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 2020)

Instructors: Tim Brecht, Navid Esfahani, Daniel Holtby, Adrian Reetz, Dave Tompkins

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.

\[
\text{CS 135} \implies \text{CS 136} \implies \text{CS 246}
\]

\text{functional} \quad \text{imperative} \quad \text{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 W Store | Course Materials + Supplies in SCH.
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.
In some terms, the course notes may be supplemented by online **appendices** that contain additional content and examples.

You are not responsible for any material that appears in appendices.
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)* test results are available immediately and ensure your program is “runnable”.

- **Private** *(comprehensive / correctness)* test results 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*, we mark the submission with the highest score that was submitted closest to the deadline.

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

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

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

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

The design recipe has two main goals:

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

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

In this course, every function you write must have:

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

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

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

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

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

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

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

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

View your formative feedback on MarkUs.
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) and be easily translatable into “machine code” (Section 13).

Thousands of popular programs and portions of all of the popular operating systems are written in C.
C versions

There are a few different versions of the C standard.

In this course, we use the C99 standard (from 1999).

The C11 standard (2011) added some new features to the language, but those features are not needed in this course.

The C18 standard (2018) only fixes a few bugs from C11.

The C2x standard (202?) is currently in development.
Introductory C

In this section we learn to write simple functions and programs 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 topics not yet covered (e.g., loops).
Comments

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

Any text between /* and */ is also a comment.

/* ... */ can extend over multiple lines and can comment out large sections of code.

// C comment (one-line only)

/* This is a
   multi-line comment */

C’s multi-line comment cannot be “nested”:
/* this /* nested comment is an */ error */
Expressions

C expressions use traditional *infix* algebraic notation: (e.g., 3 + 3).

Use parentheses to specify the **order of operations**
(normal arithmetic rules apply).

\[
\begin{align*}
1 + 3 \times 2 & \Rightarrow 7 \\
(1 + 3) \times 2 & \Rightarrow 8
\end{align*}
\]

Racket uses *prefix* notation: (+ 3 3)

Languages that use prefix (or postfix) notation do not require parenthesis to specify the order of operations.
Operators

In addition to the traditional mathematical operators (e.g., +, -, *), C also has non-mathematical operators (e.g., data operators).

With over 40 operators in total, the order of operations is complicated (see CP:AMA Appendix A).

C does not have an exponentiation operator (e.g., $x^n$).

Confusingly, the “bitwise exclusive or” operator (^) looks like an exponentiation operator. Bitwise operators are beyond the scope of this course.
In C, each operator is either *left* or *right* associative to further clarify any ambiguity (see CP:AMA 4.1).

The multiplication operators are *left*-associative:

\[ 4 \times 5 / 2 \] is equivalent to \[ (4 \times 5) / 2 \].

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.

\[(4 \times 5) / 2 \Rightarrow 10\]

\[4 \times (5 / 2) \Rightarrow 8\]

\[-5 / 2 \Rightarrow -2\]

Remember, use parentheses to clarify the order of operations.

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

The C modulo operator (\%) produces the remainder after integer division.

9 \% 2 \Rightarrow 1
9 \% 3 \Rightarrow 0
9 \% 5 \Rightarrow 4

The value of \((a \% b)\) is equal to: \(a - (a / b) \times b\).

It is often best to avoid using % with negative integers.

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

Every function, variable and structure requires an identifier (or “name”).

C identifiers must start with a letter, and can only contain letters, underscores and numbers.

In this course, use underscore_style (or snake case) for identifiers with compound words.

For example: hst_rate, trace_int, quick_sort

Use underscore_style in your code.
underscore style is the most popular style for C projects.

In other languages (e.g., Java) camelCaseStyle is popular.

In practice, it is important to use the recommended style for the language and/or follow the project (or corporate) style guide.

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.
Anatomy of a function definition

```c
int my_add(int a, int b) {
    return a + b;
}
```

- braces (```{}```) indicate the beginning/end of a function **block**
- `return` keyword, followed by an expression, followed by a semicolon (`;`)
- parameters (a, b) are separated by a comma
- the function and parameter **types** are specified (i.e., `int`)

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

C uses a static type system: all types must be known before the program is run and the type of an identifier cannot change.

For now, we will only use C integers (more types in Section 04).

```c
int my_add(int a, int b) {
    return a + b;
}
```

The return type of my_add is an int (appears before my_add).

The parameters a and b are also both ints.

Racket uses a dynamic type system.
If the type in a function definition is omitted:

```c
int my_add(int a, int b) { // properly typed
    return a + b;
}
```

```c
bad_add(a, b) { // missing types
    return a + b;
}
```

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

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

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

```racket
(my-add "hello" 3) ; Racket runtime error
```

In C, it is impossible to violate the contract type.

“Type” runtime errors do not exist.

```c
my_add("hello", 3)    // does not run in C
```
Function terminology

We *call* a function by *passing* it *arguments*.

A function *returns* a value.

```
my_add(1, 2) ⇒ 3
```

We *call* *my_add* and *pass* it the *arguments* 1 and 2.

*my_add*(1, 2) *returns* 3.

In “functional” language terminology (e.g., Racket) we *apply* a function, which *consumes* arguments and it *produces* a value.
Functions without parameters

Use the void keyword to indicate a function has no parameters.

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

To call a parameterless function, put nothing between the parentheses (do not pass void).

```c
my_num() ⇒ 42
```
If the `void` is omitted in a parameterless function definition **definition:**

```c
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).

**Always use void** to clearly communicate (and enforce) that there are no parameters.

```c
int my_num(void) {
    // ...
}
```
No nested functions

In C, functions cannot be “nested” (defined) inside of another function (a.k.a. local functions).

```c
int outer(int i) {
    int inner(int j) { // INVALID
        // ...
    }
    // ...
}
```

The GNU C environment (gcc) has introduced a language extension for nested C functions, but it is not part of the C standard.
Function documentation

Provide a purpose for every function that shows an example of it being called, followed by a brief description of what the function does (not how it does it).

No contract types are necessary (they are part of the definition).

Add a requires comment if appropriate.

```c
// my_divide(x, y) evaluates x/y using integer division
// 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 a, int b) {
    return a + b;
}

int my_add(int a, int b) { return a + b; } // BAD

int my_add(int a, int b) { return a + b; } // RIDICULOUSLY BAD

Follow the course style. The course staff and markers may not follow your code if it is poorly formatted.
CS 136 style

```c
int my_add(int a, int b) {
    return a + b;
}
```

- 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 there are a large number of parameters, a large expression or a long purpose, continue (indented) on the following line.

```c
// my_super_long_function(a, b, c, d, e, f, g) does some // amazing things with those parameters...

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

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

In many interpreted languages (including Racket), the entry point is simply the **top** of the file being “run”.
main

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

```c
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 `errno`).

\(^\dagger\) main has optional parameters (discussed in Section 13).

A successful program returns **zero** (no error code).
main is a special function and does not require an explicit return value. The default value is success (zero).

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

Unless an assignment has special instructions, your main function should **never** return a non-zero value, as it causes your marmoset tests to fail.

There is no consensus on whether requiring a return 0; in main is “good style”. In this course, we do not require it.
Top-level expressions

In C, *top-level expressions* (code outside of a function) are **not** allowed.

Code only executes **inside** of a function.

```
1 + 1;       // INVALID

int my_add(int a, int b) {
    return a + b;
}

my_add(1, 2); // INVALID
```
In DrRacket, the final values of *top-level expressions* are displayed in the “interactions window”.

`;; my racket program

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

(define (my-add a b)
  (+ a b))

(my-add 1 2) ;; <-- top level

2

3
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):

```c
int main(void) {
    trace_int(1 + 1);
    trace_int(my_add(1, 2));
}
```

1 + 1 => 2
my_add(1, 2) => 3

Leave your *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 (documentation omitted)

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

int my_add(int a, int b) {
    return a + b;
}

int main(void) {
    trace_int(1 + 1);
    trace_int(my_add(1, 2));
}

Note the necessary #include line at the top of the program.
For now, always add this line (it is explained in Section 06).
Function ordering

If the two functions are re-ordered:

```c
int main(void) {
    trace_int(1 + 1);
    trace_int(my_add(1, 2));
}

int my_add(int a, int b) { // now below main
    return a + b;
}
```

There is an error (such as):

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

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

Document a program (state its purpose) at the top of the file (not necessarily where main is defined).

    // This program informally tests the my_add function
    
    #include "cs136.h"
    
    // my_add(a, b) calculates the sum of a and b
    
    int my_add(int a, int b) {
        return a + b;
    }
    
    int main(void) {
        trace_int(1 + 1);
        trace_int(my_add(1, 2));
    }

There is no need to add additional documentation for main itself.
Testing

Our *tracing tools* are an excellent way for you to interact with your code and help you “see” what is happening.

They are helpful to *informally* test your code.

They are not a viable strategy to *thoroughly* test.

However, to facilitate more thorough testing, we will need Boolean expressions...
Boolean expressions

In C, Boolean expressions do not produce “true” or “false”.

They produce either:

- zero (0) for “false”, or
- one (1) for “true”.

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

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)

The operators `<`, `<=`, `>` and `>=` behave exactly as expected.

(2 < 3) ⇒ 1 (true)
(2 >= 3) ⇒ 0 (false)

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.
Logical Operators

The Logical operators are: ! (not), && (and), || (or):

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

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

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

does not generate an error if a is 0.

A common mistake is to use a single & or | instead of && or ||.
All non-zero values are true

Operators that produce a Boolean value (e.g., ==) will always produce 0 or 1.

Operators (or functions) that expect a Boolean value (e.g., &&) will consider any non-zero value to be “true”.

Only zero (0) is “false”†.

(2 && 3) ⇒ 1
(0 || -2) ⇒ 1
!5 ⇒ 0

† The value NULL (Section 05) is also considered false.
You are not expected to “memorize” the order of operations. When in doubt (or to add clarity) **add parentheses**.

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>!</td>
</tr>
<tr>
<td>multiplicative</td>
<td>* / %</td>
</tr>
<tr>
<td>additive</td>
<td>+ -</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; &lt;= &gt;= &gt;</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
</tbody>
</table>
bool type

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

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

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

Use the assert function to test functions:

```c
assert(my_add(1, 2) == 3);
```

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

If exp is true (non-zero), it does “nothing” and continues to the next line of code.

assert is very similar to Racket’s check-expect:

```racket
(check-expect (my-add 1 2) 3)
```
// My second C program (now with better testing!)

#include "cs136.h"

int my_add(int a, int b) {
    return a + b;
}

int main(void) {
    assert(my_add(0, 0) == 0);
    assert(my_add(1, 1) == 2);
    assert(my_add(-2, 1) == -1);
}

We discuss additional testing methods later. For now, test your code with asserts in your main function as above.
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.
Function requirements

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

```c
// 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;
}
```

In the slides, we often omit asserts to save space.

assert any feasible function requirements.
Infeasible requirements

Some requirements are infeasible to assert, or (as we will discuss in Section 08) they would be *inefficient* to assert.

It is good style to communicate (*i.e.*, document) that a requirement is not asserted:

```c
// my_function(n) ....
// requires: n is a prime number [not asserted]
```
Multiple requirements

With multiple requirements, it is better to have several small asserts.

It makes it easier to determine which assertion failed (which requirement was not met).

```
// 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)); // OK
    assert(x > 0); // BETTER
    assert(y < z);
    //...
}
```
Statements

Blocks ({} ) can contain multiple statements:

```c
int my_divide(int x, int y) {
    assert(y);    // statement
    trace_int(y);  // statement
    return x / y;  // statement
    trace_int(x);  // unreachable statement
}
```

Statements are executed in sequence (one after the other).

The return statement ends the function.

In the above code, `trace_int(x);` will never execute.

We define statements more precisely in Section 03.
Brief introduction to control flow

The return statement is a special kind of statement known as a control flow statement.

return “controls the flow” of the program by ending the function and returning to the caller.

We explore control flow statements in Section 04, but first we will introduce one more control flow statement...
Conditionals

The if control flow statement allows us to have functions with conditional behaviour.

```c
int my_abs(int n) {
    if (n < 0) { // note: the () are required
        return -n;
    } else {
        return n;
    }
}
```

There can be more than one return in a function, but only one value is ever returned.

The function stops when the first return is executed.
example: recursion in C

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

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

If there are more than two possible results, use else if.

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

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

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

```c
int my_function(...) {
  if (q1) {
    return a1;
  } else if (q2) {
    return a2;
  } else {
    return a3;
  }
}
```
C’s if *statement* does not produce a value: it only controls the “flow of execution” and cannot be used *inside* of an expression.

We revisit if in Section 04 after we understand how “statements” differ from expressions. For now, only use if as we have demonstrated.

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. Unlike if, cond does produces a value and can be used inside of an expression:

```
(+ y (cond [(< x 0) -x] [else x]))
```
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.
When working with integer values in C, do not add a leading (preceeding) 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:

\[
\begin{align*}
\text{trace\_int(17)}; & \quad 17 \Rightarrow 17 \\
\text{trace\_int(017)}; & \quad 017 \Rightarrow 15 \\
\text{trace\_int(my\_add(010, 010))}; & \quad \text{my\_add(010, 010)} \Rightarrow 16
\end{align*}
\]

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.
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

• Write a simple function in C

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

In CS 135 we used the **functional programming paradigm**:

- functions are “pure” (a.k.a. “mathematical”):
  - functions **only return values**
  - return values **only depend** on argument values

- only **constants** are used

In the **imperative programming paradigm** functions may be “impure” and we will use variables.

A programming **paradigm** can also be thought of as a programming “approach”, “philosophy” or “style”.
**Compound statement**

In English, an imperative is an instruction: “Give me your money!”

In imperative programming, a sequence of instructions (or “statements”) are executed. We have already seen this:

```c
int main(void) {
    trace_int(1 + 1); // do this first
    assert(3 > 2); // then do this
    return 0; // and then do this
}
```

A block `{}` is formally known as a **compound statement**, which is simply a sequence of statements† (to be executed in order).

† Blocks can also contain local variable definitions.
Imperative programming

A program is mostly a sequence of statements to be executed.

*Control flow* is used to change the order of statements (Section 04).

The most significant difference between the functional and imperative paradigms is:

**Imperative programming uses *side effects***

(functional programming does not).

In this section we explore *side effects*, beginning with *output*.
I/O

I/O (Input/Output) is the term used to describe how programs interact with the “real world”.

A program (“app”) on your phone may interact with you in many different ways:

**Input:** touch screen (onscreen keyboard), voice, camera

**Output:** screen (display), sounds, vibrations

It may also interact with non-human entities:
files, printers, GPS, other computers on the internet

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

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

// My first program with I/O

#include "cs136.h"

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

Hello, World

printf is different than the tracing tools we use to debug and informally test our code (more on this distinction later).
int main(void) {
    printf("Hello, World");
    printf("C is fun!");
}
Hello, WorldC is fun!

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

```c
printf("Hello, World\n");
printf("C is\nfun!\n");
Hello, World
C is
fun!
```
The first parameter of printf is a "string".

To output values, use a \textit{format specifier} (the \texttt{f} in \texttt{printf}) within the string and provide an \textbf{additional argument}.

For an integer in “decimal format” the format specifier is "\texttt{%d}".

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

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

In the output, the format specifier is \textbf{replaced} by the additional argument value.

\textbf{Strings are introduced in Section 09.}
There can be multiple format specifiers, each requiring an additional argument.

```c
printf("%d plus %d is: %d\n", 2, 10 / 5, 2 + 2);
2 plus 2 is: 4
```

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

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

I am 100% sure you should watch your spacing!
```

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 format specifier syntax as C.

The full C printf format specifier syntax controls the format and alignment of output.

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

See CP:AMA 22.3 for more details.

In this course, simple "%d" formatting is usually sufficient.
Functions with side effects

Consider the two functions below:

```c
int sqr(int n) {
    return n * n;
}

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

Both functions return the same value.

However, `noisy_sqr` does more than return a value. 

In addition to returning a value, it also produces output.

`noisy_sqr` has a side effect.
Side effects and state (introduction)

In general, a programming side effect is when the state of something “changes”.

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

Consider the following “real world” example: You have a blank piece of paper, and then you write your name on that paper.

You have changed the state of that paper: at one moment it was blank, and in the next it was “autographed”.

In other words, the side effect of writing your name was that you changed the state of the paper.
Documenting side effects

The `printf` function has a side effect: it changes the output (or “display”).

By calling `printf`, the function `noisy_sqr` also has a side effect.

Clearly communicate if a function has a side effect.

```
// noisy_sqr(n) computes n^2
// effects: produces output

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

Add an **effects**: section to document any side effects.
If a side effect occurs conditionally, add the word “may”:

// noisy_abs(n) computes |n|
// effects: may produce output

int noisy_abs(int n) {
    if (n < 0) {
        printf("Yo! I'm changin' the sign!\n");
        return -n;
    } else {
        return n;
    }
}

In this course, there is no need to provide any detail in the “effects:” section ( // effects: produces output is sufficient).

Occasionally, you may want to describe the output in the purpose (e.g., understanding the output is essential to understanding the core behaviour of the function).
Debugging tools

Statements used for debugging and informal testing (e.g., `assert`, `trace_int`) are **not** considered side effects.

Do not add asserts or tracing tools to the “effects:” section.

Leave your tracing code in your assignments: they will not affect your test results.

---

Large software projects often have thousands of assert and tracing statements to aid debugging.

They are usually disabled (or “turned off”) when a project is finalized to improve performance, but they remain in the code.
I/O terminology

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

```c
int sqr(int n) {
    return n * n;
}
```

Informally, someone might say:

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

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

Instead, say:

“if 7 is **passed** to `sqr`, it **returns** 49”.

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

For noisy_sqr, say:
“if 7 is passed 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 carefully.
Testing I/O

asserts can **test** that `sqr` and `noisy_sqr` **return** the correct values:

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

But **assert** **cannot** be used to **test** that `noisy_sqr` produces the **correct output**.

We will need a new method to test output...
In Seashell there are two ways of “running” a 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

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

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

test1.expect:

```
Yo! I'm squaring -3!
Yo! I'm squaring 7!
```

A blank test1.in would also be necessary (I/O tests always require an .in file).
Our tracing tools print to a different output stream than printf (like writing on two different pieces of paper).

By default, printf outputs to the stdout (standard output) stream.

Our tracing tools output to the stderr (standard error) 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 you [RUN] your code, the two streams may appear mixed together in the screen (console) output.
void functions

noisy_sqr has a side effect and returns a value.

A function may only have a side effect, and no return a value.

The void keyword is used to indicate a function returns “nothing”.

    // display_score(score, max) displays the player score
    // effects: produces output

    void display_score(int score, int max) {
        printf("your score is %d out of %d\.\n", score, max);
        return;  // optional
    }

In a void function, the return is optional and has no expression (when the end a void function is reached, it returns automatically).
**printf return value**

Surprisingly, `printf` is **not** a void function (it returns an int).

`printf` returns the number of characters printed.

`printf("hello!\n")` returns 7 (`\n` is a single character).

In the following code, where does the 7 “go”?

```c
int main(void) {
    printf("hello!\n");
}
```

Let’s revisit what a **statement** is...
Expression statements

The following expressions all have a value of 11:

\[
\begin{align*}
11 \\
10 + 1 \\
\text{sqr}(6) - \text{sqr}(5) \\
\text{printf}("\text{expression}\n") \\
\text{printf}("\text{five}\n") + 6
\end{align*}
\]

An **expression statement** is an expression with a semicolon (;).

```c
int main(void) {
    11;
    10 + 1;
    \text{sqr}(6) - \text{sqr}(5);
    \text{printf}("\text{expression}\n")
    \text{printf}("\text{five}\n") + 6;
}
```

What happens to all of those elevens?
The value of an expression statement is **discarded** after it is executed.

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

Imperative programming is “programming by side effects”.

Seashell will give a warning if an expression statement *obviously* has no side effect: `8 + 1;`.

If an expression contains a function call there is no warning because side effects are “assumed” (even if there are none).
Statements

There are only three types of C statements:

• **compound statements (blocks) {}**
  a sequence of statements (to be executed in order)

• **expression statements**
  for generating side effects (values are discarded)

• **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.
More side effects

In addition to:

- Output (*e.g.*, `printf`)

we will encounter two more types of side effects (in this section):

- Input

- Mutation (modifying variables)

All side effects involve *changing state*. 
Variables

*Variables* store values.

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

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

```c
int my_variable = 7;    // definition
```

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

*Definitions are not statements.*

They are different “things” (syntactic units).
Mutation

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

```c
int main(void) {
    int m = 5; // definition (with initialization)
    trace_int(m);
    m = 6;    // mutation!
    trace_int(m);
    m = -1;   // more mutation!
    trace_int(m);
}
```

m => 5
m => 6
m => -1
At every moment in time, a variable must have a value.

When mutation occurs, the “state” (value) of the variable changes.

Mutation is a side effect.

For a “real world” example, consider your bank account balance.

At any moment in time, your account balance has a specific value (or “state”).

When you withdraw money from your account, the balance changes to a new value.

Withdrawing money has a side effect: it changes the value your bank balance.
For most imperative programmers, mutation is second nature and not given a special name (they rarely use the term “mutation”). The word “mutation” does not appear in the CP:AMA textbook.
Assignment Operator

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

\[ m = m + 1; \]

- The “right hand side” (RHS) must be an expression that produces a value with the same type as the LHS.
- The LHS must be the name of a variable (for now).
- The LHS variable is changed (mutated) to store the value of the RHS expression. In other words, the RHS value is assigned to the variable.
- This is a side effect: the state of the variable has changed.
The use of the equal sign (=) can be misleading.

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 the assignment more obvious.
In addition to the mutation side effect, the assignment operator (=) also produces the right hand side value.

This is occasionally used to perform multiple assignments.

```c
x = y = z = 0; // (x = (y = (z = 0)));
```

Avoid having more than one side effect per expression statement.

```c
printf("%d\n", y = 5); // never do this!
printf("%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).

```c
if (i = 13) {
    printf("disaster!\n");
}
```

(i = 13) assigns 13 to i and produces the value 13, so the if expression is always true, and it always prints disaster!

**Pro Tip:** some defensive programmers get in the habit of writing (13 == i) instead of (i == 13). This causes an error if they accidentally use a single =.
Initialization

C allows a variable definition *without* initialization, but it is bad style.

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

Always initialize variables.

In Section 04 we discuss the behaviour of uninitialized variables.
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.

This distinction is especially important in C++.
C allows you to define more than one variable at once.

    int x = 0, y = 2, z = 3;

Most modern style guides discourage this (bad style).

In the following example, x is *uninitialized*.

    int x, y = 0;
More assignment operators

The *compound* addition assignment operator (+=) combines the addition and assignment operator (for convenience).

```c
x += 2; // x = x + 2;
```

Additional compound operators include: -=, *=, /=, %=.

There are also *increment* and *decrement* operators that increase or decrease a variable by one (either *prefix* or *postfix*).

```c
++x; x++; // x += 1;
--x; x--; // x -= 1;
```

If you follow our "one side effect per expression" rule it does not matter if you use prefix or postfix (see next slide).
The language C++ is a pun: one bigger (better) than C.

The *prefix* (;++x) and *postfix* (x++) increment operators 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

Prefix (;++x) is usually preferred to improve clarity and efficiency.
Constants

A \textit{constant} is a “variable” that is \textit{immutable} (not mutable).

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

\begin{verbatim}
const int my_constant = 42;
\end{verbatim}

To define a C \textit{constant}, we add the \texttt{const} keyword to the type.

\begin{quote}
In this course, the term “\textit{variable}” is used for both variable and constant identifiers.

In the few instances where the difference matters, we use the terms “\textit{mutable variables}” and “\textit{constants}”.
\end{quote}
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.
Global and local variables

Variables are either *global* or *local*.

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

*Local* variables are defined *inside* of functions.

```c
int my_global_variable = 7;

void f(void) {
    int my_local_variable = 11;
    //...
}
```
Variable Scope

The *scope* of a variable is the region of code where it is “accessible” or “visible”.

For global variables, the scope is anywhere *below* its definition.

```
int g = 7;

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

We will revisit global scope in Section 06.
Block (local) scope

Local variables have **block scope**. Their *scope* extends from their definition to the *end of the block* they are defined in.

```c
void f(int n) {
    if (n > 0) {
        int b = 19;
        // b IN scope
        // ...
    }
    // b OUT of scope
    // ...
}
```
Variables with the same name can \textit{shadow} other variables from outer scopes, but this is obviously poor style.

The following code defines three \textbf{different} variables named n.

\begin{verbatim}
int n = 1;

int main(void) {
    trace_int(n); \hfill // n => 1
    int n = 2;
    trace_int(n); \hfill // n => 2
    {
        int n = 3;
        trace_int(n); \hfill // n => 3
    }
    trace_int(n); \hfill // n => 2
}
\end{verbatim}
In older versions of C, all the local variable definitions had to be at the start of the function block (before any statements).

In C99, you may define a local variable anywhere in a block.

Modern programming guides recommend that you define a variable:

- in the narrowest scope possible
- as close to its first use as possible

This improves readability and ensures that when a variable is first used its type and initial value are accessible.
“Impure” functions

Recall that the functional paradigm requires “pure” functions:

- functions only return values (no side effects)
- return values only only depend on argument values

For example, the noisy_sqr function is “impure” because it has a side effect (produces output).

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

“Impure” functions are sometimes called “procedures” or “routines” to distinguish their behaviour from “pure” functions.
Mutating global variables

A function that mutates a global variable has a *mutation side effect* (which makes it “impure”).

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

// increment() returns the number of times it has been called
// effects: modifies counter
int increment(void) {
    counter += 1;
    return counter;
}

int main(void) {
    assert(increment() == 1);
    assert(increment() == 2);
}
```

Document any functions with mutation side effects.
Mutating local variables

Mutating a local variable does not give a function a side effect.

It does not affect state outside of the function (global state).

```c
int add1(int n) {
    int k = 0;
    k += 1;
    return n + k;
}

int main(void) {
    assert(add1(3) == 4);
}
```

The statement “k += 1;” has a side effect (mutation), but it only affects state inside of the function (local state).

add1 has no side effects and is still a “pure” function.
Mutating parameters

Parameters are nearly *indistinguishable* from local variables, and can also be mutated.

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

int main(void) {
    int j = 3;
    assert(add1(j) == 4);
    assert(add1(j) == 4);
}
```

This version of `add1` is also a “pure” function (no side effects).

We model how parameters work in Section 04.
Global dependency

A function that depends on a global mutable variable is “impure” even if it has no side effects.

A “pure” function only depends on its argument values.

```c
int n = 10;

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

int main(void) {
    assert(addn(5) == 15);
    n = 100;
    assert(addn(5) == 105);
}
```
Avoiding global mutable variables

Global *mutable* variables are almost always poor style and should be avoided.

Unless otherwise specified, you are not allowed to use global *mutable* variables on your assignments.

On the other hand, global *constants* are great style and strongly encouraged.

This topic is revisited in Section 05 in more detail.
Static local variables have the scope of a local variable, but the duration of a global variable (discussed in Section 04). Their value persists between function calls.

```c
int increment(void) {
    static int counter = 0;
    counter += 1;
    return counter;
}
```

Like global mutable variables, they are almost always poor style and should be avoided.

They are not allowed in this course.
Text input

Earlier, we learned how to output text with printf.

We will now learn how to input text.

The converse of printf is scanf, but we are not quite ready to use it (we introduce scanf in Section 05).

For now, we will use an alternative method for reading input...
read helper functions

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

```c
// 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
```

Note how the side effect of reading input is documented:

```c
// effects: reads input
```
example: reading input (recursively)

// 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());
}
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]` (End Of File) 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       test1.expect
1             5
2
2
3
4
4
4
5
6
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

Be careful when reading input in our Seashell environment: once you read in a value, it can no longer be read again.

Typically, it is best to store the read value (e.g., returned by read_int) in a variable so it can be referenced multiple times.

For example, consider this incorrect partial implementation:

```c
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.
Testing harness

To summarize our function testing strategies:

- **return values**: use assertions (e.g., in main)
- **input and output**: [I/O TEST] (.in and .expect files)

There is an *alternate* approach for testing return values.

To test a function \( f \) we can write a *dedicated test function* that reads in argument values from input, passes those values to \( f \), and then prints out the corresponding return values.

This strategy is known as a *testing harness*. 
example: testing harness for sqr

void test_sqr(void) {
    int n = read_int();
    if (n != READ_INT_FAIL) {
        printf("%d\n", sqr(n));
        test_sqr(); // recurse
    }
}

int main(void) {
    test_sqr();
}
This strategy is useful for testing both “pure” and “impure” functions.

On some assignment questions, we may build a testing harness for you (later, you may be expected to build your own).

For this harness, new tests can be added by editing text files.

In the “real world”, this strategy allows non-coders (e.g., “end users) to develop tests.
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

- 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)
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.

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

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

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

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

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

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

```c
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 gotos, *labels* (code locations) are required.

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

Do not use goto in this course.
Looping

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

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

```c
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.

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

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

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

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

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

Looping is very “imperative”. Without mutation (side effects), the while loop condition would not change, causing an “endless loop”.
Loops can be “nested” within each other.

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

```
*****
****
***
**
*
```
Tracing tools

The provided *tracing tools* can be used to help you understand your control flow and “see” what is happening in your program.

This can help you **debug** your code.

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

---

On your assignments, never *printf* any unnecessary output as it may affect your correctness results.

Always use our tracing tools to help debug your code.
```c
int sum(int k) {
    trace_msg("sum called");
    int s = 0;
    trace_msg("loop starting");
    while (k > 0) {
        trace_int(k);
        s += k;
        trace_int(s);
        --k;
    }
    trace_msg("loop ended");
    return s;
}

int main(void) {
    trace_int(sum(3));
}
```

>> "sum called"
>> "loop starting"
>> k => 3
>> s => 3
>> k => 2
>> s => 5
>> k => 1
>> s => 6
>> "loop ended"
>> sum(3) => 6
while errors

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

```c
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 to exit from the middle of a loop.

break immediately terminates the current (innermost) loop.

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

```c
while (1) {
    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.

```c
// 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:

```
ssetup 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

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

is equivalent to

```c
for (i = 100; i >= 0; --i) {
    printf("%d\n", i);
}
```
Statement A;
for (setup; exp; update) {
    Code Block;
}
Statement Z;
Statement A;
for (setup; exp; update) {
    Code Block;
}
Statement Z;
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 can be a definition.

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

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

The equivalent while loop would have an extra block.

```c
{
  int i = 100;
  while (i >= 0) {
    printf("%d\n", i);
    --i;
  }
}
```
Any of the three components of a `for` statement can be omitted.

If the expression is omitted, it is always “true”.

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

The *comma operator* (,) allows for multiple sub-expressions in the *setup* and *update* statements of a `for` loop. Do not use it in this course. See CP:AMA 6.3 for more details.

```c
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.
Accessing memory

The smallest accessible unit of memory is a byte.

To access a byte of memory, its position in memory, which is known as the address of the byte, must be known.

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.

<table>
<thead>
<tr>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 MB of storage)</td>
<td>(one byte per address)</td>
</tr>
<tr>
<td>0x00000</td>
<td>00101001</td>
</tr>
<tr>
<td>0x00001</td>
<td>11001101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xFFFFF</td>
<td>00010111</td>
</tr>
<tr>
<td>0xFFFFFE</td>
<td>00010111</td>
</tr>
<tr>
<td>0xFFFFFF</td>
<td>01110011</td>
</tr>
</tbody>
</table>

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

For a variable definition, C

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

For example, with the definition

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

C reserves space (an address) to store n, “keeps track of” the address n, and stores the value 0 at that address.
In our CS 135 substitution model, a variable is a “name for a value”. When a variable appears in an expression, a substitution occurs and the name is replaced by its value.

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

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

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

It depends on the type of the variable.

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

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

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

In this course, the size of an integer 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, only use int, and there are always 32 bits in an int.
example: variable definition

```
int n = 0;
```

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

<table>
<thead>
<tr>
<th>identifier</th>
<th>type</th>
<th># bytes</th>
<th>starting address</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>( \text{int} )</td>
<td>4</td>
<td>( 0x5000 )</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>address</th>
<th>0x5000</th>
<th>0x5001</th>
<th>0x5002</th>
<th>0x5003</th>
</tr>
</thead>
<tbody>
<tr>
<td>contents</td>
<td>000000000</td>
<td>000000000</td>
<td>000000000</td>
<td>000000000</td>
</tr>
</tbody>
</table>
Integer limits

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

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

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.

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, sometimes overflow can be avoided.

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

```c
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 \((\text{expt 2 1000})\).

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 \textit{structure} that can use an arbitrary number of bytes (imagine a \textit{list} of bytes).

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

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

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

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

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

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

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

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

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

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

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

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

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

For example, 'a' is equivalent to 97 and 'z' is 122.

C “translates” 'a' into 97.

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

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

Always use **single** quotes with characters:

"a" is **not** the same as 'a'.
The printf format specifier to display a character is "\%c".

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

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

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

letter_a as a character:  a
ninety_seven as a char:  a
letter_a in decimal:     97
ninety_seven in decimal: 97
Character arithmetic

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

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

// to_lowercase(c) converts upper case letters to // lowercase letters, everything else is unchanged
```c
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 the application.

`read_char` has a parameter for specifying if whitespace is ignored.
Symbol type

There is no C equivalent of the Racket 'symbol type.

C *symbols* are constants (often ints) with meaningful identifiers ("names") but arbitrary (meaningless) values.

Use **ALL CAPS** for symbol names.

```c
const int UP = 1;
const int DOWN = 2;

int direction = UP;
```

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 enums in a C reference and understand how to use them.

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

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

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

Unfortunately, floats are susceptible to precision errors.

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

```racket
(sqr 2)  ; => #i1.4142135623730951
(sqr (sqr 2))  ; => #i2.0000000000000004
```
example 1: inexact floats

```c
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 format specifier 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, do 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 $6.022 \times 10^{23}$, a float uses a similar strategy.

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

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

Floats and their internal representation are discussed in CS 251 / 230 and in detail in CS 370 / 371.
Structures

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

```c
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.

```c
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:

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

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

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

```c
struct posn p = {.x = 3}; // .y = 0
```
Mutation with structures

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

```c
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 cannot 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”.

```c
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.

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

Also, `printf` only works with elementary types. Print each field of a structure individually:

```c
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:

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

Memory is only reserved when a struct variable is defined.

```c
struct posn p = {3, 4};
```
sizeof a struct

```c
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.

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

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

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

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

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

```c
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:

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

Other courses may use alternative names.

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

When you try to access memory outside of a segment, a segmentation fault occurs (more on this in CS 350).
**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:

```c
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 returns, the variable (and the entire frame) is popped and effectively “disappears”.

In C, local variables are known as *automatic* variables because they are “automatically” created when needed. There is an auto keyword in C but it is rarely used.
```c
int h(int i) {
    int r = 10 * i;
    return r;
}

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

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

int main(void) {
    int a = f(2);
    //...
    return a;
    return address: OS
}
```
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
When a function returns:

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

The return `value` (for non-`void` functions) is stored in a `temporary` memory area we are not discussing in this course. This is discussed further in CS 241.
The return address is a code location from the **calling function**.

It has **nothing** to do with the **location of** any return statement(s), or if one does not exist (e.g., a **void** function).

There is **always** one (and only one) return address in a stack frame.

```c
int sign(int n) {
    if (n > 0) {
        return 1;
    } else if (n < 0) {
        return -1;
    }
    return 0;
}
```
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

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

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

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 variable definitions 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.

```c
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)

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

```c
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
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).

```c
int g = 42;

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

the value of g is: 42
the address of g is: 0x71a0a0

The printf format specifier 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.

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

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

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

\[
\text{int } \ast p = \&i; \quad \text{// } p \text{ "points at" } i
\]

is comparable to the following definition and assignment:

\[
\text{int } \ast p; \quad \text{// } p \text{ is defined (not initialized)} \\
p = \&i; \quad \text{// } p \text{ now "points at" } i
\]

The \( \ast \) is part of the definition of \( p \) and is **not part of the variable name**. The name of the variable is simply \( p \), not \( \ast p \).

As with any variable, its value can be changed.

\[
p = \&j; \quad \text{// } p \text{ now "points at" } j \\
p = \&i; \quad \text{// } p \text{ now "points at" } i
\]
The **value** of a pointer is an **address**.

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

<table>
<thead>
<tr>
<th>identifier</th>
<th>type</th>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>int</td>
<td>0xf020</td>
<td>42</td>
</tr>
<tr>
<td>p</td>
<td>int *</td>
<td>0xf024</td>
<td>0xf020</td>
</tr>
</tbody>
</table>

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

![Memory Diagram](image)
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.

\[
\begin{align*}
\text{sizeof(int *)} & \Rightarrow 8 \\
\text{sizeof(char *)} & \Rightarrow 8
\end{align*}
\]
Indirection operator

The **indirection operator** (*), also known as the **dereference operator**, is the **inverse** of the **address operator** (&).

*p* produces the **value** of what pointer p “points at”.

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

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".

\[
\text{\*p} \Rightarrow 42
\]
The * symbol is used in three different ways in C:

• as the *multiplication operator* between expressions
  \[ k = i \times i; \]

• in pointer *definitions* and pointer *types*
  \[ \text{int } *p = \&i; \]
  \[ \text{sizeof(int *} \]

• as the *indirection operator* for pointers
  \[ j = *p; \]
  \[ *p = 5; \]

\(( *p \times *p \) \) is a confusing but valid C expression.
C mostly ignores whitespace, so these are equivalent

```c
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)

```c
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?”

```c
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`.  

---

CS 136 Winter 2020 05: Pointers
In C, we can define a **pointer to a pointer**:

\[
\begin{align*}
\text{int } i &= 42; \\
\text{int } *p1 &= \&i; \quad // \text{pointer } p1 \text{ points at } i \\
\text{int } **p2 &= \&p1; \quad // \text{pointer } p2 \text{ points at } p1
\end{align*}
\]

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

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

A void pointer (void *) can point at anything, including a void pointer (itself).
Dereferencing pointers to structures

Unfortunately, the structure operator (.) has higher precedence than the indirection operator (*).

Awkward parenthesis are required to access a field of a pointer to a structure: (*ptr).field.

Fortunately, the *indirection selection operator*, also known as the “arrow” operator (->) combines the indirection and the selection operators.

ptr->field is equivalent to (*ptr).field
Example: Indirection selection operator

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

int main(void) {
    struct posn my_posn = {0, 0};
    struct posn *ptr = &my_posn;

    (*ptr).x = 3; // awkward
    ptr->y = 4;  // much better

    //...
}
```
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.

```c
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:

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

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

```c
p = NULL;
i = *p;  // crash!
```
Pointer assignment

Consider the following code

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

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

p = q;
```

The statement `p = q;` is a **pointer assignment**. It means “change `p` to point at what `q` points at”. It changes the value of `p` to be the value of `q`. In this example, it assigns the address of `j` to `p`.

It does not change the value of `i`. 
p = q;
Using the same initial values,

    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.
*p = *q;
example: aliasing

```c
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:

```c
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 its own copy of the argument (in the stack frame) without changing the original variable.
```c
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`”.

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

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

int main(void) {
    int x = 5;
    inc(&x);
}
```
Most pointer parameters should be **required** to be valid (e.g., non-NULL).

```c
// 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

// 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

```c
// 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, multiple format specifiers can be used 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 an int, and either:

- the quantity (count) of values successfully read. This will be zero if the input is not formatted properly (e.g., the input [hello] is not a valid int).

- the constant EOF: the End Of File (EOF) has been reached.

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

In our seashell environment, EOF is defined as -1, but it is much better style to use the constant EOF instead of -1.
Invalid input

Always check the return value of `scanf`: one is “success”.
(if you are following our advice to read one value per `scanf`).

```c
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");
}
```

The `read_int()` function returns `READ_INT_FAIL (INT_MIN)`
if the return value is not 1 (i.e., 0 or EOF).
example: reading integers

This function reads in ints from input (until EOF or an unsuccessful read occurs) and returns their sum.

```c
int read_sum(void) {
    int sum = 0;
    int n = 0;
    while (scanf("%d", &n) == 1) {
        sum += n;
    }
    return sum;
}
```
Whitespace

When reading an int with scanf ("%d") C **ignores any whitespace** (spaces and newlines) that appears before the next int.

When reading in a char, you *may or may not* want to ignore whitespace: it depends on your application.

```
// 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 leading whitespace is ignored.
Using pointers to “return” multiple values

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.

```c
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:

```c
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).

```c
bool divide(int num, int denom, int *quot, int *rem) {
    if (denom == 0) return true;
    *quot = num / denom;
    *rem = num % denom;
    return false;
}
```

Some C library functions use this approach to return an error. Other functions use “invalid” sentinel values such as -1 or NULL to indicate when an error has occurred.
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.

```c
int *bad_idea(int n) {
    return &n; // NEVER do this
}

int *bad_idea2(int n) {
    int a = n*n;
    return &a; // NEVER do this
}
```

As soon as the function returns, the stack frame “disappears”, and all memory within the frame is 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.

```c
struct bigstruct {
    int a;
    int b;
    int c;
    int d;
    ...
    int y;
    int z;
};
```
To avoid structure copying, it is very common to pass the \textit{address} of a structure to a function.

\begin{verbatim}
// sqr_dist(p1, p2) calculates the square of the distance between p1 and p2

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;
}
\end{verbatim}
Passing the address of a structure to a function (instead of a copy) also allows the function to mutate the fields of the structure.

```c
// 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.

```c
// 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.

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

Place const 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.

```c
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”.

```c
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.

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

```c
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*.

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

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

Because a **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.

Global `mutable` variables are **strongly discouraged** and considered “poor 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. **“Pure” function**
   
   No side effects or dependencies on global *mutable* variables.

2. **Only I/O side effects**
   
   If possible, avoid any mutative side effects.

3. **Mutate data through pointer parameters**
   
   If mutation is necessary, use a pointer parameter.

4. **Global dependencies**

   Mutable global variables should be avoided.

5. **Mutate global data**

   Only when absolutely necessary (it rarely is).
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.
```c
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. There are 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”.

---

CS 136 Winter 2020

06: Modularization & ADTs
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)?

```c
// main.c [CLIENT]               // fun.c [MODULE]

int main(void) {
    //...
    b = is_fun(k);  // ERROR
    //...
}
```

The is_fun function is not in scope.

Also, because C is statically typed, it needs to know the return and parameter types of is_fun...
Declarations

In C, a function or variable must be declared before (“above”) it can be “accessed” (or referenced).

A declaration introduces an identifier (“name”) into a program and specifies its type.

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

Here, when we use “identifiers” it does not include structures (structs) – they are declared differently (more on them later).
**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.

It specifies the function type (the return and parameter types).

**example: function declaration**

```c
int my_add(int a, int b); // function DECLARATION

int main(void) {
    trace_int(my_add(1, 2)); // this is now ok
}

int my_add(int a, int b) { // function DEFINITION
    return a + b;
}
```

By declaring `my_add` above `main`, it is now available in `main`.
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:

int my_add(int a, int b);
int my_add(int, int);
int my_add(int these_are, int ignored);
```

It is good style to include the correct (and meaningful) parameter names in the declaration to aid communication.
A **variable declaration** starts with the `extern` keyword, followed by the type and then the variable name. There is **no initialization**.

**example: variable declaration**

```c
extern int g;  // variable DECLARATION

int main(void) {
    printf("%d\n", g);  // this is now ok
}

int g = 7;  // variable DEFINITION
```

Variable declarations are uncommon.
Revisiting Scope

One way to think about declarations is that they **extend** the *scope* of an identifier.

```c
extern int g; // declaration: 
// g is now IN scope

int main(void) {
    printf("%d\n", g);   // ok
}

int g = 7; // definition
```
Program scope

A declaration can bring an identifier from another file into scope:

```
// main.c [CLIENT]     // module.c [MODULE]

extern int g;
// g IN scope

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

By default, C global identifiers are “accessible” to every file in the program (if they are declared).

We will refer to this as program scope.
Revisiting module functions

We have solved our previous problem. We can now call our module function \texttt{is\_fun} from our \texttt{client} (\texttt{main.c}) by declaring it:

\begin{verbatim}
// main.c [CLIENT]
// *** NEW declaration ***
bool is_fun(int n);

int main(void) {
    //...
    b = is_fun(k);    // ok
    //...
}
// fun.c [MODULE]
bool is_fun(int n) {
    //...
}
\end{verbatim}
Module scope

To “hide” a global identifier from other files, prefix the definition with the static keyword.

// main.c [CLIENT]                                           // module.c [MODULE]
extern int g;

static int g = 7;

int main(void) {
    printf("%d\n", g);  // ERR
}

In other words, the static keyword restricts the scope of a global identifier to the file (module) it is defined in.

We will refer to this as module scope.
Types of scope (revisited)

- **local** (block) identifiers
  
  only available inside of the function (or *block*)

- **global** identifiers:
  
  - *program scope* identifiers (default)
    
    available to any file in the program (if declared)
  
  - *module scope* identifiers (defined as static)
    
    only available in the file they are defined in

We continue to use *global scope* to refer to identifiers that have *either* program or module scope.
Use static to give module functions and variables *module scope* if they are not meant to be **provided**.

// fun.c [MODULE]

static const int module_scope_variable = 42;

static int is_fun_helper(int n) {
    // ...
}

bool is_fun(int n) {
    //...
}

The static keyword is **not** related to *static typing*. private would have been a better choice than static.
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 would also provide *examples* to illustrate how the module is used and how the interface functions interact.

Examples are not required in this course.
Interface (.h) files

For C modules, the interface is placed in a separate file with a .h file extension.

```c
// 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 header 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.

```c
// 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 the 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);
    //...
}

CS 136 Winter 2020 06: Modularization & ADTs
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.

```c
// 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.

```c
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 constant 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.

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

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

In C99, it is 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`
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 declaration* **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: transparent and opaque structures

// module.h [INTERFACE]

struct transparent {
    int t_field;
};

struct opaque;

// module.c [IMPLEMENTATION]

#include "module.h"

struct opaque {
    int o_field;
};

// main.c [CLIENT]

#include "module.h"

int main(void) {
    struct transparent t = {0};  // VALID
    struct opaque o = {0};     // INVALID
    struct opaque *ptr;        // VALID
}

CS 136 Winter 2020 06: Modularization & ADTs 48
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 minutes and seconds 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.

```c
// 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

<table>
<thead>
<tr>
<th>key (student number)</th>
<th>value (student name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234567</td>
<td>&quot;Sally&quot;</td>
</tr>
<tr>
<td>3141593</td>
<td>&quot;Archie&quot;</td>
</tr>
<tr>
<td>8675309</td>
<td>&quot;Jenny&quot;</td>
</tr>
</tbody>
</table>
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")))
```

![Diagram of association list]
We also implemented a dictionary with a **Binary Search Tree (BST)** data structure.

\[
\text{(define bst (make-node 3141593 "Archie"
  (make-node 1234567 "Sally" empty empty)
  (make-node 8675309 "Jenny" empty empty)))}
\]
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:

- **structures**
- **arrays**

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

An array is a data structure that contains a **fixed number** of elements that all have the **same type**.

Because arrays are *built-in* to C, they are used for many tasks where *lists* are used in Racket, but **arrays and lists are very different**. In Section 11 we construct Racket-like lists in C.
int my_array[6] = {4, 8, 15, 16, 23, 42};

To define an array we must know the **length** of the array in advance (we address this limitation in Section 10).

Each individual value in the array is known as an **element**. To access an element, its **index** is required.

The first element of `my_array` is at index 0, and it is written as `my_array[0]`.

The second element is `my_array[1]` and the last is `my_array[5].`

In computer science we often start counting at 0.
example: accessing array elements

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

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

int j = a[0];                       // j is 4
int *p = &a[j - 1];                // p points at a[3]

a[2] = a[a[0]];                    // a[2] is now 23
++a[1];                            // a[1] is now 9
```
example: arrays & iteration

Arrays and iteration are a powerful combination.

```c
int a[6] = {4, 8, 15, 16, 23, 42};
int sum = 0;

for (int i = 0; i < 6; ++i) {
    printf("a[%d] = %d\n", i, a[i]);
    sum += a[i];
}
printf("sum = %d\n", sum);
```

```
a[0] = 4
a[1] = 8
a[2] = 15
a[3] = 16
a[4] = 23
a[5] = 42
sum = 108
```
Array initialization

Arrays can only be initialized with braces ({}).

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

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.

```c
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:

```c
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:

```c
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.

```c
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.

```cpp
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.

```cpp
int a[6] = ...;
```
A preferred syntax to specify the length of an array is to define a macro.

```c
#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.

```c
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 `sizeof` can be used to determine the length of an array.

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

The CP:AMA textbook uses this on occasion.

However, in practice (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: $(6 \times \text{sizeof(int)}) = 6 \times 4 = 24$.

Not everyone uses the same terminology for length and size.
example: array in memory

```c
int a[6] = {4, 8, 15, 16, 23, 42};
printf("&a[0] = %p ... &a[5] = %p\n", &a[0], &a[5]);
&a[0] = 0x5000 ... &a[5] = 0x5014
```

<table>
<thead>
<tr>
<th>addresses</th>
<th>contents (4 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5000 ... 0x5003</td>
<td>4</td>
</tr>
<tr>
<td>0x5004 ... 0x5007</td>
<td>8</td>
</tr>
<tr>
<td>0x5008 ... 0x500B</td>
<td>15</td>
</tr>
<tr>
<td>0x500C ... 0x500F</td>
<td>16</td>
</tr>
<tr>
<td>0x5010 ... 0x5013</td>
<td>23</td>
</tr>
<tr>
<td>0x5014 ... 0x5017</td>
<td>42</td>
</tr>
</tbody>
</table>
The array identifier

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

```c
int a[6] = {4, 8, 15, 16, 23, 42};
trace_ptr(a);
trace_ptr(&a);
trace_ptr(&a[0]);
```

\(a \Rightarrow 0x5000\)
\(&a \Rightarrow 0x5000\)
\(&a[0] \Rightarrow 0x5000\)

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]).

```c
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.
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.

```c
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.

```c
int sum_array(const int a[], int len) {
    int sum = 0;
    for (int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```
Because a structure can contain an array:

```c
struct mystruct {
    int big[1000];
};
```

It is *especially* important to pass a pointer to such a structure, otherwise, the **entire array** is copied to the stack frame.

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

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 size of whatever p points at.

According to the official C standard, pointer arithmetic is only valid within an array (or a structure) context. This becomes clearer later.
Pointer arithmetic rules

• When adding an integer \( i \) to a pointer \( p \), the address computed by \((p + i)\) in C is given in “normal” arithmetic by:

\[ p + i \times \text{sizeof}(*p). \]

• Subtracting an integer from a pointer \((p - i)\) works in the same way.

• Mutable pointers can be incremented (or decremented). \;++p \text{ is equivalent to } p = p + 1.\]
• You cannot add two pointers.

• A pointer \( q \) can be subtracted from another pointer \( p \) if the pointers are the same type (point to the same type). The value of \( (p - q) \) in C is given in “normal” arithmetic by:

\[
(p - q)/\text{sizeof}(*p).
\]

In other words, if \( p = q + i \) then \( i = p - q \).

• Pointers (of the same type) can be compared with the comparison operators: \(<, \leq, =, \neq, \geq, >\) (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.

```c
int sum_array(const int *a, int len) {
    int sum = 0;
    for (const int *p = a; p < a + len; ++p) {
        sum += *p;
    }
    return sum;
}
```

Note that the above code behaves *identically* to the previously defined `sum_array`:

```c
int sum_array(const int a[], int len) {
    int sum = 0;
    for (int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```
another example: pointer notation

// count_match(item, a, len) counts the number of occurrences of item in the array a

int count_match(int item, const int *a, int len) {
    int count = 0;
    const int *p = a;
    while (p < a + len) {
        if (*p == item) {
            ++count;
        }
        ++p;
    }
    return count;
}
The choice of notation (pointers or []) is a matter of style and context. You are expected to be comfortable with both.

C makes no distinction between the following two function declarations:

```c
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.

```c
// 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]);
    }
}
```
#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...

\[
\begin{array}{ccccccc}
8 & 6 & 7 & 5 & 3 & 0 & 9 \\
\end{array}
\]

and then we *swap* the first element with the smallest.

\[
\begin{array}{ccccccc}
0 & 6 & 7 & 5 & 3 & 8 & 9 \\
\end{array}
\]

Then, we find the next smallest element...

\[
\begin{array}{ccccccc}
0 & 6 & 7 & 5 & 3 & 8 & 9 \\
\end{array}
\]

and then we *swap* that element with the second one, and so forth...

\[
\begin{array}{ccccccc}
0 & 3 & 7 & 5 & 6 & 8 & 9 \\
\end{array}
\]
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.
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.

\[
\text{(define (quick-sort lon)} \\
\quad \text{(cond [(empty? lon) empty]} \\
\quad \quad \text{[else (define pivot (first lon))]} \\
\quad \quad \quad \text{(define less (filter (lambda (x)} \\
\quad \quad \quad \quad \quad \quad \quad (<= x pivot)) (rest lon)))} \\
\quad \quad \quad \text{(define greater (filter (lambda (x)} \\
\quad \quad \quad \quad \quad \quad \quad (> x pivot)) (rest lon)))} \\
\quad \quad \text{(append (quick-sort less)} \\
\quad \quad \quad \text{(list pivot)} \\
\quad \quad \quad \text{(quick-sort greater))]})))
\]

For simplicity, we select the first element as the “pivot”. A more in-depth discussion of pivot selection occurs in CS 240.
In our C implementation of quick sort, we:

- select the first element of the array as our “pivot”
- move all elements that are larger than the pivot to the back of the array
- move (“swap”) the pivot into the correct position
- recursively sort the “smaller than” sub-array and the “larger than” sub-array

The core quick sort function quick_sort_range has parameters for the range of elements (first and last) to be sorted, so a wrapper function is required.
void quick_sort_range(int a[], int first, int last) {

    if (last <= first) return;  // length is <= 1

    int pivot = a[first];       // first element is the pivot
    int pos = last;             // where to put next larger

    for (int i = last; i > first; --i) {
        if (a[i] > pivot) {
            swap(&a[pos], &a[i]);
            --pos;
        }
    }
    swap(&a[first], &a[pos]);   // put pivot in correct place
    quick_sort_range(a, first, pos - 1);
    quick_sort_range(a, pos + 1, last);
}

void quick_sort(int a[], int len) {
    quick_sort_range(a, 0, len - 1);
}
Binary search

In Racket, the built-in function `member` can be used to determine if a list contains an element.

We can write a similar function in C that finds the index of an element in an array:

```c
// find(item, a, len) finds the index of item in a,
// or returns -1 if it does not exist

int find(int item, const int a[], int len) {
    for (int i = 0; i < len; ++i) {
        if (a[i] == item) {
            return i;
        }
    }
    return -1;
}
```
But what if the array was previously *sorted*?

We can use **binary search** to find the element faster:

```c
int find_sorted(int item, const int a[], int len) {
    int low = 0;
    int 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.

```c
int data[6] = {1, 2, 3, 7, 8, 9};
```

To access the entry in row \( r \) and column \( c \), we simply access the element at \( \text{data}[r \times 3 + c] \).

In general, it would be \( \text{data}[\text{row} \times \text{NUMCOLS} + \text{col}] \).
C supports multiple-dimension arrays, but they are not covered in this course.

```c
int two_d_array[2][3];
int three_d_array[10][10][10];
```

When multi-dimensional arrays passed as parameters, the second (and higher) dimensions must be fixed. (e.g., `int function_2d(int a[][10], int numrows)`).

Internally, C represents a multi-dimensional array as a 1D array and performs “mapping” similar to the method described in the previous slide.

See CP:AMA sections 8.2 & 12.4 for more details.
Oversized Arrays

A significant limitation of an array is that the length of the array must be known **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 an **oversized** array with a “maximum” length.

In general, oversized arrays should only be used when appropriate:

- They are wasteful if the maximum length is excessively large.
- They are restrictive if the maximum length is too small.
When working with oversized arrays, we need to keep track of

- the "actual" length of the array, and
- the maximum possible length.
To illustrate oversized 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.

```c
struct stack {
    int len;
    int maxlen;
    int data[100];
};
```

We need to provide a `stack_init` function to initialize the structure:

```c
void stack_init(struct stack *s) {
    assert(s);
    s->len = 0;
    s->maxlen = 100;
}
```
Ignoring the push operation for now, we can write the rest of the stack implementation:

```c
bool stack_is_empty(const struct stack *s) {
    assert(s);
    return s->len == 0;
}

int stack_top(const struct stack *s) {
    assert(s);
    assert(s->len > 0);
    return s->data[s->len - 1];
}

// note: stack_pop returns the element popped
int stack_pop(struct stack *s) {
    assert(s);
    assert(s->len > 0);
    s->len -= 1;
    return s->data[s->len];
}
```
What happens if we exceed the maximum length when we try to push an element?

There are a few possibilities:

- the stack is not modified and an error message is displayed
- a special return value can be used
- an assertion fails (terminating the program)
- the program explicitly **terminates** with an error message

Any approach may be appropriate as long as the contract properly documents the behaviour.
The exit function (part of `<stdlib.h>`) stops program execution. It is useful for “fatal” errors.

The argument passed to `exit` is equivalent to the return value of `main`.

For convenience, `<stdlib.h>` defines `EXIT_SUCCESS` which is 0 and `EXIT_FAILURE` which is non-zero.

```c
if (something_bad) {
    printf("FATAL ERROR: Something bad happened!\n");
    exit(EXIT_FAILURE);
}
```
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 oversized 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 (≈ 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. Power efficiency (power consumption) is also becoming an important measure.

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?

```c
int sum_array(const int a[], int len) {
    int sum = 0;
    int i = 0;
    while (i < len) {
        sum = sum + a[i];
        i = i + 1;
    }
    return sum;
}
```

The running time depends on the length of the array.

If there are \( n \) items in the array, it requires \( 7n + 3 \) operations.

We are always interested in the running time with respect to the size of the 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 clearly state what $n$ represents.

For example, with lists of strings, $n$ may represent the number of strings in the list, or it may represent the length of all of the strings in the list.

The *running Time* of an implementation is a function of $n$ and is written as $T(n)$. 
There may also be another **attribute** of the 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.

```c
// check_array(a, len, e, o) determines if array a
//     contains at least e even numbers and
//     at least o odd numbers
// requires: len > 0
//          elements of a > 0
//          e, o >= 0
```

Homer and Bart are debating the best algorithm (strategy) for implementing \texttt{check\_array}.
Bart just wants to count the total number of odd numbers in the entire array.

```cpp
bool bart(const int a[], int len, int e, int o) {
    int odd_count = 0;
    for (int i = 0; i < len; i = i + 1) {
        odd_count = odd_count + (a[i] % 2);
    }
    return (odd_count >= o) && (len - odd_count >= e);
}
```

If there are $n$ elements in the array, $T(n) = 8n + 7$. 

Remember, you are not expected to calculate this precisely.
Homer is lazy, and he doesn’t want to check all of the elements in the array if he doesn’t have to.

```c
bool homer(const int a[], int len, int e, int o) {
    // only loop while it's still possible
    while (len > 0 && e + o <= len) {
        if (a[len - 1] % 2 == 0) { // even case:
            if (e > 0) {
                e = e - 1; // only decrement e if e > 0
            }
        } else if (o > 0) {
            o = o - 1;
        }
        if (e == 0 && o == 0) {
            return true;
        }
        len = len - 1;
    }
    return false;
}
```
The problem with analyzing Homer's code is that it depends not just on the length of the array, but on the contents of the array and the parameters e and o.

```c
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// these 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.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Equation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homer</td>
<td>$T(n) = 4$</td>
<td>best case</td>
</tr>
<tr>
<td></td>
<td>$T(n) = 17n + 1$</td>
<td>worst case</td>
</tr>
<tr>
<td>Bart</td>
<td>$T(n) = 8n + 7$</td>
<td>all cases</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>Order</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$2016 = O(1)$</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>$100000 + n = O(n)$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$n + n \log n = O(n \log n)$</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td></td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>$999n + 0.01n^2 = O(n^2)$</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>$\frac{n(n+1)(2n+1)}{6} = O(n^3)$</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>$n^3 + 2^n = O(2^n)$</td>
</tr>
</tbody>
</table>
When comparing algorithms, the most efficient algorithm is the one with the lowest order.

For example, an $O(n \log n)$ algorithm is more efficient than an $O(n^2)$ algorithm.

If two algorithms have the same order, they are considered equivalent.

Both Homer’s and Bart’s implementations are $O(n)$, so they are equivalent.
Big O arithmetic

When *adding* two orders, the result is the largest of the two orders.

- $O(\log n) + O(n) = O(n)$
- $O(1) + O(1) = O(1)$

When *multiplying* two orders, the result is the product of the two orders.

- $O(\log n) \times O(n) = O(n \log n)$
- $O(1) \times O(n) = O(n)$
There is no “universally accepted” Big O notation.

In many textbooks, and in this introductory course, the notation
\[ T(n) = 1 + 2n + 3n^2 = O(1) + O(n) + O(n^2) = O(n^2) \]
is acceptable.

In other textbooks, and in other courses, this notation may be too informal.

In CS 240 and CS 341 you will study orders and Big O notation much more rigourously.
Algorithm analysis

An important skill in Computer Science is the ability to **analyze** a function and determine the **order** of the running time.

In this course, our goal is to give you experience and work toward building your intuition:

```c
int sum_array(const int a[], int len) {
    int sum = 0;
    for (int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```

“Clearly, each element is visited once, so the running time of `sum_array` is \(O(n)\)”. 

Contract update

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, specify it.

// int f(int a[], int a_len, int b[], int b_len) ...
// time: $O(n^2 + m)$ where $n$ is a_len and $m$ is b_len

For a function with a single array, it is obvious:

// sum_array(const int a[], int len) sums the elements
// of array a
// time: $O(n)$
Analyzing simple functions

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

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

If no other functions are called, there must be a fixed number of operators. Each operator is $O(1)$, so the running time is:

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

If a simple function calls other functions, its running time depends on those functions.
Iterative analysis

*Iterative analysis* uses summations.

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

\[
T(n) = \sum_{i=1}^{n} O(1) = O(1) + \cdots + O(1) = n \times O(1) = O(n)
\]

Because we are primarily interested in *orders*,

\[
\sum_{i=0}^{n-1} O(x), \sum_{i=1}^{10n} O(x), \text{ or } \sum_{i=1}^{n/2} O(x) \text{ are equivalent* to } \sum_{i=1}^{n} O(x)
\]

* 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

\[
\text{sum} = 0;
\text{for} \ (i = 0; \ i < n; \ ++i) \ \{ \\
\quad \text{sum} += i;
\}\]

\[
\sum_{i=1}^{n} O(1) = O(n)
\]
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.

```c
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.

\[
n/10 \sum_{i=1}^{n/10} O(1) = O(n)
\]
When the loop counter changes geometrically, the number of iterations is often logarithmic.

```c
k = n; // n is size of the 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.

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

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

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

To determine the running time of a recursive function we must determine the *recurrence relation*. For example,

\[ T(n) = O(n) + T(n - 1) \]

We can then look up the recurrence relation in a table to determine the *closed-form* (non-recursive) running time.

\[ T(n) = O(n) + T(n - 1) = O(n^2) \]

In later courses, you *derive* the closed-form solutions and *prove* their correctness.
The recurrence relations we encounter in this course are:

<table>
<thead>
<tr>
<th>Relation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = O(1) + T(n - k_1)$</td>
<td>$= O(n)$</td>
</tr>
<tr>
<td>$T(n) = O(n) + T(n - k_1)$</td>
<td>$= O(n^2)$</td>
</tr>
<tr>
<td>$T(n) = O(n^2) + T(n - k_1)$</td>
<td>$= O(n^3)$</td>
</tr>
<tr>
<td>$T(n) = O(1) + T\left(\frac{n}{k_2}\right)$</td>
<td>$= O(\log n)$</td>
</tr>
<tr>
<td>$T(n) = O(1) + k_2 \cdot T\left(\frac{n}{k_2}\right)$</td>
<td>$= O(n)$</td>
</tr>
<tr>
<td>$T(n) = O(n) + k_2 \cdot T\left(\frac{n}{k_2}\right)$</td>
<td>$= O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = O(1) + T(n - k_1) + T(n - k'_1)$</td>
<td>$= O(2^n)$</td>
</tr>
</tbody>
</table>

where $k_1, k'_1 \geq 1$ and $k_2 > 1$

This table will be provided on exams.
Procedure for recursive functions

1. Identify the order of the function *excluding* any recursion
2. Determine the size of the 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

```c
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, 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:

```c
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 $O(n^2)$.

```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]);
        }
    }
}
```

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

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>best case</th>
<th>worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>insertion sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>quick sort</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

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.

```c
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 \((n = \text{high}-\text{low})\) was halved, so the running time is:

\[
T(n) = \sum_{i=1}^{\log_2 n} O(1) = O(\log n)
\]
Algorithm Design

In this introductory course, the algorithms we develop 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:

\[
\begin{array}{cccccccccc}
31 & -41 & 59 & 26 & -53 & 58 & 97 & -93 & -23 & 84 \\
\end{array}
\]

the maximum sum is 187:

\[
\begin{array}{cccccccccc}
31 & -41 & 59 & 26 & -53 & 58 & 97 & -93 & -23 & 84 \\
\end{array}
\]

This problem has many applications, including pattern recognition in artificial intelligence.
Solution A: $O(n^3)$

// for every start position i and ending position j
// loop between them (k) summing elements
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.
Big O revisited

We now revisit *Big O notation* and define it more formally.

\[ O(g(n)) \] is the **set** of all functions whose “order” is **less than or equal** to \( g(n) \).

\[
\begin{align*}
    n^2 &\in O(n^{100}) \\
n^3 &\in O(2^n)
\end{align*}
\]

While you can say that \( n^2 \) is in the set \( O(n^{100}) \), it’s not very useful information.

In this course, we always want the **most appropriate** order, or in other words, the **smallest** correct order.
Big O describes the *asymptotic* behaviour of a function.

This is **different** than describing the **worst case** behaviour of an algorithm.

Many confuse these two topics but they are completely **separate concepts**. The Best case and the worst case can both be described asymptotically.

For example, the best case insertion sort is $O(n)$, while the worst case is $O(n^2)$. 
A slightly more formal definition of Big O is

\[ f(n) \in O(g(n)) \iff f(n) \leq c \cdot g(n) \]

for large \( n \) and some positive number \( c \).

This definition makes it clear why we “ignore” constant coefficients.

For example,

\[ 9n \in O(n) \quad \text{for } c = 10, \quad 9n \leq 10n, \text{ and} \]

\[ 0.01n^3 + 1000n^2 \in O(n^3) \quad \text{for } c = 1001, \quad 0.01n^3 + 1000n^2 \leq 1001n^3 \]
The full definition of Big O is

\[ f(n) \in O(g(n)) \iff \exists c, n_0 > 0, \forall n \geq n_0, f(n) \leq c \cdot g(n) \]

\( f(n) \) is in \( O(g(n)) \) if there exists a positive \( c \) and \( n_0 \) such that for any value of \( n \geq n_0 \), \( f(n) \leq c \cdot g(n) \).
In later CS courses, you will use the formal definition of Big O to prove algorithm behaviour more rigourously.

There are other asymptotic functions in addition to Big O.

(for each of the following, \( \exists n_0 > 0, \forall n \geq n_0 \ldots \))

\[
\begin{align*}
f(n) \in \omega(g(n)) & \iff \forall c > 0, c \cdot g(n) \leq f(n) \\
f(n) \in \Omega(g(n)) & \iff \exists c > 0, c \cdot g(n) \leq f(n) \\
f(n) \in \Theta(g(n)) & \iff \exists c_1, c_2 > 0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \\
f(n) \in O(g(n)) & \iff \exists c > 0, f(n) \leq c \cdot g(n) \\
f(n) \in o(g(n)) & \iff \forall c > 0, f(n) \leq c \cdot g(n)
\end{align*}
\]

\( O(g(n)) \) is often used when \( \Theta(g(n)) \) is more appropriate.
Goals of this Section

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

- use the new terminology introduced (e.g., algorithm, time efficiency, running time, order)
- compute the order of an expression
- explain and demonstrate the use of Big O notation and how $n$ is used to represent the size of the 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

• use running times in your contracts
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*.

```c
char my_string[4] = {'c', 'a', 't', '\0'};
```

The *null character*, also known as a null *terminator*, is a char with a value of zero. It is often written as '\0' instead of just 0 to improve communication and indicate that a null character is intended.

'\0' is equivalent to 0. That is different from '0', which is equivalent to 48 (the ASCII character for the symbol zero).
String initialization

The following definitions create equivalent 4-character arrays:

```c
char a[4] = {'c', 'a', 't', '\0'};
char b[4] = {'c', 'a', 't', 0};
char c[4] = {'c', 'a', 't'};
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.

```c
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.

```c
// these are equivalent
char a[4] = {'c', 'a', 't', '\0'};
char b[] = "cat";
```

As we will explain later, 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.

```
// time: O(n)
int my_strlen(const char s[]) {
    int len = 0;
    while (s[len]) {
        ++len;
    }
    return len;
}
```
Here is an alternative implementation of `my_strlen` that uses pointer arithmetic.

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

Traditionally, string functions often used pointer notation. It is slightly faster than array index notation (`s[i]`), which requires an extra addition per iteration. In modern environments, the speedup is negligible.
Do **NOT** put the `strlen` function within a loop unnecessarily.

```c
int char_count(char c, char *s) {
    int count = 0;
    for (int i = 0; i < strlen(s); ++i) {  // BAD !!!!
        if (s[i] == c) ++count;
    }
    return count;
}
```

By using an $O(n)$ function (`strlen`) inside of the loop, the function becomes $O(n^2)$ instead of $O(n)$.

Unfortunately, this mistake is common amongst beginners.

This will be harshly penalized on assignments & exams.
Lexicographical order

Characters can be easily compared ($c_1 < c_2$) as they are numbers, so the character order is determined by the ASCII table.

If we try to compare two strings ($s_1 < s_2$), C compares their addresses (pointers), which is not helpful.

To compare strings we are typically interested in using a lexicographical order.

Strings require us to be more careful with our terminology, as “smaller than” and “greater than” are ambiguous: are we considering just the length of the string? To avoid this problem we use precedes (“before”) and follows (“after”).
To compare two strings using a **lexicographical order**, we first compare the first character of each string. If they are different, the string with the smaller first character *precedes* the other string. Otherwise (the first characters are the same), the second characters are compared, and so on.

If the end of one string is encountered, it *precedes* the other string. Two strings are equal (the same) if they are the same length and all of their characters are identical.

The following strings are in lexicographical order:

```
"" "a" "az" "c" "cab" "cabin" "cat" "catastrophe"
```
The `<string.h>` library function `strcmp` uses lexicographical ordering.

`strcmp(s1, s2)` returns zero if the strings are identical. If `s1` precedes `s2`, it returns a negative integer. Otherwise (`s1` follows `s2`) it returns a positive integer.

```c
// time: O(n), n is min of the lengths of s1, s2
int my_strcmp(const char s1[], const char s2[]) {
    int i = 0;
    while (s1[i] == s2[i] && s1[i]) {
        ++i;
    }
    return s1[i] - s2[i];
}
```
The `<string.h>` library function `strcmp` uses lexicographical ordering.

`strcmp(s1, s2)` returns zero if the strings are identical. If `s1` precedes `s2`, it returns a negative integer. Otherwise (`s1` follows `s2`) it returns a positive integer.

```c
// time: O(n), n is min of the lengths of s1, s2
int my_strcmp(const char s1[], const char s2[]) {
    int i = 0;
    while (s1[i] == s2[i] && s1[i]) {
        ++i;
    }
    return s1[i] - s2[i];
}
```
To compare if two strings are equal (identical), use the `strcmp` function and check for zero (false).

```cpp
cchar a[] = "the same?";
cchar b[] = "the same?";
cchar 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.
String I/O

The printf format specifier for strings is `%s`.

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

printf prints out characters until the null character is encountered.

printf does not print out the null character.
When using %s with scanf, it stops reading the string when a 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);

Be very careful to reserve enough space for the string to be read in. **Do not forget the null character.**

scanf("%s", ...) automatically adds the null character.

The running time of printf and scanf with "%s" is $O(n)$.  

CS 136 Winter 2020 09: Strings 15
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.

```c
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

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

```c
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 chars 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 chars.
To read in a string that includes whitespace, the `gets` function reads until a newline (`\n`) is encountered (CP:AMA 13.3).

It is also very susceptible to overruns.

```c
char name[81];
printf("What is your full name?\n");
char *result = gets(name);
if (result == NULL) {
    // handle the error
}
```

The return value is either the address of the string (success) or `NULL` (failure).
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.
While *writing to* a buffer can cause dangerous buffer overruns, *reading from* 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`.
strcpy

The strcpy(dest, src) function (part of <string.h>) overwrites the contents of dest with the contents of src.

    // time: O(n), n is length of src
    char *my_strcpy(char *dest, const char *src) {
        char *d = dest;
        while (*src) {
            *d = *src;
            ++d;
            ++src;
        }
        *d = '\0';
        return dest;
    }

For historical reasons, the return value of strcpy is the address of dest. This is not useful and typically ignored.
strcpy can be a source of buffer overrun: always ensure that the dest array is large enough (and don’t forget the null terminator).

strcpy can also cause problems if the dest and src regions overlap.

Consider this dangerous call:

```c
char s[9] = "spam";
mystrcpy(s + 4, s);
```

The null terminator of src is overwritten, so it will continue to fill up memory with spamspamspam... until a crash occurs.
strcat

strcat(dest, src) is similar to strcpy, except it copies (appends or concatenates) src to the end of dest.

    // time: O(n + m) n,m are lengths of src, dest

    char *my_strcat(char *dest, const char *src) {
        strcpy(dest + strlen(dest), src);
        return dest;
    }

Again, ensure that the dest array is large enough.
String literals

C strings in quotations (e.g., "string") that are in an expression (i.e., not part of an array initialization) are known as string literals.

```c
printf("literal\n");

printf("literal %s\n", "another literal");

if (!strcmp(s, "literal")) ...

strcpy(dest, "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

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

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

```c
const char string_literal_1[] = "i = %d\n";
const char string_literal_2[] = "the value of j is %d\n";
```

```c
void foo(int i, int j) {
    printf(string_literal_1, i);
    printf(string_literal_2, j);
}
```

Do not try to modify a string literal. The behaviour is undefined, and it causes an error in Seashell.
Note the subtle difference between the following two definitions:

```c
int main(void) {
    char a[] = "mutable char array";
    char *p = "constant string literal";
    //...
}
```

Once again, it is helpful to think of the string literal as a separately defined `const char array`.

```c
const char string_literal_1[] = "constant string literal";
```

```c
int main(void) {
    char a[] = "mutable char array";
    char *p = string_literal_1;
    //...
}
```
Arrays vs. pointers

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

Consider again two similar string definitions:

```c
void f(void) {
    char a[] = "pointers are not arrays";
    char *p = "pointers are not arrays";
    ...
}
```

- The first reserves space for an initialized 24 character array (a) in the stack frame (24 bytes).
- The second reserves space for a char pointer (p) in the stack frame (8 bytes), initialized to point at a string literal (const char array) created in the read-only data section.
example: more arrays vs. pointers

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”).

\[
\text{int } a[6] = \{4, 8, 15, 16, 23, 42\};
\]
\[
\text{int } * \text{ const } p = a;
\]

In most practical expressions \(a\) and \(p\) would be equivalent. The only significant differences between them are:

- \(a\) has the same value as \&\(a\), while \(p\) and \&\(p\) have different values
- The size of \(a\) is 24 bytes, while sizeof\((p)\) is 8
Arrays of Strings

An array of strings can be defined as a 2D array of chars, but this approach 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.

Even though it is not a “proper” 2D array, any char can be accessed 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, also 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.

Define each mutable string separately.

```c
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).

```c
char aos2d[3][21] = {"my", "two dimensional", "char array"};
```

This is awkward because a function would need to know the fixed length in advance.

```c
void aos_function(char 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 a program.

Memory is dynamically “borrowed” from the heap. We call this allocation.

When the borrowed memory is no longer needed, it can be “returned” and possibly reused. We call this deallocation.

If too much memory has already been allocated, attempts to borrow additional memory fail.
Unfortunately, there is also a data structure known as a heap, and the two are unrelated.

To avoid confusion, prominent computer scientist Donald Knuth campaigned to use the name “free store” or the “memory pool”, but the name “heap” has stuck.

A similar problem arises with “the stack” region of memory because there is also a Stack ADT. However, their behaviour is very similar so it is far less confusing.
malloc

The malloc (memory allocation) function obtains memory from the heap dynamically. It is provided in `<stdlib.h>`.

// malloc(s) requests s bytes of contiguous memory from the heap
// and returns a pointer to a block of s bytes, or
// NULL if not enough contiguous memory is available
// time: O(1) [close enough for this course]

For example, for an array of 100 ints:

```c
int *my_array = malloc(100 * sizeof(int));
```

or a single struct posn:

```c
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.
Always use sizeof with malloc to improve portability and to improve communication.

Seashell allows

```c
int *my_array = malloc(400);
```

instead of

```c
int *my_array = malloc(100 * sizeof(int));
```

but the latter is much better style and is more portable.
Strictly speaking, the type of the malloc parameter is `size_t`, which is a special type produced by the `sizeof` operator.

`size_t` and `int` are different types of integers.

Seashell is mostly forgiving, but in other C environments using an `int` when C expects a `size_t` may generate a warning.

The proper `printf` format specifier to print a `size_t` is `%zd`. 
The declaration for the malloc function is:

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

The return type is a (void *) (void pointer), a special pointer that can point at any type.

```c
int * my_array = malloc(10 * sizeof(int));
struct posn * my_posn = malloc(sizeof(struct posn));
```
example: visualizing the heap

```c
int main(void) {
    int *arr1 = malloc(10 * sizeof(int));
    int *arr2 = malloc(5 * sizeof(int));
    //...
}
```

![Diagram of stack and heap with dynamic memory allocation]
An unsuccessful call to malloc returns NULL.

In practice it’s good style to check every malloc return value and gracefully handle a NULL instead of crashing.

```c
int *my_array = malloc(n * sizeof(int));
if (my_array == NULL) {
    printf("Sorry, I'm out of memory! I'm exiting....\n");
    exit(EXIT_FAILURE);
}
```

In the “real world” you should always perform this check, but in this course, you do not have to check for a NULL return value unless instructed otherwise.

In these notes, we omit this check to save space.
The heap memory provided by malloc is **uninitialized**.

```c
int *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, 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 must eventually be freed (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, every block must be freed.

int *my_array = malloc(n * sizeof(int));
// ...
// ...
free(my_array);
Invalid after free

Once a block of memory is freed, reading from or writing to that memory is invalid and may cause errors (or unpredictable results).

Similarly, it is invalid to free memory that was not returned by a malloc or that has already been freed.

```c
int *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 freed, so it is often good style to assign NULL to a freed pointer variable.
Memory leaks

A memory leak occurs when allocated memory is not eventually freed.

Programs that leak memory may suffer degraded performance or eventually crash.

```c
int *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 freed.
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 merged 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 returned.

```c
int *build_array(int len) {
    assert(len > 0);
    int *a = malloc(len * sizeof(int));
    for (int i = 0; i < len; ++i) {
        a[i] = i;
    }
    return a; // array exists beyond function return
}
```

This is one of the primary advantages of using the heap.
Dynamic memory side effect

Allocating (and deallocating) memory has a side effect: it modifies the “state” of the heap.

A function that allocates persistent memory (i.e., not freed) has a side effect and must be documented.

The caller (client) is responsible for freeing the memory (communicate this).

// build_array(n) returns a new array...
// effects: allocates memory (caller must free)
A function could also free memory it did not allocate.

That would also be a side effect:

```plaintext
// process_and_destroy_array(a, len) ...
// requires: a is a heap-allocated array
// effects: frees a (a is now invalid)
```

This behaviour is rare outside of ADTs.
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
- freeing the old array
example: resizing an array

As we will see shortly, this is not how it is done in practice, but this is an illustrative example.

```c
// my_array has a length of 100
int *my_array = malloc(100 * sizeof(int));

// stuff happens...

// oops, my_array now needs to have a length of 101
int *old = my_array;
my_array = malloc(101 * sizeof(int));
for (int i = 0; i < 100; ++i) {
    my_array[i] = old[i];
}
free(old);
```
realloc

To make resizing arrays easier, there is a realloc function.

```c
// realloc(p, newsize) resizes the memory block at p
// to be newsize and returns a pointer to the
// new location, or NULL if unsuccessful
// requires: p must be from a previous malloc/realloc
// effects: the memory at p is invalid (freed)
// time: O(n), where n is \text{min}(newsize, oldsize)
```

Similar to our previous example, realloc preserves the contents from the old array location.

```c
int *my_array = malloc(100 * sizeof(int));
// stuff happens...
my_array = realloc(my_array, 101 * sizeof(int));
```
The pointer returned by `realloc` may actually be the *original* pointer, depending on the circumstances.

Regardless, after `realloc` *only the new returned pointer can be used*.

Assume that the address passed to `realloc` was freed and is now *invalid*.

Always think of `realloc` as a `malloc`, a “copy”, then a `free`.

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.
Be careful using realloc inside of a helper function.

```c
char *double(char * s) {
    int len = strlen(s);
    s = realloc(s, (len * 2 + 1) * sizeof(char));
    for (int i = 0; i < len; ++i) {
        s[i + len] = s[i];
    }
    s[len * 2] = '\0';
    return s;      // this is ESSENTIAL
}
```

A common mistake is to make double a void function (not return the new address for s).

This causes a memory leak if the address of s changes.
Although rare, in practice,

\[
\text{my\_array} = \text{realloc(my\_array, newsize)};
\]

could possibly cause a memory leak if an “out of memory” condition occurs.

In C99, an unsuccessful realloc returns NULL and the original memory block is not freed.

// 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.

```c
char str[81];
int retval = scanf("%s", str);
```

The target array is often oversized to ensure there is capacity to store the string. Unfortunately, regardless of the length of the array, a buffer overrun may occur.

To solve this problem we can continuously resize (realloc) an array while reading in only one character at a time.
// read_str() reads in a non-whitespace string from I/O
// or returns NULL if unsuccessful
// effects: allocates memory (caller must free)

char *read_str(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 == '
') 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 read str

Unfortunately, the running time of read_str 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 oversized arrays, we need to keep track of the “actual” length in addition to the allocated length.
char *read_str(void) {
    char c;
    if (scanf(" %c", &c) != 1) return NULL; // ignore initial WS
    int maxlen = 1;
    int len = 1;
    char *str = malloc(maxlen * sizeof(char));
    str[0] = c;
    while (1) {
        if (scanf("%c", &c) != 1) break;
        if (c == ' ' || c == '
') break;
        if (len == maxlen) {
            maxlen *= 2;
            str = realloc(str, maxlen * sizeof(char));
        }
        ++len;
        str[len - 1] = c;
    }
    str = realloc(str, (len + 1) * sizeof(char));
    str[len] = '\0';
    return str;
}
With our “doubling” strategy, most iterations are $O(1)$, unless it is necessary to resize (realloc) the array.

The resizing time for the first 32 iterations would be:

$$2, 4, 0, 8, 0, 0, 0, 0, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 64$$

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 read_str 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.
This is the **interface** we used in Section 06.

```c
// 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 oversized arrays.

The only differences are: it uses an opaque structure, it provides create and destroy functions, and there is no maximum: it can store an arbitrary number of integers.
// stack.h (INTERFACE)

struct stack;

struct stack * 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 oversized array implementation.

```c
bool stack_is_empty(const struct stack *s) {
    assert(s);
    return s->len == 0;
}

int stack_top(const struct stack *s) {
    assert(s);
    assert(s->len);
    return s->data[s->len - 1];
}

int stack_pop(struct stack *s) {
    assert(s);
    assert(s->len);
    s->len -= 1;
    return s->data[s->len];
}
```
The doubling strategy is implemented in push.

```c
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, 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 pushes and pops.

Languages that have a built-in resizable array (e.g., C++’s vector) often use a similar “doubling” strategy.
Goals of this Section

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

• describe the heap

• use the functions `malloc`, `realloc` and `free` to interact with the heap

• explain that the heap is finite, and demonstrate how to 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
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.

In C, the link in the last node is NULL pointer.

For convenience, we will initially store ints in our linked lists.
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).
Structure definitions

A llist points to the front node (which is NULL for an empty list).

Each llnode stores an item and a link (pointer) to the next node (which is NULL for the last node).

```c
struct llnode {
    int item;
    struct llnode *next;
};
```

```c
struct llist {
    struct llnode *front;
};
```

A C structure can contain a pointer to its own structure type. llnode is a recursive data structure (but llist is not).
There is no “official” way of naming or implementing a linked list node in C.

The CP:AMA textbook and other sources use slightly different conventions.

The structure we present here is often called a “wrapper strategy”, because the llist structure “wraps” around the front of the list.
Creating a linked list

// list_create() creates a new, empty list
// effects: allocates memory

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();
    // ...
}
Adding and creating nodes

// new_node(i, pnext) creates a new linked list node
// effects: allocates memory

struct llnode * new_node(int i, struct llnode * pnext) {
    struct llnode * node = malloc(sizeof(struct llnode));
    node->item = i;
    node->next = pnext;
    return node;
}

// add_front(i, lst) adds i to the front of lst

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.

```c
int list_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 list (**llist**).

```c
int length_nodes(const struct llnode *node) {
    if (node == NULL) {
        return 0;
    }
    return 1 + length_nodes(node->next);
}

int list_length(const struct llist *lst) {
    return length_nodes(lst->front);
}
```
Destroying a list

In C, we don’t have a garbage collector, so we must be able to free our linked list. We need to free every node and the list itself.

When using an iterative approach, we are going to need two node pointers to ensure that the nodes are freed in a safe way.

```c
void list_destroy(struct llist *lst) {
    struct llnode *curnode = lst->front;
    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.

```c
void free_nodes(struct llnode *node) {
    if (node) {
        free_nodes(node->next);
        free(node);
    }
}

void list_destroy(struct llist *lst) {
    free_nodes(lst->front);
    free(lst);
}
```
Duplicating a list

The recursive function is the most straightforward.

```c
struct llnode *dup_nodes(const struct llnode *oldnode) {
    if (oldnode == NULL) {
        return NULL;
    }
    return new_node(oldnode->item, dup_nodes(oldnode->next));
}

struct llist *list_dup(const struct llist *oldlist) {
    struct llist *newlist = list_create();
    newlist->front = dup_nodes(oldlist->front);
    return newlist;
}
```
The iterative solution is more complicated:

```c
struct llist *list_dup(struct llist *oldlist) {
    struct llist *newlist = list_create();
    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;
}
```
Insert into the “middle”

When inserting into the “middle” of a linked list, we need to know \textit{where} to insert.

Consider inserting into a \textbf{sorted} linked list:

\begin{verbatim}
insert( 5, a);
insert(30, a);
\end{verbatim}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{linked_list_insert.png}
\end{figure}

\begin{verbatim}
insert( 5, a);
insert(30, a);
\end{verbatim}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{linked_list_insert2.png}
\end{figure}
// insert(i, slst) inserts i into sorted list slst
// requires: slst is sorted
// effects: modifies slst
// time: \(0(n)\), 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

```c
void remove_front(struct llist *lst) {
    assert(lst->front);
    struct llnode *old_front = lst->front;
    lst->front = lst->front->next;
    free(old_front);
}
```

In Racket, the `rest` function does not actually remove the first element, instead it provides a pointer to the next node.
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;
}
```
Caching information

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 llist structure, so the length can be retrieved in $O(1)$ time.

```c
struct llist {
    struct llnode *front;
    int length;
};
```
Naturally, other list functions would have to update the length as necessary:

- `list_create` would initialize length to zero
- `add_front` would increment length
- `remove_front` would decrement length
- `etc.`
Data integrity

The introduction of the length field to the linked list may seem like a great idea to improve efficiency.

However, it introduces new ways that the structure can be corrupted. What if the length field does not accurately reflect the true length?

For example, imagine that someone implements the remove_item function, but forgets to update the length field?

Or a naïve coder may think that the following statement removes all of the nodes from the list.

```c
lst->length = 0;
```
Whenever the same information is stored in more than one way, it is susceptible to *integrity* (consistency) issues.

Advanced testing methods can often find these types of errors.

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 $O(1)$. 

// 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.

```c
int queue_front(struct queue *q) {
    assert(q->front);
    return q->front->item;
}

bool queue_is_empty(struct queue *q) {
    return q->front == NULL;
}

void queue_destroy(struct queue *q) {
    while (!queue_is_empty(q)) {
        queue_remove_front(q);
    }
    free(q);
}
```
Node augmentation strategy

In 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, 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 $i$. 

![Binary Search Tree Diagram](image-url)
Our BST node (bstnode) is very similar to our linked list node definition.

```c
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.

```c
struct bstnode *new_leaf(int i) {
    struct bstnode *leaf = malloc(sizeof(struct bstnode));
    leaf->item = i;
    leaf->left = NULL;
    leaf->right = NULL;
    return leaf;
}
```

As with lists, we can write tree functions *recursively or iteratively*. 
For the recursive version, we need to handle the special case that the tree is empty.

```c
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.

```c
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.

```c
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 $\text{bst\_insert}$ 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 **unbalanced tree** is a tree with a height that is not $O(\log n)$. The height of an unbalanced tree is $O(n)$.

Using the $\text{bst\_insert}$ 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.

```c
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.

```c
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.
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.

```c
struct bstnode {
    int item;          // key
    char *value;       // additional value (augmentation)
    struct bstnode *left;
    struct bstnode *right;
};
```

```c
struct dictionary {
    struct bstnode *root;
};
```

```c
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.

```c
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 insert is essentially the same:

```c
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.

```c
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.

```c
} else if (node->left == NULL) {
    struct bstnode *new_root = node->right;
    free(node->value);
    free(node);
    return new_root;
} else if (node->right == NULL) {
    struct bstnode *new_root = node->left;
    free(node->value);
    free(node);
    return new_root;
} else // continued...
    // (neither child is NULL)
```

Otherwise, we replace the key/value at this node with next largest key/value, and then remove the next key from the right subtree...
} 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 frees the children and the (string) value before itself.

```c
void free_bstnode(struct bstnode *node) {
  if (node) {
    free_bstnode(node->left);
    free_bstnode(node->right);
    free(node->value);
    free(node);
  }
}

void dict_destroy(struct dictionary *d) {
  free_bstnode(d->root);
  free(d);
}
```
Graphs

Linked lists and trees can be thought of as “special cases” of a graph data structure. Graphs are the only core data structure we are not working with in this course.

*Graphs link nodes with edges.* Graphs may be undirected (i) or directed (ii), allow cycles (ii) or be acyclic (iii), and have labeled edges (iv) or unlabeled edges (iii).
Goals of this Section

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

- use the new linked list and tree terminology introduced
- use linked lists and trees with a recursive or iterative approach
- use a cache 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Dynamic Array</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>item_at</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>search</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>insert_at</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>insert_front</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert_back</td>
<td>$O(1)^*$</td>
<td>$O(1)^{†}$</td>
</tr>
<tr>
<td>remove_at</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>remove_front</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove_back</td>
<td>$O(1)$</td>
<td>$O(1)^{◊}$</td>
</tr>
</tbody>
</table>

* amortized

† requires a back pointer – $O(n)$ without

◊ requires a *doubly* linked list and a back pointer – $O(n)$ without.
Data structure comparison: unsequenced (sorted) data

<table>
<thead>
<tr>
<th>Function</th>
<th>Sorted Dynamic</th>
<th>Sorted Linked List</th>
<th>Regular BST</th>
<th>Self-Balancing BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(h)$†</td>
<td>$O(\log n)$†</td>
</tr>
<tr>
<td>search</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

† 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 **Sets**): self-balanced BSTs or hash tables*.

* A hash table is typically an array of linked lists (more on hash tables in CS 240).
Beyond integers

In Section 10, we presented an implementation of a Stack ADT that only supported a stack of integers.

What if we want to have a stack of a different type?

There are three common strategies to solve this “type” problem in C:

- 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:

```cpp
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 creates new “types” from previously existing types. This is typically done to improve the code readability, or to hide the type (for security or flexibility).

```c
typedef int Integer;
typedef int *IntPtr;

Integer i;
IntPtr p = &i;
```

It is common to use a different coding style (we use CamelCase) when defining a new “type” with typedef.
typedef is often used to simplify complex declarations (e.g., function pointer types).

typedef int ( * MapFn)(int);

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

void array_map(MapFn f, int a[], int len) { // <- cleaner!
    for (int i = 0; i < len; ++i) {
        a[i] = f(a[i]);
    }
}

int main(void) {
    int arr[6] = {4, 8, 15, 16, 23, 42};
    array_map(add1, arr, 6);
    MapFn f = add1;
    array_map(f, arr, 6);
    //...
}
Stack ADT: cleaner interface

struct stack;

// use [Stack] instead of [struct stack *]
typedef struct stack *Stack;

// operations:
Stack stack_create(void);
bool stack_is_empty(Stack s);
int stack_top(Stack s);
int stack_pop(Stack s);
void stack_push(int item, Stack s);
void stack_destroy(Stack s);
Some programmers consider it poor style to use `typedef` to “abstract” that a type is a `pointer`, as it may accidentally lead to memory leaks.

A compromise is to use a type name that reflects that the type is a pointer (e.g., `StackPtr`).

The Linux kernel programming style guide recommends avoiding `typedefs` altogether.
The “typedef” strategy is to define the type of each item
(ItemType) in a separate header file ("item.h") that can be
provided by the client.

// item.h
typedef int ItemType; // for stacks of ints
or...

// item.h
typedef struct posnItemType; // 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.

```c
int i = 42;
void *vp = &i;
int j = *vp; // INVALID
int *ip = vp;
int k = *ip; // VALID
```
While some C conversions are *implicit* (e.g., char to int), there is a C language feature known as *casting*, which *explicitly* “forces” a type conversion.

To cast an expression, place the destination type in parentheses to the left of the expression. This example casts a “void *” to an “int *”, which can then be dereferenced

```c
int i = 42;
void *vp = &i;
int j = *(int *)vp;
```

A useful application of casting is to avoid integer division when working with floats (see CP:AMA 7.4).

```c
float one_half = ((float) 1) / 2;
```
Implementing ADTs with void pointers

There are two complications that arise from implementing ADTs with void pointers:

- **Memory management** is a problem because a protocol must be established to determine if the client or the ADT is responsible for freeing item data.

- **Comparisons** are a problem because some ADTs must be able to compare items when searching and sorting.

Both problems also arise in the typedef approach.
The solution to the **memory management** problem is to make the
**ADT interface explicitly clear** whose responsibility it is to free any
item data: the client or the ADT. Both choices present problems.

For example, when it is the **client’s responsibility** to free items,
care must be taken to retrieve and free every item before a
destroy operation, otherwise destroy could cause memory leaks.
A precondition to the destroy operation could be that the ADT is
empty (all items have been removed).
When it is the **ADT's responsibility**, problems arise if the items contain additional dynamic memory.

For example, consider if we desire a **sequence of 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

// This program reads in strings
// and then prints them in reverse order

#include "stack.h"

int main(void) {
    Stack s = stack_create();
    while(1) {
        char *str = read_str(); // from Sec 10
        if (!str) break;
        push(s, str);
    }
    while(!is_empty(s)) {
        char *str = pop(s);
        printf("%s\n", str);
        free(str);
    }
    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 \( b \)

```c
// 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.

```c
typedef int (*CompFuncPtr) (const void *, const void *);
```
example: dictionary

// dictionary.h (partial interface)

struct dictionary;
typedef struct dictionary *Dictionary;

typedef int (*DictKeyCompare) (const void *, const void *);

// create a dictionary that uses key comparison function f
Dictionary dict_create(DictKeyCompare f);

// lookup key k in Dictionary d
const void *dict_lookup(Dictionary d, void *k);
struct bstnode {
    void * item; // key
    void * value; // additional value (augmentation)
    struct bstnode * left;
    struct bstnode * right;
};

struct dictionary {
    struct bstnode * root;
    DictKeyCompare key_compare; // function pointer
};

Dictionary dict_create(DictKeyCompare f) {
    Dictionary d = malloc(sizeof(struct dictionary));
    d->root = NULL;
    d->key_compare = f;
    return d;
}
This implementation of `dict_lookup` illustrates how the comparison function would work.

```c
const void * dict_lookup(void * key, Dictionary d) {
    struct bstnode * 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.

```c
void qsort(void *arr, int len, size_t size, CompFuncPtr f);
```

The other parameters of `qsort` are an array of any type, the length of the array (number of elements), and the `sizeof` each element.
example: qsort

// see previous definition
int compare_ints (const void *a, const void *b);

int main(void) {

    int a[7] = {8, 6, 7, 5, 3, 0, 9};

    qsort(a, 7, sizeof(int), compare_ints);

    // ...
}

C also provides a generic binary search (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.

```c
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:

```c
int sum_first(int n) {
    int sum = 0;
    for (int i = 1; i <= n; ++i) {
        sum += i;
    }
    return sum;
}
```

generates the following machine code (shown as bytes) when it is compiled on an Intel machine.

```
55 89 E5 83 EC 10 C7 45 F8 00 00 00 00 C7 45 FC 01 00 00 00 EB 0A 8B 45 FC 01 45 F8 83 45 FC 01 8B 45 FC 3B 45 08 7E EE 8B 45 F8 C9 C3.
```

How to compile code is covered in CS 241.
When source code is compiled, the identifiers (names) disappear. In the machine code, only addresses are used.

The machine code generated for this function

```c
int sum_first(int n) {
    int sum = 0;
    for (int i = 1; i <= n; ++i) sum += i;
    return sum;
}
```

is identical to the machine code generated for this function

```c
int fghjkl(int qwerty) {
    int zxcv = 0;
    for (int asdf = 1; asdf <= qwerty; ++asdf) zxcv += asdf;
    return zxcv;
}
```
One of the most significant differences between C and Racket is that C is compiled, while Racket is typically interpreted.

An interpreter reads source code and “translates” it into machine code while the program is running. JavaScript and Python are popular languages that are typically interpreted.

Another approach that Racket supports is to compile source code into an intermediate language ("bytecode") that is not machine specific. A virtual machine “translates” the bytecode into machine code while the program is running. Java and C# use this approach, which is faster than interpreting source code.
Compilation

There are three separate steps required to compile a C program.

• preprocessing

• compilation

• linking

In modern environments the steps are often merged together and simply referred to as “compiling”.

CS 136 Winter 2020 13: Beyond
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.

```bash
$ 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`-0), and to exit the editor, press (`Ctrl`-X).
Create hello.c

1) Create a new folder and a new file:

```bash
$ mkdir cs136
$ cd cs136
$ nano hello.c
```

2) Type in the following program:

```c
#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.

```bash
$ ls
hello.c
```

CS 136 Winter 2020 13: Beyond
We are now ready to compile and execute our program. The most popular C compiler is known as gcc.

```bash
$ 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 .):

```bash
$ ./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.

```c
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 \((\text{argv} - 1)\).
```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.

// 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!
```

To *redirect* output to a file, the `>` symbol is used (*i.e.*, `>` *filename*).
You can redirect input and output at the same time.

```bash
$ ./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.

```bash
$ ./hello Bob | ./swapcase
hELLO, bOB!
```

```bash
$ ./hello DoubleSwap | ./swapcase | ./swapcase
Hello, DoubleSwap!
```
The Seashell environment

We can now understand all of the tasks that Seashell performs.

- scan the “run” file for #includes to determine the required modules, then compile and link all of the modules together
- if “running”: execute while reading stdio from seashell
- if “testing”: for each .in file, execute the program redirecting from the .in file to an output file:
  
  $ ./program < mytest.in > mytest.out

Next, use a comparison program to compare the output files to the .expect files and display the differences

  $ diff mytest.out mytest.expect
Full C language

We have skipped many C language features, including:

- unions and enumerations
- integer and machine-specific types
- switch
- multi-dimensional arrays
- #define macros and other directives
- bit-wise operators and bit-fields
- advanced file I/O
- several C libraries (e.g., math.h)
CS 246

The successor to this course is:

CS 246: Object-Oriented Software Development

- the C++ language
- object-oriented design and patterns
- tools (bash, svn, gdb, make)
- introduction to software engineering
Feedback welcome

Please send any corrections, feedback or suggestions to improve these course notes to:

Dave Tompkins
dtompkins@uwaterloo.ca

Good Luck on your final exams!