Appendix T

This appendix contains additional content covered in Tutorials.

You are still responsible for this content, even if it is not presented in the lectures.

Slides that begin with a star (*) are (mostly) review slides that duplicate content in the lecture notes.
Tutorial 0: Overview of CS 136
Course Personnel

(more details on the course website)

- Instructors: Tim Brecht and Dan Holtby
- Course Coordinators: Travis Bartlett
- (Full-Time) Instructional Support Assistants (ISA)s
  - Jamie Demorim, Nolan Smith and Frank Wang
- Instructional Apprentices (IA)s
  - Rina Wehbe and Kaleb Alway
*Getting help*

- office hours (see website)
- lab hours (Wednesdays, see course website / Piazza for hours and rooms)
- tutorials (Mondays)
- textbook
- piazza

Course announcements are made on piazza and are considered mandatory reading.
CS 136 Student Request System [DEMO]

This is a new system for you to submit requests.

www.student.cs.uwaterloo.ca/~cs136/cgi-bin/requests/

- Remark requests
- Clicker registration or marking errors
- General requests

Do NOT email your requests or post them on piazza (also, never try to submit your assignment via email).

You may contact the course coordinator (illness, accessibility services, confidential feedback) over email:

trbartlett@uwaterloo.ca
Lab Hours

A computer lab is booked exclusively for CS 136 use:

- See Course Website for lab hours and rooms

Course Staff will be present in the lab.
Office Hours

Office Hours will be updated on the course website:

www.student.cs.uwaterloo.ca/~cs136/officeHours

Many will be held in the CS Learning Centre (MC 4065)
Course Website

Course website is:

www.student.cs.uwaterloo.ca/~cs136/

- Assignments
- Marks
- Office hours
- Links to Piazza, Markus, Marmoset, Request System
- Exam Seating
*Piazza etiquette*

- **read** the *official assignment post* before asking a question
- **search** to see if your question has already been asked
- **use** meaningful titles (e.g., “A0q7 where is the secret code?”)
- **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 the staff *may* provide some assistance.
Seashell [DEMO]

http://www.student.cs.uwaterloo.ca/seashell
Marmoset [DEMO]

- **Public** (*simple / basic*) tests
- **vs. Private** (*full / comprehensive*) tests

- A0q4 is *special* - you can see your secret tests immediately
- During peak times, there are often server delays
*Assignments*

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 share and discuss your code with *individuals*, but **not** online or electronically (piazza, facebook, github, email, IM, USB Stick, etc.)
- You can discuss assignment *strategies* openly (including online)
- You can search the Internet for strategies or code examples
If you submit any work that is not your own, you must still cite the origin of the work in your source code.
For **GOLD** questions, **no collaboration** is permitted:

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

You may discuss your code with course staff.

**Academic integrity will be strictly enforced for gold questions.**
Plagiarism

- We have an automatic detection system designed to detect “similar” code
- Don’t share code or test cases
- Keep your eyes on your own code
- Don’t collaborate with other students outside of the bounds outlined in the assignment
Plagiarism - Penalty

- Marked 0 on the assignment
- -5% to your overall grade in the course
- Repeat offenders may face much harsher consequences

All cases are reviewed by course staff before any penalties are applied.
Reminders

- **A0 must be completed** before you can receive any other assignment marks.
- Register your clickers
- Get the course notes (online or through media.doc)
- Read the top threads in Piazza
Appendix A

Appendix A contains additional content, examples and language syntax details that may not be covered in the lectures.

Some of this content will be covered in tutorials (time permitting).

You are still responsible for this appendix, even if it is not presented in the lectures.

- Appendix A.1 - Review of CS 135
Tutorial 1

• Quick review of C Syntax

• Translating Racket to C

• Implementing a ceiling function
#include <stdio.h>

// expt(x,y) calculates x to the power of y
// requires: y >= 0
int expt(int x, int y) {
    if (y) {
        return x * expt(x, y - 1);
    } else {
        return 1;
    }
}

int main(void) {
    printf("expt(4,3) = %d\n", expt(4, 3));
}
The previous program illustrates many C syntax elements.

- `#include <stdio.h>`
- `purpose` statement, `requires` in contract
- use of `{}`, `()`, indentation, `;`
- static `types`: `int x`
- any non-zero value is "true"
- `if`, `return`
- `int main(void)`
- `printf` syntax with "%d"
Racket Conversion

Convert the following functions into C:

```scheme
(define (checkprime n k)
  (cond [(= k 1) 1]
        [(zero? (remainder n k)) 0]
        [else (checkprime n (sub1 k))]))

(define (isprime n)
  (checkprime n (sub1 n)))

(define (nextprime n)
  (cond [ (= 1 (isprime (add1 n))) (add1 n)]
        [else (nextprime (add1 n))]))
```
**Integer Division**

**Implement** the following **interface** in C:

```c
// ceiling(a,b) produces the value of a/b, rounded up to
// the next largest integer
// requires: a >= 0, b > 0
int ceiling(int a, int b);
```
Using the readnat module

`stdio.h` includes input functions as well as output functions, but they’re a bit advanced, so we have create our own simple module `readnat.h` for reading natural numbers.

```c
// TASK: Write a program that reads in an initial natural number n followed by an arbitrary number of natural numbers, and determines if there were more numbers >= n than < n

// it prints out one of 3 possibilities:
// -1 if there is no input (cannot read in n)
// 1 if there were more numbers >= n than < n
// 0 otherwise
```

Use *accumulative* recursion.

*Pro Tip*: Design your tests **before** you code!
Tutorial 2

- More on working with seashell
- Working with the `io_test` module
- More C Syntax Examples
- Working with C Structures and Modules
- Accumulative Recursion & `doubles`
[HOW TO RESET SEASHELL]

[SETTING THE RUN FILE]

• Identify the file with the main you wish to run
  (there can be multiple clients)

[RUN]

• Assertion test clients

• Interactive I/O programs
  (input from keyboard, output to console)

[TEST]

• Automated I/O testing
  (input from .in file(s), compare to .expect)
The I/O Test module

To test C modules interactively you need an interactive test client. This is complicated, so we’ve provided io_test.h to make it easier.

Let’s look at an example from last week

```c
int main(void) {
    add_int_test("isprime", isprime, 1);
    add_int_test("nextprime", nextprime, 1);
    add_stop_test("quit");
    io_test();
}
```

This program creates an interactive program that accepts 3 commands ("isprime", "nextprime", and "quit") and then reads commands until it reaches EOF or the "quit" command.
More C syntax examples

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

static const int my_private_int = 5;

// requires: x > 0
static int my_private_function(int x) {
    assert(x > 0);
    // ...
}
```
// rectangle.h

// A module for working with rectangles in a Cartesian plane
// where all (x,y) points are on an integer grid

#include <stdbool.h>

struct point {
    int x;
    int y;
};

// A structure that uses other structures
struct rectangle {
    struct point top_left;
    struct point bottom_right;
};

// rectangle_perimeter(r) determines the perimeter of rectangle r
int rectangle_perimeter(struct rectangle r);
Accumulative recursion with double

The chemical trumponium is very radioactive and toxic.

It has a half-life of one year. Each year its toxic radiation levels are decreased by 50%.

Write a function to determine how many years are required for levels of trumponium to be reduced to some safe threshold.
!!!!!!!!! WARNING !!!!!

• THINGS ARE BEING MODIFIED AFTER THIS.
Tutorial 3

- Working with seashell
- More C Syntax Examples
- Working with C Structures
- Accumulative Recursion & doubles
- Global variables & mutation
[HOW TO RESET SEASHELL]

[SETTING THE RUN FILE]

- Identify the file with the main you wish to run (there can be multiple clients)

[RUN]

- Assertion test clients

- Interactive I/O programs (input from keyboard, output to console)

[TEST]

- Automated I/O testing (input from .in file(s), compare to .expect)
More C syntax examples

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

static const int my_private_int = 5;

// requires: x > 0
static int my_private_function(int x) {
    assert(x > 0);
    // ...
}
```
Side Effects: printing vs. returning

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

// effects: displays a message
void just_a_side_effect(int n) {
    printf("n squared is %d\n", n * n);
    return; // (optional)
}

// effects: displays a message
int has_both_side_effect_and_return_value(int n) {
    printf("n squared is %d\n", n * n);
    return n * n;
}
```
Reviewing struct syntax in C

Implement a `rectangle` module using the following structures:

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

struct rectangle {
    struct point top_left;
    struct point bottom_right;
};
```
Accumulative recursion with double

The chemical trumponium is very radioactive and toxic.

It has a half-life of one year. Each year its toxic radiation levels are decreased by 50%.

Write a function to determine how many years are required for levels of trumponium to be reduced to some safe threshold.
Global variables & mutation

Implement a simple “guess my number” module that also keeps track of how many guesses you have taken.
Tutorial 4

- C memory model
- The sizeof operator
- Another struct example
- int limits & overflow
C memory model

- Every variable declaration reserves space in memory to store the variable
- The variable’s name refers to the corresponding address in memory

// What would this program look like in memory?
int main(void) {
    int x = 42; // 4 bytes
    double y;  // 8 bytes
}
The `sizeof` operator

Each variable we declare takes up space in memory.

The `sizeof` operator produces the number of bytes required to store a type.

```c
printf("%zd\n", sizeof(int)); // prints 4 in Seashell
```

In Seashell, `sizeof(int)` is always 4. However, you should always use `sizeof` to find the size of a type. Why?

There is an example in Seashell that shows one reason why.
More Fun with Structures

- Complete the implementation of a simple “stopwatch” module.
- Create an assertion-based test client to test your module.
- Create an I/O test client that displays the following message, using the timer functions:

  The time is [1:02:03], which is a new world record!
**int limits**

In Seashell, each `int` is stored in 4 bytes (32 bits).

All `ints` must have a value in the (inclusive) range:

<table>
<thead>
<tr>
<th>INT_MIN</th>
<th>$-2^{31}$</th>
<th>$-2,147,483,648$</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>INT_MAX</td>
<td>$2^{31} - 1$</td>
<td>$2,147,483,647$</td>
</tr>
</tbody>
</table>

It is **impossible** for C to represent a value outside of that range in an `int`. 
Overflow

Overflow occurs when a calculated value would normally lie outside of the range of an `int`.

C cannot store values outside of the `int` range, so the result is an “arbitrary” value inside of the range.

Assuming `a` and `b` are positive, overflow will occur when:

\[ a + b > \text{INT\_MAX} \]

or if they are both negative, then overflow will occur when:

\[ a + b < \text{INT\_MIN} \]

Write a function to detect if overflow will occur in C if both `a` and `b` are positive, without calculating `(a + b)`.
Tutorial 5

- Memory model & control flow
- Loops...

```c
while (!understand(loop_knowledge)) {
    ++loop_knowledge;
}
```
int g = 0;

int foo(int a, int b) {
    ++a;
    g += a;
    int c = a + b;
    c *= g;
    return c;
}

int bar(int d) {
    d = foo(d, 1);
    d = foo(d, 2);
    return d;
}

int main(void) {
    const int e = bar(0);
    printf("e = %d\n", e);
}

while

```c
Statement A;
while (exp) {
    Code Block;
}
Statement Z;
```
*for vs. while*

Recall the `for` syntax.

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

This `while` example

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

is equivalent to

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

Write a program that reads in an arbitrary number of natural numbers and then prints the range of the numbers ($\max - \min$).

If there is no input, the range is 0.
More fun with loops

Caesar Cypher

Write a program that first reads in natural number $N$ from input. Then it reads in characters, offsetting each character by $+N$, and then prints them out.

For example, if $N$ is 1, then ‘A’ would be printed as ‘B’.

It “wraps around” within the printable characters, so if $N$ is 1, then ‘~’ would be printed as space ‘ ’.

Newlines are printed normally.
Draw a circle with the given \texttt{radius} inside of a square box.

\begin{verbatim}
void circle(int radius, char border, char draw);
\end{verbatim}

Loop from \texttt{-radius} to \texttt{+radius} horizontally and vertically and print out \texttt{draw} if the location is inside of the circle, or space otherwise.

\begin{verbatim}
circle(4, '*', '#');
\end{verbatim}

\begin{verbatim}
************
*   #   *
*  #####  *
*  ####### *
*  ####### *
*  ####### *
*  ####### *
*  ####### *
*  ####### *
*   #   *

************
\end{verbatim}
Tutorial 6

- Pointers
- Interactive Programs
Pointers

A **pointer** is a variable that stores a **memory address**.

In Seashell, memory addresses are 8 bytes (64 bits).

So far, their usefulness is limited to:

- passing a pointer to a variable to a function so the function can modify the variable
- passing a pointer to a **struct** to avoid a “deep copy” onto the stack
- function pointers
Pointer syntax examples

Pointer definition:

```c
int *ptr;
```

Storing an address with the address operator (&):

```c
ptr = &i;
```

The dereference (indirection) operator (*):

```c
i = *ptr;
*ptr = 42;
```
Pointer parameters

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

struct rectangle {
    struct point top_left;
    struct point bottom_right;
};

// use const to indicate that a parameter is not modified
bool valid_rectangle(const struct rectangle *r) {
    return r->top_left.x < r->bottom_right.x &&
           r->top_left.y > r->bottom_right.y;
}
```

// r is a pointer to a struct, so use -> operator
// r->top_left is a struct (NOT a pointer), so use . operator
function pointer (seashell)

Complete the eval_range module function that evaluates a function over a given range.
Interactive Programs

You are provided with a module stack.h. Your task is to write an interactive program to test this module. You should support these commands:

- + ARG should push the integer ARG onto the stack.
- - should pop a value off the top of the stack.
- t should print the value on top of the stack.
- s should print the size of the stack.
- q should quit the program.
Any invalid command should cause the program to print "INVALID\n" and quit.

Make sure to write some tests in Seashell to make sure the stack module is thoroughly tested.
Tutorial 7

• scanf

• Arrays

• General Q & A
scanf

Always check the return value of `scanf` to ensure it is “successful” (returns 1).

Three basic forms of using `scanf` (so far):

```c
// read in an integer (ignore leading whitespace)
retval = scanf("%d", &i);

// read in a character (ignore leading whitespace)
retval = scanf(" %c", &c);

// read in a character (including whitespace)
retval = scanf("%c", &c);
```
HEX to Decimal (seashell)

Write a program that continuously reads in HEX values one character at a time, and then prints out their equivalent decimal value.
Arrays

- Array definition & initialization syntax
  
  ```
  int my_array[4] = {1337, 4010, 420, 8675309};
  ```

- Passing arrays to functions & use of `const`
  
  ```
  int array_function(const int a[], int len) {
  ```

- Pointer notation & Array organization in memory
  
  ```
  a[i]  <=>  *(a + i)
  ```

- Fixed-Length Arrays
  
  ```
  int my_array[100] = {0};
  const int maxlen = 100;
  int curlen = 0;
  ```
Array Min & Max (seashell)

Complete the implementation for the minmax module.
Tutorial 8

- Strings
  - arrays of `char`
  - null terminated `\0`
  - `printf` & `scanf` with "%s"
  - string literals
Shrnk (seashell)

Write a function that “shrinks” a string by removing all vowels.
Monster Hunt (seashell)

Implement the functions for the “Monster Hunt” game.
Tutorial 9

- Efficiency
  - Big O notation: use $n$ to represent “input” size
  - loops & iteration ($\Sigma$)
  - recurrence relations
  - don’t forget time of built-in functions

- Intro to Dynamic Memory
  - every `malloc` needs a `free`
  - memory can persist beyond function call
Repeat (seashell)

// repeat(s) generates a new string that repeats s
// for example, "hello" produces "hellohello"
// effects: allocates memory (caller must free)
// time: O(n)
char *repeat(const char *s);
Remove Dups 1 (seashell)

// remove_dups(a, len) generates a new array that
// has all of the elements of a (in any order)
// without any duplicates
// effects: allocates memory (caller must free)
// time: O(?)

int *remove_dups(const int a[], int len, int *newlen);

1. count the number of unique values

2. generate the new array

3. fill in the new array

Analyze the time efficiency.
Remove Dups 2 (seashell)

Implement \texttt{remove\_dups} by sorting the elements first (using the provided \texttt{merge\_sort}).

Re-analyze the time efficiency.
Join (seashell)

// join(words, num_words, sep) generates a new string
// that contains every string in words,
// separated by sep
// effects: allocates memory (caller must free)
// time: O(???)
char *join(const char *words[], int num_words, char sep);
Tutorial 10

- More Dynamic Memory
  - realloc
  - “doubling” strategy for arrays

- Abstract Data Types (ADTs)

- Intro to Linked Lists
// REVIEW: "safe" readstr() with doubling
char *readstr(void) {
    char c;
    if (scanf(" %c", &c) != 1) return NULL; // ignore initial WS
    int maxlen = 1;
    int len = 1;
    char *str = malloc(maxlen * sizeof(char));
    str[0] = c;
    while (1) {
        if (scanf("%c", &c) != 1) break;
        if (c == ' ' || c == '
') break;
        if (len == maxlen) {
            maxlen *= 2;
            str = realloc(str, maxlen * sizeof(char));
        }
        ++len;
        str[len - 1] = c;
    }
    str = realloc(str, (len + 1) * sizeof(char));
    str[len] = '\0';
    return str;
}
A freq-y ADT

In this tutorial, we are going to introduce a new freq ADT that keeps track of how frequently a number has occurred.

For example, in the following sequence:

5, 5, 1, 2, 3, -1, 5, 10, -1, 5, 10000, 5

-1 has occurred 2 times
3 has occurred 1 time
5 has occurred 5 times
etc...
freq has two core operations:

// freq_increment(f, i) increments count of i by one in f
// effect: f is modified
// time: O(n) if this is the first occurrence of i
//       O(logn) otherwise
void freq_increment(struct freq *f, int i);

// freq_get(f, i) determines the count of i in f
// time: O(logn)
int freq_get(const struct freq *f, int i);

How should we implement freq? (discussion)
We are going to use a *sorted* dynamic array:

5, 5, 1, 2, 3, -1, 5, 10, -1, 5, 10000, 5

<table>
<thead>
<tr>
<th>values</th>
<th>-1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>10</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>counts</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Implement the two operations:

// freq_increment(f, i) increments count of i by one in f
// effect: f is modified
// time: O(n) if this is the first occurrence of i
//       O(logn) otherwise
void freq_increment(struct freq *f, int i);

// freq_get(f, i) determines the count of i in f
// time: O(logn)
int freq_get(const struct freq *f, int i);
Reverse (seashell)

Implement a `reverse` function for a linked list

- iteratively
- recursively

Note: you do not need to `malloc` any new nodes
Tutorial 11

More Linked Lists & Trees
A Doubly Linked List (Seashell)

A doubly linked list has been partially implemented.

- write `dll_print`
- write `dll_reverse`
Printing a BST (Seashell)

Print the nodes of a BST in order along with their “index”.