct llist *newlist = list_create();
  newlist->front = dup_nodes(oldlist->front);
  return newlist;
}
```

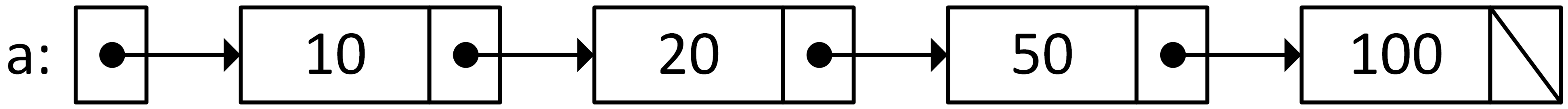
The iterative solution is more complicated:

```
struct llist *list_dup(struct llist *oldlist) {
  struct llist *newlist = list_create();
  if (oldlist->front) {
    newlist->front = new_node(oldlist->front->item, NULL);
    struct llnode *oldnode = oldlist->front->next;
    struct llnode *node = newlist->front;
    while (oldnode) {
      node->next = new_node(oldnode->item, NULL);
      node = node->next;
      oldnode = oldnode->next;
    }
  }
  return newlist;
}
```

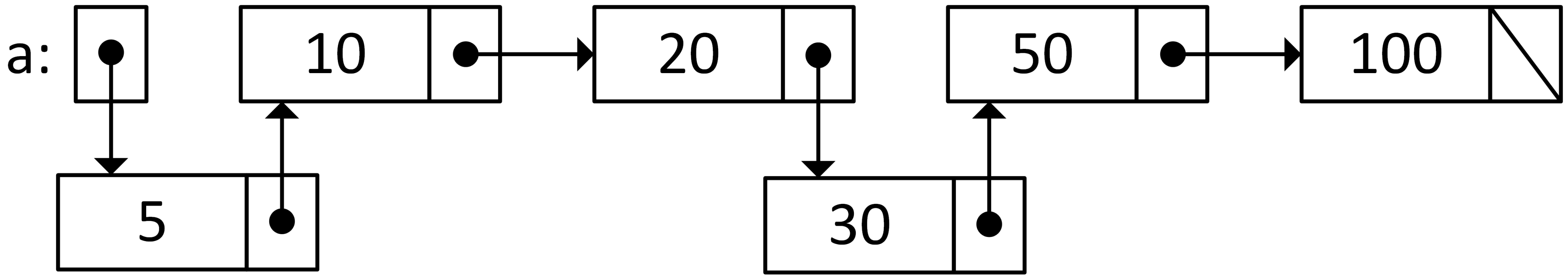
Imperative insert

Earlier, we saw how the Racket (functional) implementation of *insert* (into a sorted list) would be problematic in C.

For an `insert` function in C, we expect the following behaviour:



```
insert( 5, a);
insert(30, a);
```



```c
// insert(i, slst) inserts i into sorted list slst
// effects: modifies slst
// time: O(n), where n is the length of slst

void insert(int i, struct llist *slst) {
  if (slst->front == NULL || i < slst->front->item) {
    add_front(i, slst);
  } else {
    struct llnode *prevnode = slst->front;
    while (prevnode->next && i > prevnode->next->item) {
      prevnode = prevnode->next;
    }
    prevnode->next = new_node(i, prevnode->next);
  }
}
```

Removing nodes

In Racket, the `rest` function does not actually *remove* the `first` element, instead it provides a pointer to the next node.

In C, we can implement a function that removes the first node.

```
void remove_front(struct llist *lst) {
  assert(lst->front);
  struct llnode *old_front = lst->front;
  lst->front = lst->front->next;
  free(old_front);
}
```

Removing a node from an arbitrary list position is more complicated.

```
// remove_item(i, lst) removes the first occurrence of i in lst
//   returns true if item is successfully removed

bool remove_item(int i, struct llist *lst) {
  if (lst->front == NULL) return false;
  if (lst->front->item == i) {
    remove_front(lst);
    return true;
  }
  struct llnode *prevnode = lst->front;
  while (prevnode->next && i != prevnode->next->item) {
    prevnode = prevnode->next;
  }
  if (prevnode->next == NULL) return false;
  struct llnode *old_node = prevnode->next;
  prevnode->next = prevnode->next->next;
  free(old_node);
  return true;
}
```

Revisiting the wrapper approach

Throughout these slides we have used a ***wrapper*** strategy, where we wrap the link to the first node inside of another structure (`llist`).

Some of the advantages of this strategy are:

- cleaner function interfaces
- reduced need for double pointers
- reinforces the imperative paradigm
- less susceptible to misuse and list corruption

The disadvantages of the wrapper approach include:

- slightly more awkward recursive implementations
- extra “special case” code around the first item

However, there is one more significant advantage of the wrapper approach: **additional information** can be stored in the list structure.

Consider that we are writing an application where the `length` of a linked list will be queried often.

Typically, finding the length of a linked list is $O(n)$.

However, we can store (or “cache”) the length *in the wrapper structure*, so the length can be retrieved in $O(1)$ time.

```
struct llist {
  struct llnode *front;
  int length;
};
```

Naturally, other list functions would have to update the `length` as necessary:

- `list_create` would initialize length to zero
- `add_front` would increment length
- `remove_front` would decrement length
- *etc.*

Data integrity

The introduction of the `length` field to the linked list may seem like a great idea to improve efficiency.

However, it introduces new ways that the structure can be corrupted.

What if the `length` field does not accurately reflect the true length?

For example, imagine that someone implements the `remove_item` function, but forgets to update the `length` field?

Or a naïve coder may think that the following statement removes all of the nodes from the list.

```
  lst->length = 0;
```

Whenever the same information is stored in more than one way, it is susceptible to *integrity* (consistency) issues.

Advanced testing methods can often find these types of errors, but you must exercise caution.

If data integrity is an issue, it is often better to repackage the data structure as a separate ADT module and only provide interface functions to the client.

This is an example of **security** (protecting the client from themselves).

Queue ADT

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

Typical queue ADT operations:

- `add_back`: adds an item to the end of the queue
- `remove_front`: removes the item at the front of the queue
- `front`: returns the item at the front
- `is_empty`: determines if the queue is empty

A Stack ADT can be easily implemented using a dynamic array (as we did in Section 10) or with a linked list.

While it is possible to implement a Queue ADT with a dynamic array, the implementation is a bit tricky. Queues are typically implemented with linked lists.

The only concern is that an `add_back` operation is normally $O(n)$.

However, if we maintain a pointer to the back (last element) of the list, in addition to a pointer to the front of the list, we can implement `add_back` in $O(1)$.

Maintaining a `back` pointer is a popular modification to a traditional linked list, and another reason to use a wrapper.

```
// queue.h

// all operations are O(1) (except destroy)

struct queue;

struct queue *queue_create(void);

void queue_add_back(int i, struct queue *q);

int queue_remove_front(struct queue *q);

int queue_front(struct queue *q);

bool queue_is_empty(struct queue *q);

void queue_destroy(struct queue *q);
```

```
// queue.c (IMPLEMENTATION)

struct llnode {
  int item;
  struct llnode *next;
};

struct queue {
  struct llnode *front;
  struct llnode *back;      // <--- NEW
};

struct queue *queue_create(void) {
  struct queue *q = malloc(sizeof(struct queue));
  q->front = NULL;
  q->back = NULL;
  return q;
}
```

```
void queue_add_back(int i, struct queue *q) {
  struct llnode *node = new_node(i, NULL);
  if (q->front == NULL) {
    q->front = node;
  } else {
    q->back->next = node;
  }
  q->back = node;
}

int queue_remove_front(struct queue *q) {
  assert(q->front);
  int retval = q->front->item;
  struct llnode *old_front = q->front;
  q->front = q->front->next;
  free(old_front);
  if (q->front == NULL) {
    q->back = NULL;
  }
  return retval;
}
```

The remainder of the Queue ADT is straightforward.

```
int queue_front(struct queue *q) {
  assert(q->front);
  return q->front->item;
}

bool queue_is_empty(struct queue *q) {
  return q->front == NULL;
}

void queue_destroy(struct queue *q) {
  while (!queue_is_empty(q)) {
    queue_remove_front(q);
  }
  free(q);
}
```

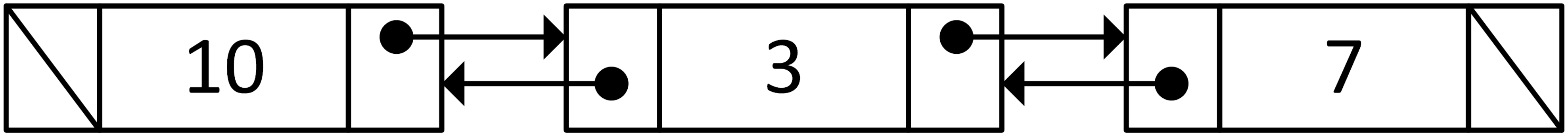
Node augmentation strategy

In a **node augmentation strategy**, each *node* is *augmented* to include additional information about the node or the structure.

For example, a **dictionary** node can contain both a *key* (item) and a corresponding *value*.

Or for a **priority queue**, each node can additionally store the priority of the item.

The most common node augmentation for a linked list is to create a ***doubly linked list***, where each node also contains a pointer to the *previous* node. When combined with a `back` pointer in a wrapper, a doubly linked list can add or remove from the front **and back** in $O(1)$ time.

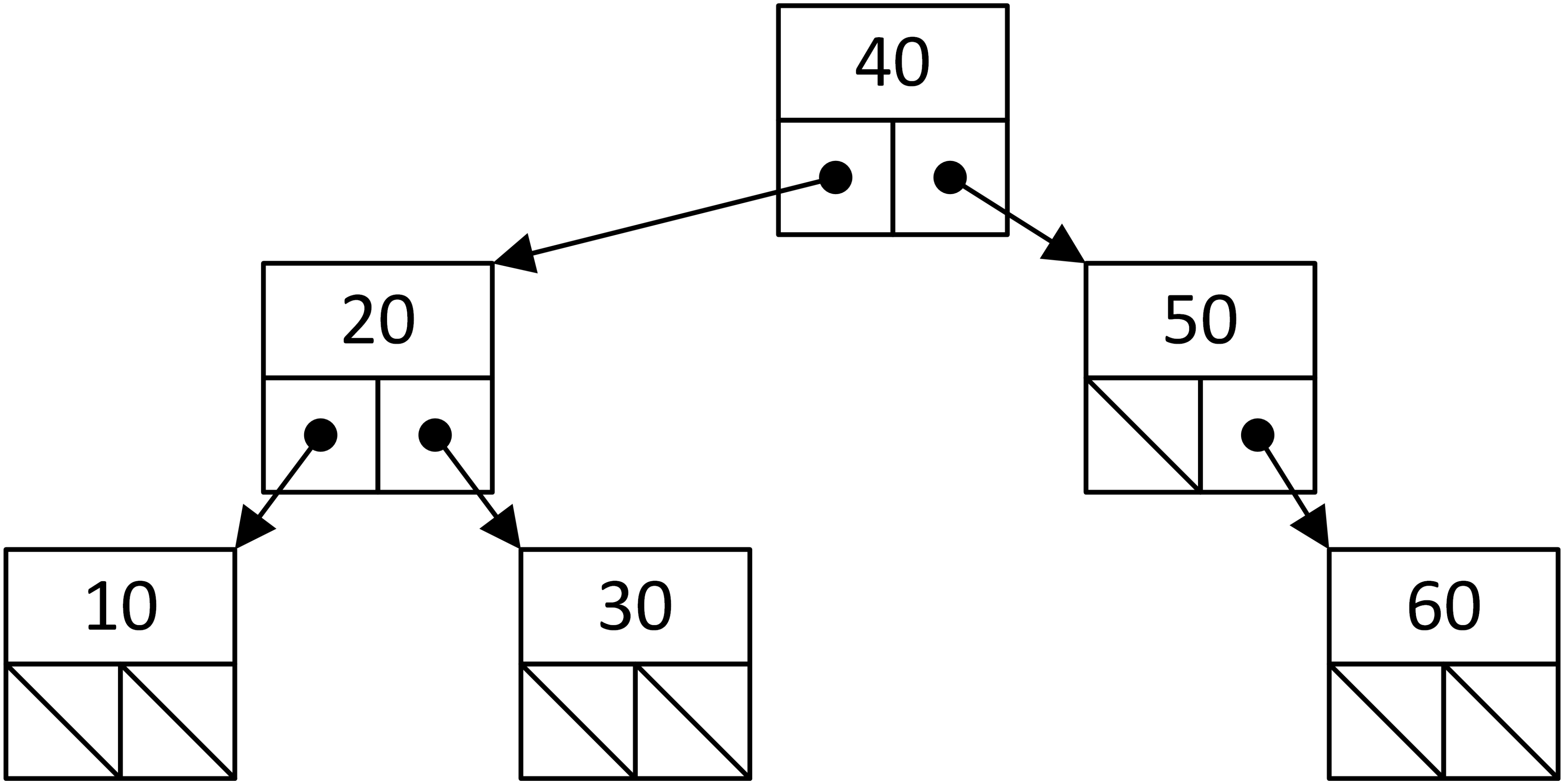


Many programming environments provide a Double-Ended Queue (dequeue or deque) ADT, which can be used as a Stack or a Queue ADT.

Trees

At the implementation level, ***trees*** are very similar to linked lists.

Each node can *link* to more than one node.

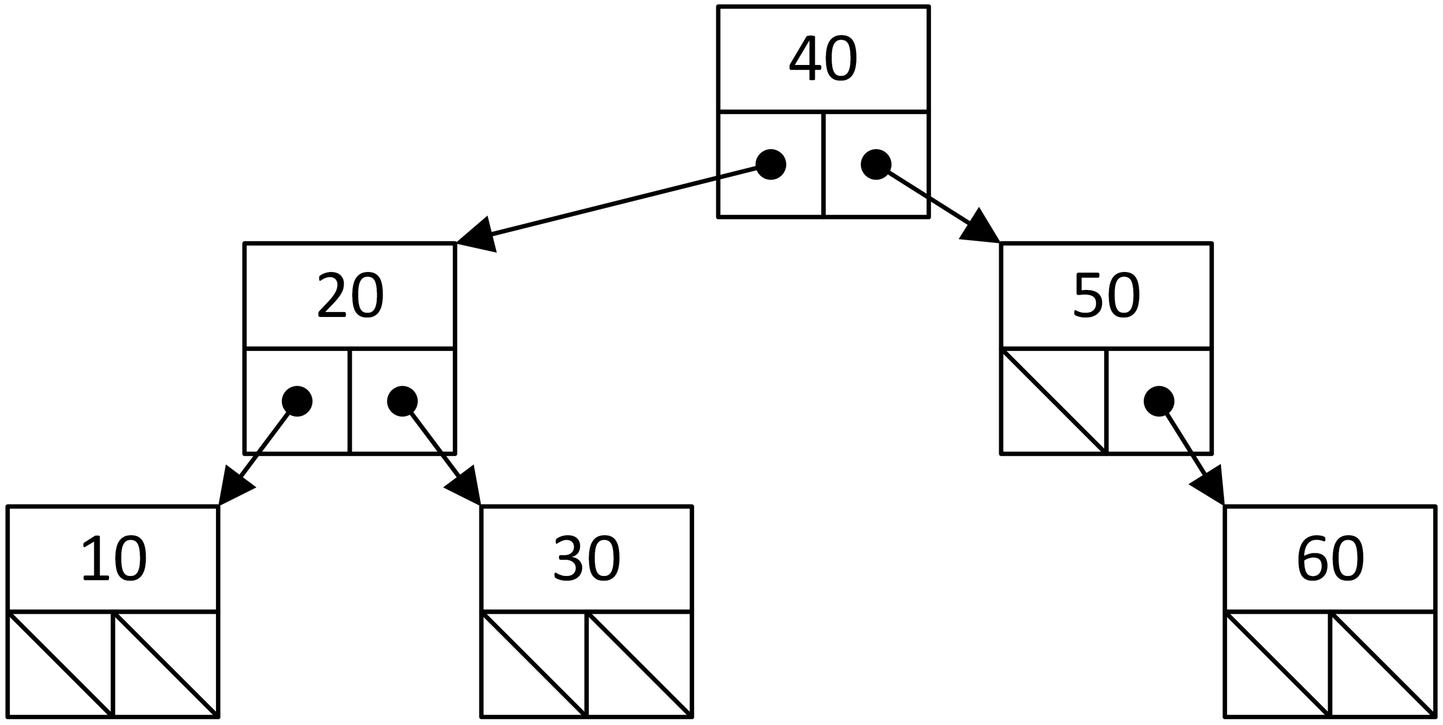


Tree terminology

- the **root node** has no **parent**, all others have exactly one
- nodes can have multiple **children**
- in a **binary tree**, each node has at most two children
- a **leaf node** has no children
- the **height** of a tree is the maximum possible number of nodes from the root to a leaf (inclusive)
- the height of an empty tree is zero
- the number of nodes is known as the **node count**

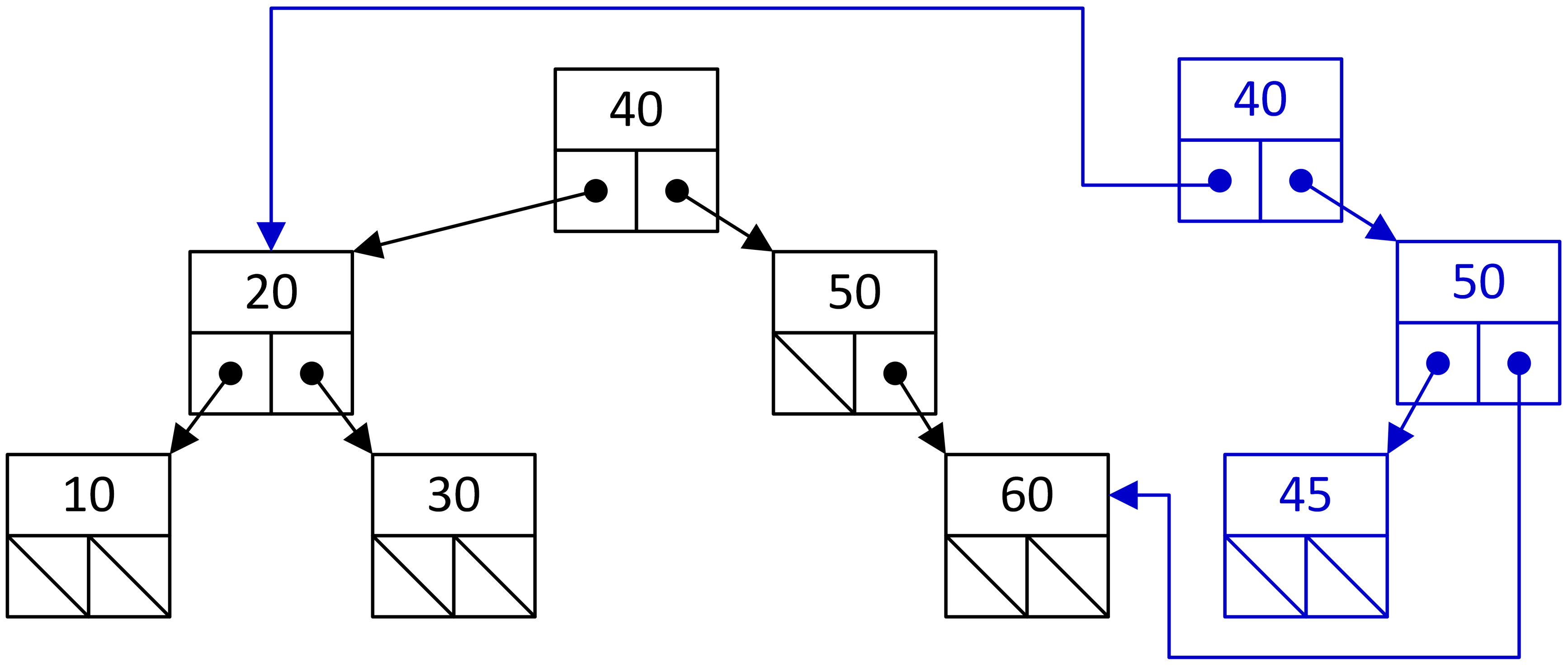
Binary Search Trees (BSTs)

Binary Search Tree (BSTs) enforce the **ordering property**: for every node with an item i, all items in the left child subtree are less than i, and all items in the right child subtree are greater than i.



Mixing paradigms

As with linked lists, we have to be careful not to mix functional and imperative paradigms, especially when adding nodes. The following example visualizes what Racket returns when a node (45) is added to the BST illustrated earlier.



Our *BST node* (`bstnode`) is very similar to our linked list node definition.

```
struct bstnode {
  int item;
  struct bstnode *left;
  struct bstnode *right;
};

struct bst {
  struct bstnode *root;
};
```

In CS 135, BSTs were used as *dictionaries*, with each node storing both a key and a value. Traditionally, a BST only stores a single item, and additional values can be added as *node augmentations* if required.

As with linked lists, we need a function to *create* a new BST.

```
// bst_create() creates a new BST
// effects: allocates memory: call bst_destroy

struct bst *bst_create(void) {
  struct bst *t = malloc(sizeof(struct bst));
  t->root = NULL;
  return t;
}
```

Before writing code to *insert* a new node, first we write a helper to create a new *leaf* node.

```
struct bstnode *new_leaf(int i) {
  struct bstnode *leaf = malloc(sizeof(struct bstnode));
  leaf->item = i;
  leaf->left = NULL;
  leaf->right = NULL;
  return leaf;
}
```

As with lists, we can write tree functions *recursively* or *iteratively*.

For the recursive version, we will need a wrapper, and we can handle the special case that the tree is empty.

```
void bst_insert(int i, struct bst *t) {
  if (t->root) {
    insert_bstnode(i, t->root);
  } else {
    t->root = new_leaf(i);
  }
}
```

For the core function, we recurse on *nodes*.

```
void insert_bstnode(int i, struct bstnode *node) {
  if (i < node->item) {
    if (node->left) {
      insert_bstnode(i, node->left);
    } else {
      node->left = new_leaf(i);
    }
  } else if (i > node->item) {
    if (node->right) {
      insert_bstnode(i, node->right);
    } else {
      node->right = new_leaf(i);
    }
  } // else do nothing, as item already exists
}
```

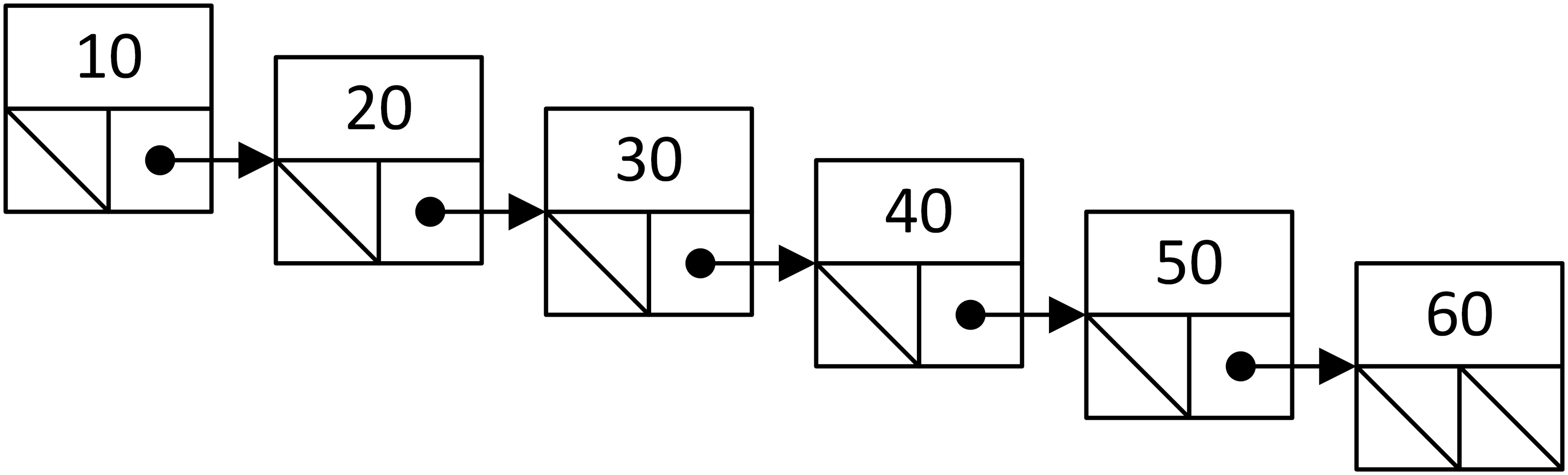
The iterative version is similar to the linked list approach.

```
void bst_insert(int i, struct bst *t) {
  struct bstnode *node = t->root;
  struct bstnode *parent = NULL;
  while (node) {
    if (node->item == i) return;
    parent = node;
    if (i < node->item) {
      node = node->left;
    } else {
      node = node->right;
    }
  }
  if (parent == NULL) {  // tree was empty
    t->root = new_leaf(i);
  } else if (i < parent->item) {
    parent->left = new_leaf(i);
  } else {
    parent->right = new_leaf(i);
  }
}
```

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 *number of nodes* in 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 standard tree functions (*e.g.*, `insert`, `remove`, `search`) are all $O(\log n)$.

With an **unbalanced** tree, the running time of each function is $O(h)$.

A ***self-balancing tree*** "re-arranges" the nodes to ensure that tree is always balanced.

With a good self-balancing implementation, all standard tree functions *preserve the balance of the tree* **and** have an $O(\log n)$ running time.

In CS 240 and CS 341 you will see *self-balancing trees*.

Self-balancing trees often use node augmentations to store extra information to aid the re-balancing.

Count node augmentation

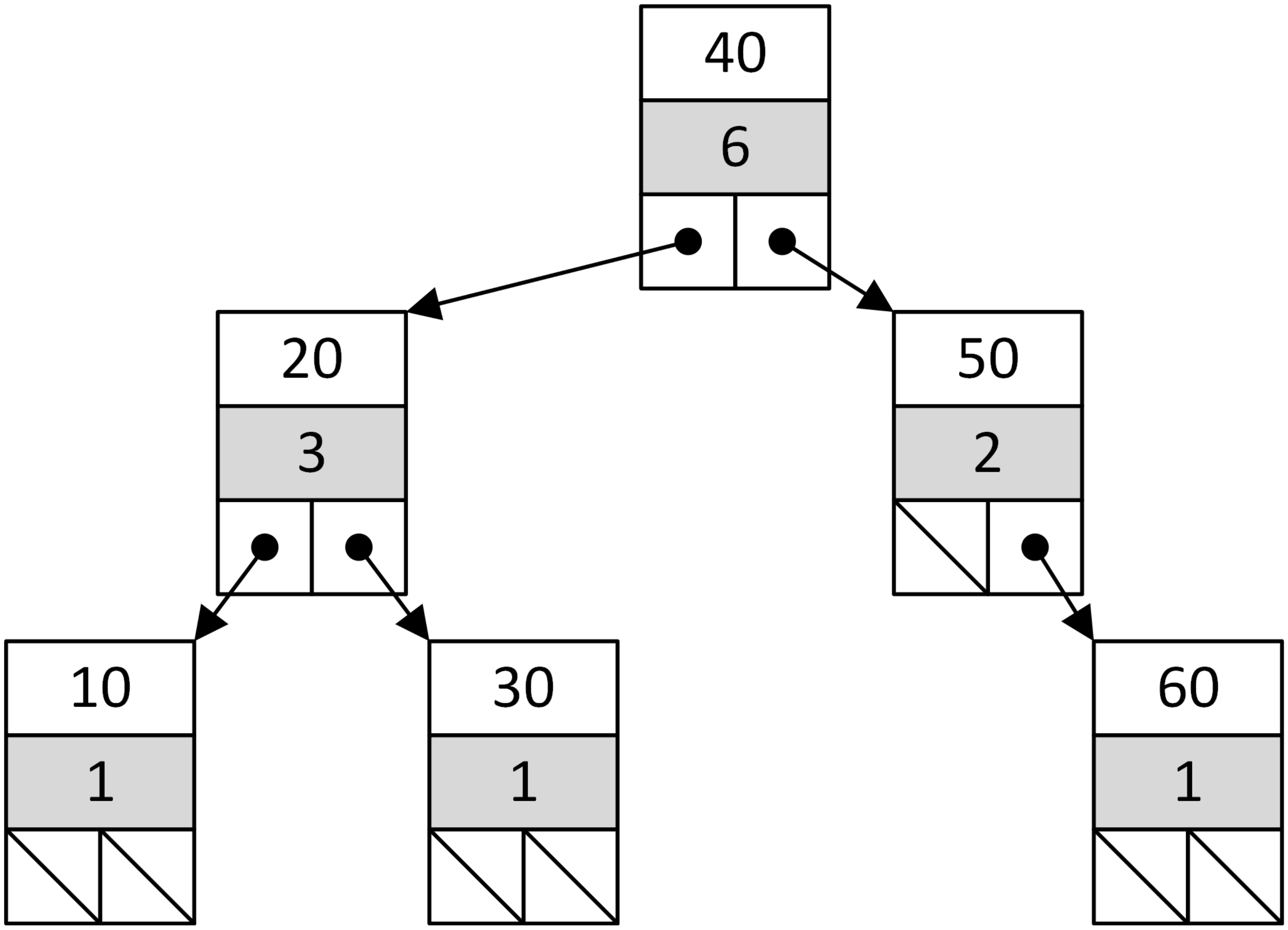
A popular tree **node augmentation** is to store in *each node* the **count** (number of nodes) in its subtree.

```
  struct bstnode {
    int item;
    struct bstnode *left;
    struct bstnode *right;
    int count;                    // *****NEW
  };
```

This augmentation allows us to retrieve the number of nodes in the tree in $O(1)$ time.

It also allows us to implement a `select` function in $O(h)$ time. `select(k)` finds item with index `k` in the tree.

example: count node augmentation



The following code illustrates how to select item with index k in a BST with a count node augmentation.

```
int select_node(int k, struct bstnode *node) {
  assert(node && 0 <= k && k < node->count);
  int left_count = 0;
  if (node->left) left_count = node->left->count;
  if (k < left_count) return select_node(k, node->left);
  if (k == left_count) return node->item;
  return select_node(k - left_count - 1, node->right);
}

int bst_select(int k, struct bst *t) {
  return select_node(k, t->root);
}
```

`select(0, t)` finds the smallest item in the tree.

Array-based trees

For some types of trees, it is possible to use an **array** to store a tree.

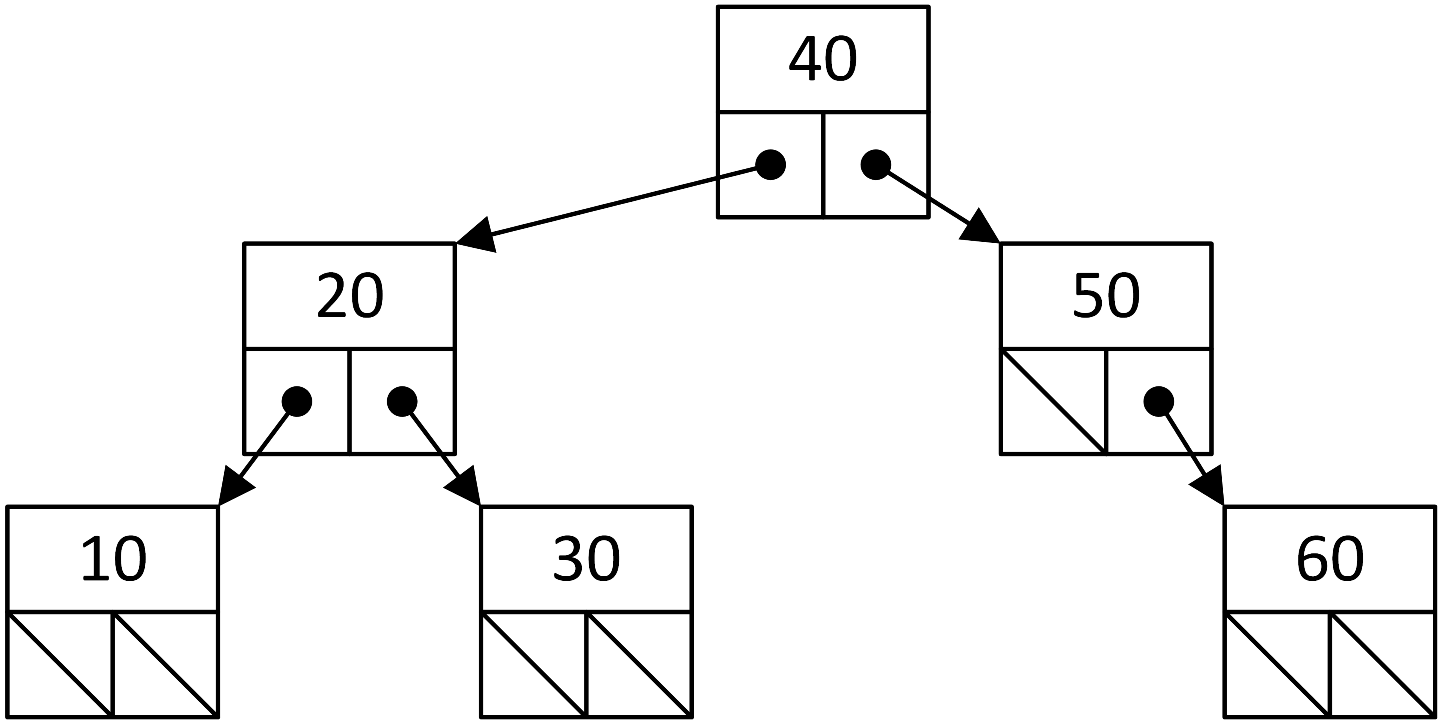
- the root is stored at `a[0]`
- for the node at `a[i]`
  - its `left` is stored at `a[2 * i + 1]`
  - its `right` is stored at `a[2 * i + 2]`
  - its `parent` is stored at `a[(i - 1) / 2]`
- a special *sentinel value* can be used to indicate an empty node
- a tree of height h requires an array of length $2^h - 1$

  (a dynamic array can be `realloc`'d as the tree height grows)

example: array-based tree representation

```
left:  2i+1
right: 2i+2
```

40	20	50	10	30	-	60



Array-based trees are often used to implement “complete trees”, where there are no *empty* nodes, and every level of the tree is filled (except the bottom).

The *heap* data structure (not the section of memory) is often implemented as a complete tree in an array.

For *self-balancing* trees, the self-balancing (*e.g.,* rotations) is often more awkward in the array notation. However, arrays work well with *lazy* rebalancing, where a rebalancing occurs infrequently (*i.e.,* when a large inbalance is detected). The tree can be rebalanced in $O(n)$ time, typically achieving *amortized* $O(\log n)$ operations.

Dictionary ADT (revisited)

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

Typical dictionary ADT operations:

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

In the following example, we implement a Dictionary ADT using a BST data structure.

As in CS 135, we use `int` *keys* and `string` *values*.

```
// dictionary.h

struct dictionary;

struct dictionary *dict_create(void);

void dict_insert(int key, const char *val, struct dictionary *d);

const char *dict_lookup(int key, struct dictionary *d);

void dict_remove(int key, struct dictionary *d);

void dict_destroy(struct dictionary *d);
```

Using the same `bstnode` structure, we *augment* each node by adding an additional `value` field.

```
struct bstnode {
  int item;                     // key
  char *value;                  // additional value (augmentation)
  struct bstnode *left;
  struct bstnode *right;
};

struct dictionary {
  struct bstnode *root;
};

struct dictionary *dict_create(void) {
  struct dictionary *d = malloc(sizeof(struct dictionary));
  d->root = NULL;
  return d;
}
```

When inserting key/value pairs to the dictionary, we make a **copy** of the string passed by the client. When removing nodes, we also `free` the value.

If the client tries to insert a duplicate key, we replace the old value with the new value.

First, we will modify the `new_leaf` function to make a **copy** of the value provided by the client.

```
  struct bstnode *new_leaf(int key, const char *val) {
    struct bstnode *leaf = malloc(sizeof(struct bstnode));
    leaf->item = key;
    leaf->value = my_strdup(val);     // make a copy
    leaf->left = NULL;
    leaf->right = NULL;
    return leaf;
  }
```

And the wrapper is essentially the same:

```
void dict_insert(int key, const char *val, struct dictionary *d) {
  if (d->root) {
    insert_bstnode(key, val, d->root);
  } else {
    d->root = new_leaf(key, val);
  }
}
```

```
void insert_bstnode(int key, const char *val, struct bstnode *node)

  if (key == node->item) { // must replace the old value
    free(node->value);
    node->value = my_strdup(val);

  } else if (key < node->item) { // otherwise, it's the same
    if (node->left) {
      insert_bstnode(key, val, node->left);
    } else {
      node->left = new_leaf(key, val);
    }
  } else if (node->right) {
    insert_bstnode(key, val, node->right);
  } else {
    node->right = new_leaf(key, val);
  }
}
```

This implementation of the `lookup` operation returns `NULL` if unsuccessful.

```
const char *dict_lookup(int key, struct dictionary *d) {
  struct bstnode *node = d->root;
  while (node) {
    if (node->item == key) {
      return node->value;
    }
    if (key < node->item) {
      node = node->left;
    } else {
      node = node->right;
    }
  }
  return NULL;
}
```

There are several different ways of removing a node from a BST.

We implement `remove` with the following strategy:

- A) If the node with the key (“key node”) is a leaf, we remove it.
- B) If one child of the key node is empty (`NULL`), the other child is “promoted” to replace the key node.
- C) Otherwise, we find the node with the *next largest* key (“next node”) in the tree (*i.e.,* the smallest key in the right subtree). We replace the key/value of the key node with the key/value of the next node, and then remove the next node from the right subtree.

```
void dict_remove(int key, struct dictionary *d) {
  d->root = remove_bstnode(key, d->root);
}

struct bstnode *remove_bstnode(int key, struct bstnode *node) {
  // key did not exist:
  if (node == NULL) return NULL;

  // search for the node that contains the key
  if (key < node->item) {
    node->left = remove_bstnode(key, node->left);
  } else if (key > node->item) {
    node->right = remove_bstnode(key, node->right);
  } else if // continued on next page ...
            // (we have now found the key node)
```

If either child is NULL, the node is removed (free'd) and the other child is promoted.

```
  } else if (node->left == NULL) {
    struct bstnode *new_root = node->right;
    free(node->value);
    free(node);
    return new_root;
  } else if (node->right == NULL) {
    struct bstnode *new_root = node->left;
    free(node->value);
    free(node);
    return new_root;
  } else  // continued...
          // (neither child is NULL)
```

Otherwise, we replace the key/value at this node with next largest key/value, and then remove the next key from the right subtree...

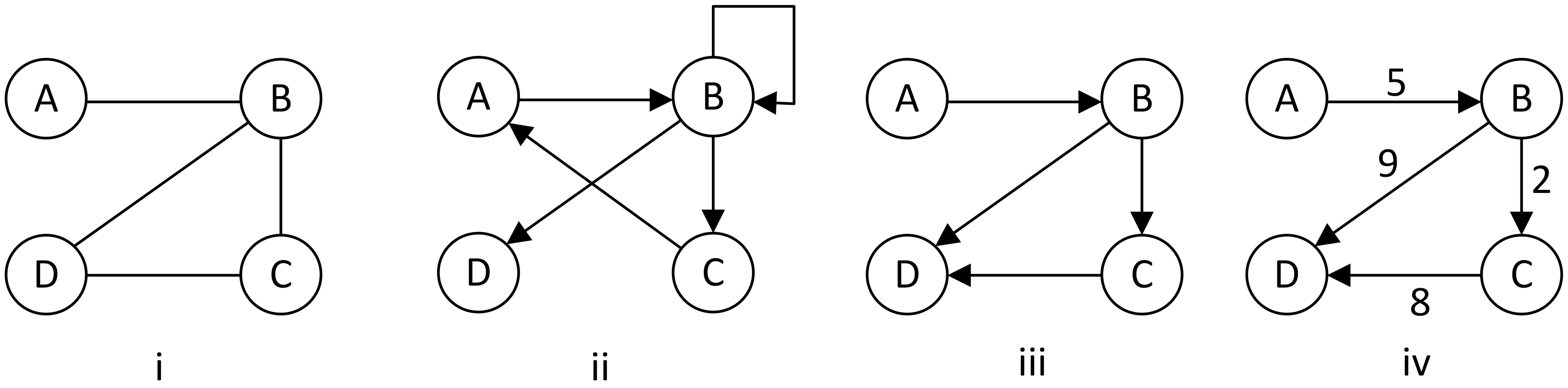
```
  } else {
    // find next largest key and its parent
    struct bstnode *next = node->right;
    struct bstnode *parent_of_next = NULL;
    while (next->left) {
      parent_of_next = next;
      next = next->left;
    }
    // free old value & replace key/value of this node
    free(node->value);
    node->item = next->item;
    node->value = next->value;
    // remove next largest node
    if (parent_of_next) {
      parent_of_next->left = next->right;
    } else {
      node->right = next->right;
    }
    free(next);
  }
  return node;
}
```

Finally, the recursive `destroy` operation `free`s the children and the (string) value before itself.

```
void free_bstnode(struct bstnode *node) {
  if (node) {
    free_bstnode(node->left);
    free_bstnode(node->right);
    free(node->value);
    free(node);
  }
}

void dict_destroy(struct dictionary *d) {
  free_bstnode(d->root);
  free(d);
}
```

Graphs

Linked lists and trees can be thought of as *"special cases"* of a ***graph*** data structure. Graphs are the only core data structure we are **not** working with in this course.



Graphs link **nodes** with **edges**. Graphs may be undirected (i) or directed (ii), allow cycles (ii) or be acyclic (iii), and have labeled edges (iv) or unlabeled edges (iii).

Goals of this Section

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

- use the new linked list and tree terminology introduced
- use linked lists and trees with a recursive or iterative approach
- use wrapper structures and node augmentations to improve efficiency
- explain why an unbalanced tree can affect the efficiency of tree functions

Abstract Data Types (ADTs) & Design

Readings: CP:AMA 19.5, 17.7 (`qsort`)

Selecting a data structure

In Computer Science, every data structure is some **combination** of the following “**core**” data structures.

- primitives (*e.g.*, an `int`)
- structures (*i.e.*, `struct`)
- arrays
- linked lists
- trees
- graphs

Selecting an appropriate data structure is important in **program design**. Consider a situation where you are choosing between an array, a linked list, and a BST. Some design considerations are:

- How frequently will you add items? remove items?
- How frequently will you search for items?
- Do you need to access an item at a specific position?
- Do you need to preserve the "original sequence" of the data, or can it be re-arranged?
- Can you have duplicate items?

Knowing the answers to these questions and the efficiency of each data structure function will help you make design decisions.

Sequenced data

Consider the following strings to be stored in a data structure.

```
"Wei" "Jenny" "Ali"
```

Is the **original sequencing** important?

- If it's the result of a competition, yes: `"Wei"` 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`	$O(1)^*$	$O(1)^\dagger$
`remove_at`	$O(n)$	$O(n)$
`remove_front`	$O(n)$	$O(1)$
`remove_back`	$O(1)$	$O(1)^\diamond$

$*$ amortized

$\dagger$ requires a `back` pointer – $O(n)$ without

$\diamond$ requires a *doubly* linked list and a `back` pointer – $O(n)$ without.

Data structure comparison: unsequenced (sorted) data

Function	Sorted Dynamic Array	Sorted Linked List	Regular BST	Self-Balancing BST
`select`	$O(1)$	$O(n)$	$O(h)^\dagger$	$O(\log n)^\dagger$
`search`	$O(\log n)$	$O(n)$	$O(h)$	$O(\log n)$
`insert`	$O(n)$	$O(n)$	$O(h)$	$O(\log n)$
`remove`	$O(n)$	$O(n)$	$O(h)$	$O(\log n)$

† requires a `count` augmentation – $O(n)$ without.

`select(k)` finds the item with index k in the structure.

For example, `select(0)` finds the smallest element.

example: design decisions

- An array is a good choice if you frequently access elements at specific positions (random access).
- A linked list is a good choice for sequenced data if you frequently add and remove elements at the start.
- A self-balancing BST is a good choice for unsequenced data if you frequently search for, add and remove items.
- A sorted array is a good choice if you rarely add/remove elements, but frequently search for elements and select the data in sorted order.

Implementing collection ADTs

A significant benefit of a collection ADT is that a client can use it “abstractly” without worrying about how it is implemented.

In practice, ADT modules are usually well-written, optimized and have a well documented interface.

In this course, we are interested in how to implement ADTs.

Typically, the collection ADTs are implemented as follows.

- **Stack**: linked lists or dynamic arrays
- **Queue**: linked lists
- **Sequence**: linked lists or dynamic arrays.

  Some libraries provide two different ADTs (*e.g.,* a list and a vector) that provide the same interface but have different operation run-times.
- **Dictionary** (and **Set**s): self-balanced BSTs or hash tables*.

* A hash table is typically an array of linked lists (more on hash tables in CS 240).

Beyond integers

In Section 10, we presented an implementation of a Stack ADT that only supported a stack of `int`egers.

What if we want to have a stack of a different type?

There are three common strategies to solve this “type” problem in C:

- write a separate implementation for each possible item type,
- use a `typedef` to define the item type, or
- use a `void` pointer type (`void *`).

The first option is unwieldy and unsustainable. We first discuss the `typedef` strategy, and then the `void *` strategy.

We don’t have this problem in Racket because of dynamic typing.

This is one reason why Racket and other dynamic typing languages are so popular.

Some statically typed languages have a *template* feature to avoid this problem. For example, in C++ a stack of integers is defined as:

```
  stack<int> my_int_stack ;
```

The stack ADT (called a stack “container”) is built-in to the C++ STL (standard template library).

typedef

The C `typedef` keyword allows you to create your own “type” from previously existing types. This is typically done to improve the readability of the code, or to hide the type (for security or flexibility).

```
typedef int Integer;
typedef int *IntPtr;

Integer i;
IntPtr p = &i;
```

It is common to use a different coding style (we use `CamelCase`) when defining a new “type” with `typedef`.

`typedef` is often used to simplify complex declarations (*e.g.*, function pointer types).

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

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

void array_map(MapFn f, int a[], int len) {  // <- cleaner!
  for (int i = 0; i < len; ++i) {
    a[i] = f(a[i]);
  }
}

int main(void) {
  int arr[6] = {4, 8, 15, 16, 23, 42};
  array_map(add1, arr, 6);
  MapFn f = add1;
  array_map(f, arr, 6);
  //...
}
```

Stack ADT: cleaner interface

```
struct stack;

// use [Stack] instead of [struct stack *]
typedef struct stack *Stack;

// operations:

Stack stack_create(void);

bool stack_is_empty(Stack s);

int stack_top(Stack s);

int stack_pop(Stack s);

void stack_push(int item, Stack s);

void stack_destroy(Stack s);
```

Some programmers consider it poor style to use `typedef` to "abstract" that a type is a *pointer*, as it may accidentally lead to memory leaks.

A compromise is to use a type name that reflects that the type is a pointer (*e.g.*, `StackPtr`).

The Linux kernel programming style guide recommends avoiding `typedefs` altogether.

The “`typedef`” strategy is to define the type of each item (`ItemType`) in a separate header file (`"item.h"`) that can be provided by the client.

```
// item.h
typedef int ItemType;               // for stacks of ints
```

or...

```
// item.h
typedef struct posn ItemType;    // for stacks of posns
```

The ADT module would then be implemented with this `ItemType`.

```
#include "item.h"

void stack_push(Stack S, ItemType i);

ItemType stack_top(Stack s);
```

Having a client-defined `ItemType` is a popular approach for small applications, but it does not support having two different stack types in the same application.

The `typedef` approach can also be problematic if `ItemType` is a pointer type and it is used with dynamic memory. In this case, calling `stack_destroy` may cause 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 stacks**, where each stack is an instance of the stack ADT. If the sequence `remove_at` operation simply calls `free` on the item, it causes a memory leak as the stack data is not freed.

To solve this problem, the client can provide a customized `free` function for the ADT to call (*e.g.*, `stack_destroy`).

example: stack interface with void pointers

```
// (partial interface) CLIENT'S RESPONSIBILITY TO FREE ITEMS

// stack_push(s, i) puts item i on top of the stack
//   NOTE: The caller should not free the item until it is popped
void stack_push(Stack s, void *i);

// stack_top(s) returns the top but does not pop it
//   NOTE: The caller should not free the item until it is popped
const void *stack_top(Stack s);

// stack_pop(s) removes the top item and returns it
//   NOTE: The caller is responsible for freeing the item
void *stack_pop(Stack s);

// stack_destroy(s) destroys the stack
// requires: The stack must be empty (all items popped)
void stack_destroy(Stack s);
```

example: client interface

```
#include "stack.h"

// this program reverses the characters typed
int main(void) {
  Stack s = stack_create();
  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);
  }
  stack_destroy(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.

The `return` value of a comparison function `f(a, b)` follows the `strcmp(a, b)` convention:

- negative: a precedes b
- zero: a is equivalent to b
- positive: a follows b

```
// a comparison function for integers
int compare_ints(const void *a, const void *b) {
  const int *ia = a;
  const int *ib = b;
  return *ia - *ib;
}
```

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 dict_create(DictKeyCompare f);

// lookup key k in Dictionary d
const void *dict_lookup(Dictionary d, void *k);
```

```
// dictionary.c (partial implementation)

struct bstnode {
  void *item;                   // key
  void *value;                  // additional value (augmentation)
  struct bstnode *left;
  struct bstnode *right;
};

struct dictionary {
  struct bstnode *root;
  DictKeyCompare key_compare;   // function pointer
};


Dictionary dict_create(DictKeyCompare f) {
  Dictionary d = malloc(sizeof(struct dictionary));
  d->root = NULL;
  d->key_compare = f;
  return d;
}
```

This implementation of `dict_lookup` illustrates how the comparison function would work.

```
const void *dict_lookup(void *key, Dictionary d) {
  struct bstnode *node = d->root;
  while (node) {
    int result = d->key_compare(key, node->item);
    if (result == 0) {
      return node->value;
    }
    if (result < 0) {
      node = node->left;
    } else {
      node = node->right;
    }
  }
  return NULL;
}
```

C generic algorithms

Now that we are comfortable with `void` pointers, we can use C's built-in `qsort` function.

`qsort` is part of `<stdlib.h>` and can sort an array of any type.

This is known as a "generic" algorithm.

`qsort` requires a comparison function (pointer) that is used identically to the comparison approach we described for ADTs.

```
void qsort(void *arr, int len, size_t size,
           CompFuncPtr f);
```

The other parameters of `qsort` are an array of any type, the length of the array (number of elements), and the `sizeof` each element.

example: qsort

```
// see previous definition
int compare_ints (const void *a, const void *b);

int main(void) {

  int a[7] = {8, 6, 7, 5, 3, 0, 9};

  qsort(a, 7, sizeof(int), compare_ints);

  //...
}
```

C also provides a generic binary search (`bsearch`) function that searches any sorted array for a key, and either returns a pointer to the element if found, or `NULL` if not found.

```
void *bsearch(void *key,
              void *arr,
              int len,
              size_t size,
              CompFuncPtr f);
```

Goals of this Section

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

- determine an appropriate data structure or ADT for a given design problem
- describe the memory management issues related to using `void` pointers in ADTs and how `void` pointer comparison functions can be used with generic ADTs and generic algorithms

Beyond this course

Readings: CP:AMA 2.1, 15.4

Machine code

In Section 04 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 02).

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 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 provide 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) returns 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`** changes 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 is passed arguments.

In Section 02 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 arguments 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

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 (reads input) and a source (prints 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 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 “running”: execute while reading `stdio` from seashell
- if “testing”: for each `.in` file, execute the program redirecting from the `.in` file to an output file:

  ```
  $ ./program < mytest.in > mytest.out
  ```

  Next, use a comparison program to compare the output files to the `.expect` files and display the differences

  ```
  $ diff mytest.out mytest.expect
  ```

Full C language

We have skipped many C language features, including:

- `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!