CS 136 Midterm Review Notes

Preamble

This document (build 1.06) will outline my perspective of the pre-midterm material covered in CS 136 during the summer semester of 2013. It is not aimed to replace lectures or your own studying, it is aimed to be a study aid that you can refer to. These notes will be provided under no warranty of accuracy or maintainability. If you are confused on a topic, refer to the actual course material, Piazza or meet with your professor for a further (and more accurate) explanation of a topic.

My review notes were written during the University of Waterloos' summer semester offering of CS 136. It covers only pre-midterm material (modules 1 - 5). The professors during this time were: Dave Tompkins, Olga Zorin and Patrick Nicholson.

The course material that influenced the contents of these review notes are:

- Lecture slides designed by the professors.
- "C Programming a Modern Approach - Second Edition", textbook for the course.
- CS 136 on Piazza.
- and other third party websites used to improve my understanding.

I (Jacob Pollack, student at the University of Waterloo) created this review notes document. The material this document contains are my review notes for the midterm. I went through every page in the lecture slides and assigned readings to highlight what I think is important. That highlighted information can be found here, there will be strong paraphrasing with a mix of my own perspective of the material.

To preserve the chronological order that the material was taught in, my review notes are written in the same order. I may have switched around material to introduce it sooner or later to have a different flow. I may have also added or removed content that I felt needed an extra emphasis or was not too important.

I have included some exercises for topics I thought my peers (including myself) struggled on. To determine these topics I lurked Piazza for a couple hours to find the most common help posts. These exercises are found after the review (page 47).

I am in no way, shape or form claiming these notes are accurate. They are my perspective of the pre-midterm material. These were designed to test if I knew the material through attempting to explain it.

Disclaimer

You can reach me at my student e-mail **jpollack@uwaterloo.ca** or jacobpollack@jamach.com.

Special Notices

This document would not be possible without the offering of CS 136 at the University of Waterloo. It is truly an amazing course and I am absolutely loving it. I would like to make a note on peers that contributed towards the result of this review package. Special thanks to:

- Jacob Willemsma.
- Nimesh Das.

Table of Contents

1 Introduction: Module 01

This course is described by the University of Waterloo as follows *"... 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."*. I do not think this will be asked on the midterm, however I felt that this should give you a brief understanding of what we are expected to know by the end of this course.

In the prerequisite course(s) – CS 135, CS 115/116, the focus was programming in a functional paradigm. Programming in a **functional paradigm** is the process of evaluating code as a sequence of Mathematical functions, avoiding mutation and state (no not X-Men mutation). In this course we are required to program in both a functional paradigm and in an **imperative paradigm**. There is a third paradigm called declarative programming, however we do not have to worry about it and hence it will not be referenced again in my review.

Now you are probably asking yourself, what is the **difference between programming in a functional and imperative paradigm**? We now know that programming in a functional paradigm avoids mutation and state. Hence we commonly achieve our goals through recursion. However in an imperative paradigm, we use mutation and change the state of our program frequently – giving us a lot more freedom to accomplish a given task. Here are two examples of summing the numbers from [0, n]. The first example is programming in a functional paradigm, however the second example is programming in an imperative paradigm.

```
int sum( int n ) {
  return ( 0 == n ) ? n : ( n + sum( n - 1 ) );
}
```

```
int sum( int n ) {
  int acc_sum = 0;

  for ( int i = 0; i <= n; i++ ) {
    acc_sum += i;
  }

  return acc_sum;
}
```

For the time being we will be required to program in a functional paradigm. We will be visiting imperative programming in module 05.

1.1 Full Racket

It is true! We are finally old enough (as Dave says) to use full Racket. To enable full Racket, you should have the following declaration at the top of every Racket file.

```
#lang racket
```

Keep in mind there is some funky terminology for Racket. We **apply** functions, which **consumes** arguments and **produce** a value. In the following example, **x** and **y** are **parameters**.

```
(define (f x y)
  (sqr (+ x y)))
```

Recall what a **constant** is? When you declare a constant, it is an immutable identifier in Racket (however not always). Constants are quite useful when it comes to readability and maintainability (which will be discussed later). An example:

```
(define penny 1)

(define (value-of-n-pennies n)
  (* n penny))
```

But that does not make sense... there is only one money penny :p (joking)

In Full Racket we can now declare functions without parameters. A keen observer will notice that their declaration is very similar to the declaration of a constant, however do not fool yourself. They are very different! An example:

```
; Declaring a constant.
(define some-constant 9)

; Declaring a function without parameters.
(define (some-function) 9)

; Something wacky.
(define (do-something-wacky n)
  (+ n (* some-constant (some-function))))
```

Recall what a **top-level expression** is from CS 115/135. They are expressions that are evaluated at the top-level when you run your Racket program. Racket will display the result of each top-level expression in the console window. Here is an example of everything we have learned thus far in Racket (excluding structures):

```
#lang Racket ; my-file.rkt

; Some top level expression.
(+ (+ 3 3) 3) ; => 9
(* 2 2)       ; => 4

; Declaration of a constant.
(define x 9)

; Declaration of functions.
(define (no-param-func)
  ...)

(define (some-func x y z)
  ...)
```

There are a few more things we should be able to recall: booleans, symbols, strings, characters, logical operators (and ... or) and conditional statements. I will not be going over this as they are pretty trivial. **Fun fact**, 0 is not false in Racket. However there is a new notation we are introduced to for condition statements. The old and newer notation is provided below:

```
; Old notation (still useful).
(cond
  [q1 a1]
  [q2 a2]
  [else a3])

; New notation.
(if expression answer-true answer-false)
```

Structures in Racket have always been tricky for me to syntactically remember. With the introduction of full Racket it has become even harder. Originally structures could be output to the console and it would display their contents as-is. However in full Racket structures are not displayed to the console without a special keyword, **transparent**. An example of how to declare a structure posn with the older and newer syntax:

```
; Older syntax.
(define-struct posn (x y))
```

```
; Newer syntax.
(struct posn (x y) #:transparent)
```

You should be able to recall what a list is in Racket. There are two new functions we are permitted to use: **last** and **drop-right**. In full Racket the function **member** behaves differently. If an element, e is a member of the list, it will produce the tail of the list starting at the first occurrence of e, otherwise false. To show an example:

```
(member 9 (list 1 3 9 27 81)) ; => '(27 81)
(member 5 (list 1 3 9 27 81)) ; => #f
```

Another feature full Racket that is mainly for convenience is, we are able to **implicitly declare local functions and constants**. An example will not be provided.

1.2 Binary Search Trees (BST)

No unfortunately these are not going away. I know, take a moment if you must and then return to this review... :(

Binary Search Trees (henceforth referenced as BSTs) will be an integral part of our experience as a CS student at the University of Waterloo. To review some terminology, the first **node** is called the **root node**. Each node points to two **child nodes**. The left child node must be less than or equal to the index of the parent node and the right child node must be greater than or equal to the index of the parent node – this is known as the **ordering property**. If a node does not have a left or right child node, it is commonly referred to as a **leaf**. Below are a couple examples of BSTs.

```
; Declaring the structure of a BST.
(struct bst-node (key val left right) #:transparent)

; Declaring a BST.
(define bst (make-node 10 "" (make-bst-node 8 "" empty empty)
                             (make-bst-node 12 "" empty empty)))
```

Since BSTs can be very annoying and hard to understand, I provided another example of an implementation for a function that will produce the true or false if a given key k is in a given BST *abst*. The declaration of the BST structure is omitted for the example.

```
; Declaring the structure of a BST.
```

```
(struct bst-node (key val left right) #:transparent)

(define (exists-key? k abst)
  (cond
    [(empty? abst) #f]
    [(equal? k (bst-node-key abst)) #t]
    [else (or (exists-key? k (bst-node-left abst))
              (exists-key? k (bst-node-right abst)))]))
```

1.3 Abstract List Functions and Lambda

You should be able to recall what an **abstract list** function is from your experience in CS 135 or CS 115/116. For convenience I will list the ones we have been using often on assignments (and will likely be needed for the midterm). Some abstract list functions include: *filter*, *build-list*, *map*, *foldr* and *foldl*.

These next few paragraphs and examples are going to outline my favorite feature in Racket. The feature is called an anonymous function, more commonly referred to as **lambda**. This is an extremely, exxtremely, exxxtremely powerful feature in Racket. An anonymous functions allows us to declare functions without names. This may not be obvious for why it is useful at first, however take a look at these uses of lambda.

```
; Declaring a sample list.
(define sample-list '(1 2 3 4 5 6 7 8 9 10))

; Some uses of lambda.
(filter (lambda (x) (and (even? x) (> x 5))) sample-list) ; => '(6 8 10)

(build-list 5 (lambda (x) (* x x))) ; => '(0 1 4 9 16)
```

1.4 Documentation (Design Recipe)

This is a very tedious but necessary part of designing successful software. I am almost certain we will be tested for how well we document our code on the midterm (unless it specifies to omit this step due to time constraints). From assignments 1 - 5 we have only had a few hand marked questions, however documentation is required none the less.

The main goal of the design recipe is to aid in communicating what your function should consume, what it will achieve and what are its constraints. Once we get further into modularization you will see it is extremely important to maintain well documented code. Recall from CS 115/135 that there were quite a few elements to the design recipe. In CS 136 we only need to worry about 4. These elements are referred to as the **contract**, **purpose** and **pre** and **post conditions**.

Recall the sum function I wrote on page 2 of this document? If it were in Racket, the design recipe would be as follows:

```
; (sum n): Int -> Int
; Purpose: Consumes an integer, n and produces a value greater than
;          or equal to 0. This value will be the sum from 0 ... n.
; PRE: n >= 0
; POST: produces an integer >= 0
```

2 Modularization: Module 02

In CS 136 I believe we will be expected to master the concept of modularization. A majority of our assignments have involved modularization – not only in design but as an approach to solving problems. That being said, a fair question to ask is, "what is modulization?".

In Computer Science, **modularization** is the process of separating functionality from a program into smaller, independent and interchangeable modules such that each module has a well defined purpose. Without modularization it would be extremely difficult to work in groups on a project and re-use code between programs.

There are a few key terms motivating the push for modularization. Those being, **re-usability**, **maintainability** and **abstraction**. Let's look at what these terms mean.

- Re-usability is the process of taking existing modules from other projects and applying them to future products. For example creating a module that handles MySQL database interaction can be applied to multiple projects.
- Maintainability is the process of separating the project into multiple modules with a specific purpose so that multiple programs are able to work on different aspects of the same project at the same time. Example, if one piece of the project needs an update, a programmer can easily grab that module and update it then put it back into the project without disrupting anything.
- Abstraction is the process of using a module designed by another programmer without actually knowing how it does what it does. For example, a professor using a projector to teach Math. The professor has no idea how the projector works however he is using it anyway.

Here are a couple more real world examples of modularization.

1. **Maintainability**: Going into the garage to repair your cars radio. Instead of replacing the entire car you are only replacing the radio module.
2. **Abstraction**: Putting triple A batteries into your flashlight.

2.1 Modularization, How?!

The most important aspect of modularization is the interface. The **interface** of a given module consists of a collection of functions (function definitions) that are accessible outside of the module (public), as well as the documentation for those given functions.

These terms that I am about to define have had me bogged down for a while however after writing this review pack I have finally understood their meaning. For modularization to be successful, it is important to achieve two things.

1. High Cohesion.
2. Low Coupling.

When one refers to **high cohesion**, it means that all the functions within a given module are working collectively towards a common goal or purpose. If there was a module designed to make craft dinner and there was a function that ordered a coke zero from the vending machine, it would not contribute towards the common goal of the module and hence would lead to low cohesion.

The term **low coupling** means that there are as few modular inter-dependencies as possible. This term becomes really important in re-usability. If you have a module referencing many functions from another module, you will need to copy both modules over if you plan to re-use it. Maximized low coupling is when a module can be re-used without requiring additional modules to be copied.

A module with low cohesion and high coupling has a very poor design. Be sure to stay away from that!

2.2 Information Hiding and Interface Documentation

It is important to hide information from the client from time to time. For example, you are hired to design a module that will act as an ATM for TD Canada Trust. You will want to hide some implementation functionality to avoid the client from accessing functions that would otherwise alter important information. This is referred to as security.

In Racket, when you implement a function it is automatically hidden from the client (private). To allow the client to access a given function, you must *provide* it in your module (public). An example of a module that will produce my favorite color:

```
#lang racket ; fav-color.rkt

; Provides the appropriate functions to the client.
(provide fav-color?)

; (fav-color? color): Symbol -> Boolean
; Purpose: Consumes a symbol, color and produces true if it is
;          my favorite color, otherwise false.
; PRE: true
; POST: returns a boolean

; ==========================================================

; Declaring my favorite color (private).
(define fav-color 'blue)
```

```
; (fav-color/helper): ? -> ? (private).
; Purpose: ?
; PRE: ?
; POST: ?
(define (fav-color?/helper)
  ...)

; See interface above (public).
(define (fav-color? color)
  (symbol=? color fav-color))
```

Observe the documentation. When documenting a module, all documentation must be put at the top of the module under the provided line followed by some delimiter (a lot of equal signs) – this is where the interface is. You can document multiple functions in the interface, simply separate them with a space from the one above it.

2.3 Using Racket Modules

To use a Racket module you must *require* the module. The *require* special form will stop what it is doing, start executing the module and then continue doing what it was doing in the client module.

```
#lang racket ; client.rkt

; Requiring the favorite color module.
(require "fav-color.rkt")

; Using the functions provided in the favorite color module.
(fav-color? 'green) ; => #f
(fav-color? 'orange) ; => #f
(fav-color? 'blue) ; => #t

(fav-color?/helper) ; => error, not provided.
```

2.4 How to Test Racket Modules

In full Racket there is no such thing as *check-expect*. So, how do we test our modules? To test our modules we are expected to create a separate client testing module that will perform all of our testing. If you want to test a private function, simply add it to the *provide* tag and then remove it before you submit your work. Here is an example client testing module for the favorite color module:

```
#lang racket ; client-test.rkt
```

```
; Requiring the favorite color module.
(require "fav-color.rkt")

; Performs some testing.
(equal? (fav-color? 'green) #f)
(equal? (fav-color? 'blue) #t)
```

3 Functional C: Module 03

If you recall, our transition from Racket to C was initially very easy. We were taught how to do functional C (program in the functional paradigm). This section is just going to review some of the important transition details. I will not be reviewing C99 syntax.

Keep in mind we are using C99! There are quite a few versions of C out there. The rest of this review will be implying the C99 standards.

3.1 Typing

I know this subsection probably got you excited. No, we are not learning how to type. We are learning what data types are in C. In Racket we had functions such as *integer?* and *string?* that would let us determine the data type of a constant variable (yes it is weird in Racket, constant variables). In C, all data types are known in advanced and must be a part of the declaration.

An interesting fun fact is, Racket uses dynamic typing whilst C uses static typing (ignore the mention of static, it will occur frequently with multiple meanings).

```
; Declaring Racket constant variable.
(define n 9)
```

```
// Declaring C constant.
int const n = 9;

// ... or alternatively.

const int n = 9;
```

The C99 standard says that *const* will be applied to the identifier at its left. If there is nothing to its left, then it will apply to the identifier at its right. Both methods work however it is better to declare it using the first method[1].

There are a couple styles to naming such as the underscore style and the camelCase style. In CS 136 they are expecting us to use the underscore style however as long as we are consistent and it is readable then you should be "OK".

[1]"the second lady may be who you want to date, but the first lady is better for you" - Dave.

An important thing to note is that there is no C equivalent to a *'symbol* in Racket. The closest C has to a symbol is an *enum*, however these will not be required in CS 136. No example will be provided.

3.2 Operators

In Racket the +, -, / and * were all functions. In C, these are called operators. There are a lot of operators in C, it will be extremely hard to memorize them all including their order of operations. When in doubt, use parenthesis! Here are a couple examples of the use of operators in C.

```
int const a = 1 + 1; // => 2
int const b = 4 - 2; // => 2
int const c = ( ( 8 / 2) - 2 ); // => 2
int const d = 10 % 8; // => 2
```

3.3 Terminology

In Racket we mentioned that you apply functions, they consume arguments and produce a value. In C, you **call** a function, they are **passed arguments** and **return** a value.

Make sure you have this terminology down! It is important when writing your documentation.

3.4 Function Definitions

Unlike Racket, there are a few more details you will need to provide in your function definition. Functions in C must have a specified return data type and every parameter must have a specified data type. If I were to define the following function in Racket:

```
; (sum n): Int -> Int
; Purpose: Consumes an integer, n and produces a value greater
;          than or equal to 0 that is the sum from 0 ... n.
; PRE: n >= 0
; POST: produces an integer >=0
(define (sum n)
  (if (equal? n 0) n (+ n (sum (sub1 n)))))
```

... it would be translated to the following function in C:

```
/**
```

```
 * sum( n ): Is passed an argument, n and returns a value greater
 *           than or equal to 0 that is the sum from 0 ... n.
 * PRE: n >= 0
 * POST: returns an integer >= 0
 */
int sum( int const n ) {
  return ( ( 0 == n ) ? 0 : ( n + sum( n - 1 ) ) );
}
```

Observe that the *return* control flow statement is required in every function in C (for the time being). It means, stop executing the function and return this result. Also, note the braces { }. It is required to wrap a function body in braces in C. They indicate the beginning and the end of the functions' scope (also known as the function block).

3.5 Scope

In C it is very important to be aware of scope. Scope in C is consistent with scope in Racket, with a few new complexities. Right now we are introduced to 3 types of scope, global, local and block – each of which is demonstrated below:

```
int const g = 9; // Global scope.

int some_func( int const p ) { // Local scope.
  int const l = 9; // Local scope.

  {
    int const l = 10; // Block scope.

    return l;
  }
}
```

You should avoid cases like these (using block scope) however C will evaluate the innermost brace. The difference between Racket and C is that by default all functions and constants have global scope (public). For constants we require an extra keyword to use those constants outside of a given module. Say we had the following module.

```
// Module A (.h)

// Function declaration (global scope).
int sum( int n );
```

```
// Module A (.c)

// Global constant (global scope).
int const a = 9;

// Function definition.
int sum( int n ) {
  return ( ( 0 == n ) ? n : ( n + sum( n - 1 ) ) );
}
```

To use *a* we would need to properly call it. This is done using the *extern* keyword.

```
// Module B

// Preprocessor directive to include module a.
#include "module_a.h"

// Global constant (global scope).
extern int const a;

// Main function.
int main( void ) {
  int sum_of_a = sum( a );

  return 0;
}
```

To have **modular scope**, you must add the *static* keyword in the declaration of a constant or function. This will restrict access to the constant or function to the module it was declared in

```
static int const a = 9; // Modular scope.

// Modular scope.
static int sum( int n ) {
  return ( ( 0 == n ) ? n : ( n + sum( n - 1 ) ) );
}
```

Last thing to know in C is that, if you recall in Racket we had top-level expressions. In C there is no such thing and you will not be able to run your program if you have them. This was a feature only in Racket.

```
int const a = 5; // OK
int const b = 4; // OK

(a + b); // NO! Do NOT do this!

// ...
```

3.6 Booleans and Logic Operators

In C booleans are 0 and 1 where 0 is false and 1 is true. Keep in mind any non-zero value is considered true in C. For checking equivalency, it is a common mistake to use one equals (the assignment operator) as oppose to two equals (the equivalence operator). In C it is two equals!

```
// The value of ...
( 9 == 8 ) // => false
( 9 == 9 ) // => true
( 2 == 9 ) // => false
```

The use of not, and and or in Racket is translated to !, && and || in C. Note the double & and | for and and or.

```
// The value of ...
!( 9 == 8 ) // => true
( 1 && 1 ) // => true
( 0 || 0 ) // => false
```

It is very important to keep in mind that C will short-circuit, similar to Racket, and stop evaluating an expression when a value is known. This can become very vital in complex code, such as the following (ignore everything - yellow box in review).

```
// The second statement will never get reached if the first statement
// is false and hence cause an issue with the incrementing of 'count'.
if ( ( some_var == true ) && ( ++count == another_var ) ) {
  // ...
}
```

Other operators in C include >=, <=, > and <. The last operator we need to be familiar with is the ? operator. It will return a value based on an expression. We use this similar to a *cond* statement in Racket. It is called a ternary operation in C.

```
int const a = 10;
int const b = 9;

int const max_val = ( a > b ) ? a : b;
```

We can also nest ternary operations in C similar to how we can nest *cond* statements in Racket. No example is provided.

3.7 Recursion

Recursion behaves exactly as one would expect in C. Take a look at these two examples. These are the implementations for the sum from 0 ... n using recursion in Racket and C.

```
(define (sum n)
  (if (equal? n 0) n (+ n (sum (sub1 n)))))
```

... and the following in C:

```
int sum( int const n ) {
  return ( ( 0 == n ) ? n : ( n + sum( n - 1 ) ) );
}
```

3.8 Function Definition vs Declaration

Just a quick note on this. The difference between declaring a function and defining a function is that you are not including the body of the function in a declaration. In C you are required to declare functions before you use them. You can have the definition anywhere in your module. Here are a few examples of what to do and not to do.

```
// Function declaration (includes ';').
int sum( int const n );

// Function definition (includes block scope).
int sum( int const n ) {
  return ( ( 0 == n ) ? n : ( n + sum( n - 1 ) ) );
}
```

It is important to note that you do not have to reference the parameter name in the declaration. However it is good practice to do so! This would be invalid:

```
// Function declaration (includes ';').
int sum( int const n );

// Function definition (includes block scope).
int sum( int const n ) {
  return ( sum_helper( n ) ? n : ( n + sum( n - 1 ) ) );
}

// Function definition (includes block scope).
int sum_helper( int const n ) {
  return ( 0 == n );
}
```

The declaration for *sum_helper* must be placed above where you plan to reference it in your program. This would be the correct implementation:

```
// Function declaration (includes ';').
int sum( int const n );

// Function declaration (includes ';').
int sum_helper( int const n );

// Function definition (includes block scope).
int sum( int const n ) {
  return ( sum_helper( n ) ? n : ( n + sum( n - 1 ) ) );
}

// Function definition (includes block scope).
int sum_helper( int const n ) {
  return ( 0 == n );
}
```

3.9 C Interface Files

In C we use interface files to separate our implementation (.c) from our interface (.h). An interface file will contain all declarations of functions that the module wishes to let others' use as well as all the appropriate documentation. Creating interfaces are also very efficient as all of the declarations and documentation will only need to be in one place. Here is a good example with all the material we have covered up to now:

```
// Implementation for Module A (.c).

#include "module_a.h" // This will be discussed below.

int const g = 9; // Constant with global scope.

static int const m = 9; // Constant with modular scope.

// Function declaration (modular scope).
static int sum_helper( int const l );

// Function definition with global scope and one parameter (local scope).
int sum( int const l ) {
  return ( ( sum_helper( l ) ? l : ( l + sum( l - 1 ) ) );
}

// Function definition with modular scope and one parameter (local scope).
static sum_helper( int const l ) {
  return ( 0 == l );
}
```

The interface for the module A will look as follows. Note documentation would be placed in this file:

```
// Interface for Module A (.h)

// Provides the constant g (global scope).
extern int const g;

// Provides the sum function (global scope).
//
// Documentation would go here.
int sum( int const n );
```

Did you notice the use of *#include*? This is a preprocessor directive. It will literally "copy" the contents of the interface file and "paste" it into your file. This is however hidden from us and done before compiling occurs. Now you should see why interface files drastically reduce the amount of code we will need to write and maintain.

Note *#include* is very similar to Rackets *require*, however it will not execute your code. It will simply copy and paste your code.

3.10 Standard Modules

Unlike Racket, there are no built in functions in C. All that we have in C are operators. There are a lot of libraries that C provides that we can include in our modules. These libraries are called standard libraries. Note the difference when including a standard library and an interface for a module we created.

```
// Standard C library.
#include <somemodule.h>

// Our module interface.
#include "somemodule.h"
```

Some standard libraries that we have used thus far in CS 136 include: assert, stdbool, stdio, stdlib and limits. You can research each library if you are unsure on the functionality it provides (I will not be going over this).

3.11 Main

In C you will not be able to "run" your modules. The only way to run your modules is to have a function called *main*. Every C program must have a function called *main*, otherwise the operating system does not know where to start executing your program! This is the rare occasion that I will review the syntax for something in C, however the syntax is as follows:

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

  return 0; // or 1, OS flags. (optional)
}
```

To break down this definition,

- *void* is used to declare that main accepts no parameters.
- *Main* has an *int* return type however it does not return anything.
- The operating system invokes *main*. It returns 0 upon success, another integer upon failure.
- The return is not necessary. This is a special situation for *main*.

For testing we are expected to use *main*. You can use the functionality from the assert and stdio standard libraries to create an interactive testing module. Here is an example of a testing module for my sum module.

```c
// testing module for sum.

#include <assert.h>
#include <stdio.h>

#include "sum.h"

int main( void ) {
  printf( "Initializing some tests... " );

  assert( sum( 10 ) == 55 );
  assert( sum( 0 ) == 0 );
  assert( sum( 3 ) == 6);

  printf( "done!\n"); // the '\n' character means new line.

  return 0;
}
```

3.12 Standard IO Library

There are a lot of features with the *printf* function alone that I will dedicate a section of my review to this. This library is very powerful. Here are a few examples:

```c
// Some module.
#include <stdio.h>

int const my_age = 18;
int const student_id = 20123456;

int main( void ) {
  // %d is a placeholder for an integer.
  printf( "My age is... %d\n", my_age );

  // %x is a placeholder for a hexademical number and 08 means it must
  // be 8 characters with zero as padding (ie 0xff => 0x000000ff).
  printf( "My student ID is hexademical is... 0x%08x\n", student_id );

  // Multiple placeholders.
  printf( "My age is... %d, I will be %d years old in 2 years.\n",
          my_age,
          my_age + 2 );

  return 0;
}
```

There are a numerous amount of placeholders that you can use. If you are further curious, Google *printf* placeholders and there will be a nice table displaying all of them.

3.13 Structures

Structures in C are very similar to Racket with a few subtle but important differences. Here is an example of a *posn* structure in C.

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

const struct posn p = { 1, 2 };

int const p_x = p.x; // => 1
int const p_y = p.y; // => 2

// or alternatively...

const struct posn p = { .y = 2, .x = 1 };

int const p_x = p.x; // => 1
int const p_y = p.y; // => 2
```

You should be aware on how to check if structures are equivalent. If *p1* and *p2* are structures you cannot to *p1 == p2*. Similarly to tie this into modularization, if you would like to make a structure available to anyone, simply add it into your modules' interface file. An example:

```
#include <stdbool.h>

bool is_posn_equal( const struct posn p1, const struct posn p2 ) {
  return ( ( p1.x == p2.x ) && ( p1.y == p2.y ) );
}
```

4 C Memory Model: Module 04

This section is probably one of my favorite sections of this course. I will go into a lot more detail than we need to, mainly because I enjoy the content. Feel free to skim through this, I will label important information.

Now, having grown up in the 21st century computers became an integral aspect of my life. Both my parents are Computer Science majors and grew to be quite successful. If you cannot already guess, I have been using computers before I could walk. There are a few things you should know about how a computer operates (not only for the midterm but just to know...).

Computers measure their memory capacity in **bytes** (8 bits). A **bit** is a **bin**ary dig**it** that is either on or off, more commonly known as 0 or 1. These are just units of measurement, similar to milometers, centimeters, meters, kilometers, etc. however they are specific to the world of computers.

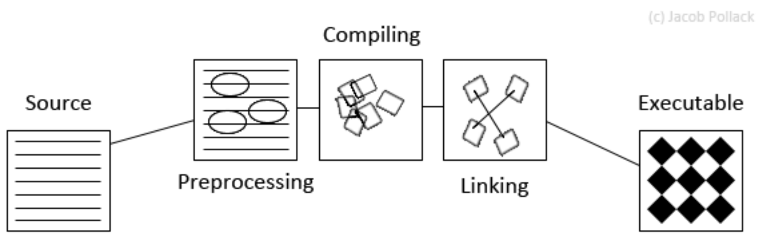
4.1 Terminator (Machine Code)

I know everyone has been dying to reach the terminator section (double pun?). Unfortunately Arnold will not be making an appearance. In the context of computers, machine code is a representation of your source code that the processor is able to interpret and execute. To "run" a program written in C, it must be compiled into machine code.

To "run" your program written in C, it must undergo a three phases. These phases are known as **preprocessing**, **compiling** and **linking**.

- Preprocessing is the process of scanning all of your source files for preprocessor directives. When a directive is found it will perform the appropriate action. The only directive we need to know for this course is the *#include* directive.
- Compiling is the process of turning your readable C source code into machine code (output as an object file). This file is compiled but not linked and hence the next phase.
- Linking is the process of grabbing all of those object files and linking them to a single executable.

The following figure will illustrate how this works.



4.2 More Operators

I know, I know. There are so many operators! Well blame CS 136 for introducing so many. The following operators are very important when it comes to memory.

We were introduced to a *sizeof* operator that will determine the numbers of bytes to store a type. Keep in mind this operator is a compile-time operator and not a run-time operator. Example (for a 32-bit environment),

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

We were also introduced to an address operator, & that will return the address in hexadecimal of an identifier in memory. Note the single & as oppose to the double && to represent *and*.

```
const int g = 9;

&g; // => address of g (&g).
```

4.3 Hexadecimal, Decimal and Binary Notation

In Computer Science we will often represent numbers in their hexadecimal, decimal and binary representations. It is important to understand what each form is and how to convert from one form to another. Let's begin with decimal.

Decimal notation is a way to represent a value in base 10. This notation should be very common to you as this is how we typically represent our values. We have 10 fingers and thus base 10 consists of the characters 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. With these 10 identifier (hence base 10) we are able to represent any value. For example,

$$1204 = (1) \times (10^3) + (2) \times (10^2) + (0) \times (10^1) + (4) \times (10^0)$$

Hexadecimal notation is very common when referring to memory addresses in C. Nowadays your computer will likely have a 32-bit CPU or a 64-bit CPU. This is referring to the amount of memory addresses your computer can have. In a 32-bit environment we see that your computer can have 2^{32} memory addresses. Similarly in a 64-bit environment we see that your computer can have 2^{64} memory addresses. On the uBuntu VPS in CS 136 it uses a 32-bit environment and hence all mentions of data types will be with respect to a 32-bit environment. Now, back to hexadecimal. Hexadecimal is base 16 and consists of the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F where A is 10, B is 11, C is 12, D is 13, E is 14 and F is

15 (base 10). Here is an example of the breakdown of a hexadecimal number,

$$\text{0x00FFAE19} = (15)\times(16^5)+(15)\times(16^4)+(10)\times(16^3)+(14)\times(16^2)+(1)\times(16^1)+(9)\times(16^0)$$

If you are a keen observer, you will notice that conversion between binary and hexadecimal is extremely easy since each hexadecimal character can be represented in 4 bits. Furthermore, two hexadecimal numbers can be represented in 8 bits. Example,

0xFF = 11111111 in binary.

This may not seem useful now, however it will become apparent once I start going over the review for how we store things in memory (it is probably the best chapter of this course).

Lastly, binary notation. It is extremely common when referring to the value that a memory address points to. An example of representing a value in binary is above (15 in hex is represented by 1111 binary).

Keep in mind that the following holds in C. Notice the use of 0x for hexadecimal.

```
int const a = 0xf;
int const b = 15;

( a == b ) // => true
```

4.4 Memory

Since a bit is such a small unit of measure. It became a standard that the smallest unit a Computer will use is a byte. Bits are still important, as every unit is measured in bits. However you will rarely (never in this course) need to access an individual bit. Here are some common units that we should be familiar with:

- Byte = 8 bits, 2^8 possible values.
- KiloByte = 1024 bytes.
- MegaByte = 1024 kilobytes.
- GigaByte = 1024 megabytes.
- TeraByte = 1024 gigabytes.

In your computer you have something called **primary memory** (RAM) and **secondary memory** (hard drive, flash drives, etc.). Your primary memory is extremely fast in comparison to your secondary memory, however secondary memory is much cheaper to buy. I do not think we will be asked a question about this however we should know the difference.

Programs are launched into primary memory (random access memory) and disappear when the program is exited or when the computer shuts down. Secondary memory will remain persistent and hence often be used for storage.

I mentioned earlier the term **memory address**. A memory address is an identifier for one byte of memory. To access this byte you need to know its' address. Dave had an example in class where to call a friend, you need to know their phone number. You cannot call them without knowing their phone number, however when you have their phone number you can call them when ever you want. Memory is managed similarly with a few subtle differences. Memory addresses are typically represented in hexadecimal notation with the first byte as 0x00000000 and the last byte as 0xFFFFFFFF (on a 32-bit machine, remember all references are to a 32-bit machine) rather than decimal notation (like a phone number would be).

Now that we understand the bare bones of how memory works, let's recall how we declare a constant in C.

```c
#include <stdio.h>

// Declaring a constant integer, a.
int const a = 9;

int main( void ) {
  printf( "The value of a is %d.\n", a ); // => 9
  printf( "The address in memory of a is 0x%08x.", &a ); // => 0x000002FF

  return 0;
}
```

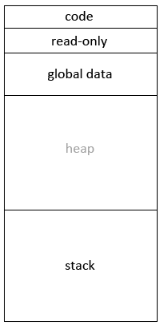
Notice that the value of a is 9, however the memory address of a is 0x000002FF. This means that the bit representation of 9 is stored in memory addresses 0x000002FF, 0x00000300, 0x00000301 and 0x00000302. Recall an integer is 4 bytes and hence it will require 4 consecutive memory addresses. Those blocks of memory will look as follows.

$$\begin{aligned}
\text{0x000002FF} &= [\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ ] \\
\text{0x00000300} &= [\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ ] \\
\text{0x00000301} &= [\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ ] \\
\text{0x00000302} &= [\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ ]
\end{aligned}$$

Note that this is using little-endianness. We do not need to know what this is however you should be aware that this is how it is stored in CS 136.

4.5 The Sections of Memory

This is one of the most important concepts to understand if you plan to design effective programs. The sections of memory. For the midterm we are responsible for knowing the following sections: **code**, **read-only**, **global data** and **stack**. The fifth section, **heap**, is not our responsibility and hence will not be referenced again.



When your program is ran, the operating system will allocate x amount of bytes for your program. The smallest byte will be allocated to the code section and the largest byte will be allocated to the stack section. The code, read-only and global data sections grow from smaller addresses to bigger addresses and the stack grows from the biggest address to smaller addresses.

```
int const g = 10; // => the memory address of g could be 0x10.

int main( void  ) {
  int const a = 10; // => the memory address of a could be 0x200.
  int const b = 10; // => the memory address of b would be 0x196.
  int const c = 10; // => the memory address of c would be 0x192.

  return 0;
}
```

If you notice, these sections are constantly converging at each other. If the stack were to grow too big or the global data were to grow too big it would result in stack overflow. This is often occurs with infinite recursion or very deep recursion with poor overhead management.

4.6 Overflow

We got into a moderate amount of detail when it came to overflow. But you may ask yourself, what is overflow? Overflow is when the input value is greater than the maximum or minimum value determined by the amount of bytes allocated for a given type. To share an example, an integer is 4 bytes and hence has 2^{32} possible values. Since we want to represent negative numbers as well, an integer really only represents a number in the range of 2^{31} ... $2^{31} - 1$ (note the -1 to account for 0).

Now that we have determined the range that an integer can take on. What if we wanted to assign an integer to its' maximum value + 1? This would result in an overflow and the value would be some ambiguous number. Be careful when it comes to overflow! Choose the proper data types for the appropriate task or stress on the order of operations to minimize overflow. An example of overflow,

```
int const a = 1000000000; // => 1 billion , OK.
int const b = a * 3; // => 3 billion will overflow , PROBLEM.
```

4.7 Characters in C

You should be familiar with the *char* data type. It is used to represent one byte. A character has 2^8 possible values (-128 ... 127). In computers it is important to be able to communicate with another computer. There became a need to represent characters as numbers, hence the development of ASCII. It stands for American Standard Code for Information Interchange. Yes, I did just Wikipedia that however it may be important to know.

Each number from 0 ... 127 represents a character on the ASCII table. For example, 65 is the uppercase character A. Note, only characters 32 ... 126 (inclusive) are printable characters. Important to understand:

- Characters in ASCII are equivalent to numbers in C.
- Characters are wrapped in single quotes, 'A' not double quotes "A" (invalid).

There is no difference between the following,

```
int const a = 65;
char const b = 'A';

( a == b ) // => true
```

An example of character arithmetic,

```
char to_lowercase( char const c ) {
  return ( ( ( c >= 'A' ) && ( c <= 'Z' ) ) ? c - 'A' + 'a' : c );
}
```

This example illustrates the power of treating characters as numbers to improve readability as well as some character arithmetic.

4.8 Control Flow

In C we have this fancy term called **control flow** which is used to model how a program is executed. When your program is ran, the program is given a specific **state** which includes the position at where the program currently is during its' execution. Operating systems invoke the *main* function, it is the start of your program. There can only be **one** main function in your program! Moreover, the location is known as the **program counter** which stores the address of the machine code for the current instruction (it is in a yellow box in the notes).

Recall the control flow statement *return*. This is used to **jump** back to where the function was called. It controls the flow of your program.

4.9 The Call Stack

In CS 136 we are expected to know what the **call stack** is. Suppose there is a function *main* that calls a function *f*, then that function calls a function *g*. We say that *f* is **pushed** onto the stack. Once it reaches a *return* control flow statement, *f* is **popped** from the stack. This history is known as the call stack.

A **stack frame** consists of the arguments passed to the function, local variables/constants and the return address for the function. When a global constant is declared it is put into the read-only section of memory. Local constants are put into the stack section of memory. When the stack is popped, the local constants disappear however the global constants remain persistent throughout the execution of the program. As of right now, all arguments are copies of their value in memory – important to be aware of this.

Here is a sample set of functions,

```
int g( int const x ) {
  return x + 3;
}

int f( int const x ) {
  int const a = ( x * 2 ) + g( x );
}

int main( void  ) {
  const int result = f( 5 );

  // ...

  return 0;
}
```

The call stack would look as follows before a stack frame is popped,

```
/**
 * ======================
 * g:
 *   x: 5
 *   return addr: f:6
 * ======================
 * f:
 *   x: 5
 *   a: ?
 *   return addr: main:10
 * ======================
 * main:
 *   result: ?
 *   return addr: OS
 * ======================
 */
```

The call stack is stored in the stack section of our memory model. This is important! If you recall what I stated earlier about stack overflow and how it occurs. You should now see why. Here is an overview of all of the memory sections we are responsible for,

```
int const g = 9; // => &g is global data.

int main( void ) { // => &main is code.
  int const s = 9; // => &s is stack.

  return 0;
}
```

4.10 Pointers

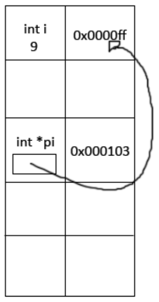
Pointers are a very powerful tool in a C programmers belt. This is arguably one of the toughest concepts to grasp however it is definitely one of the most rewarding. I will spend some time reviewing what pointers are and illustrating some examples. In my opinion, it is essential for you to understand what pointers are and how to use them. I am nearly certain there will be questions on the midterm and final involving pointers.

We have taken a look at the address of operator (&) which will give us the memory address of an identifier. But how do we point to a specific memory address? Well, we use pointers! Declaring a pointer allows us to access specific memory addresses which can be extremely useful and extremely dangerous.

```
// Declaration of a constant integer, i.
int const i = 9;

// Declaration of a pointer to a constant integer, pi.
int const *pi = &i;
```

Observe the declaration of the pointer. We use the * operator to mean "points to". You would read the declaration as, "*pi* is a pointer to a constant integer *i*.". Yes, * is also the multiplication operator however in C it has multiple uses. Note, the value of the pointer is a hexadecimal address.



Seeing as the value of a pointer is a hexadecimal address, can you guess what the size of a pointer is? No matter what data type, a pointer will always have a size of 4 bytes. This is important as we will take advantage of this fact to reduce stack overhead.

To get the value of which *pi* points to we need to use a special operator called the **indirection operator** (*), or more commonly known as **derefencing** *pi*. I know this is confusing but bare with me! The star (*) is used to multiply, point and dereference.

```
int main( void ) {
  // Declaration of a constant integer, i.
  int const i = 9;

  // Declaration of a pointer to a constant integer, pi.
  int const *pi = &i;
  printf( "i = %d\n", i ); // => 9
  printf( "i = 0x%08x\n", &i ); // => 0x00000103
  printf( "*pi = %d\n", *pi ); // => 9
  printf( "pi = 0x%08x\n", pi ); // => 0x00000103
  printf( "&pi = 0x%08x", &pi ); // => 0x000000FF

  return 0;
}
```

Note that the following is a complicated but completely valid expression in C where x is a pointer to some integer,

```
*x * *x
```

What this example is saying is that, dereference x and square it. This can get even more complicated as we are able to declare pointers to pointers.

```
int main( void ) {
  // Declaration of a constant integer, i.
  int const i = 9;

  // Declaration of a pointer to a constant integer, pi.
  int const *pi = &i;

  // Declaration of a pointer to a pointer to a constant integer, ppi.
  int const **ppi = &pi;

  printf( "i = %d\n", i ); // => 9
  printf( "&i = 0x%08x\n", &i ); // => 0x00000107
  printf( "*pi = %d\n", *pi ); // => 9
  printf( "pi = 0x%08x\n", pi ); // => 0x00000107
  printf( "&pi = 0x%08x", &pi ); // => 0x00000103
  printf( "*ppi = 0x%08x\n", *ppi ); // => 0x00000107
  printf( "**ppi = %d\n", **ppi ); // => 9
  printf( "ppi = 0x%08x\n", ppi ); // => 0x00000103
  printf( "&ppi = 0x%08x", &ppi ); // => 0x000000ff

  return 0;
}
```

You should also be aware that we can have pointers as parameters.

```
int some_func( int const *pi, int const **ppi ) {
  // ...
}
```

Keep in mind that to pass a pointer as an argument you must pass the address of the identifier using the address operator (&).

There are two special types of pointers. There is a *null* pointer and a *void* pointer. If a pointer points to *null* then it is considered as invalid or that it points to nothing. It is required that we check for *null* pointers in our pre condition of any given function that is passed a pointer as an argument. On the other hand, if a pointer is a *void* pointer (*void **) then it can accept a memory address where the data type is unknown. You cannot dereference a *void* pointer however you are able to modify the memory it points to which will become useful later.

We were introduced to function pointers which are actually very useful. Function pointers are pointers that point to functions. This gives us the ability to pass functions as arguments to other functions. Consider the following.

```
int sub1( int i ) {
  return ( i - 1 );
}

int main( void ) {
  int ( *sub1_alias )( int ) = sub1;

  int const five = sub1( six );
  int const four = sub1_alias( five );

  return 0;
}
```

Note there is no address of operator when pointer to a function.

Another very useful cool thing with pointers is they can be used to reduce stack overhead. This becomes very apparent when passing structures as arguments. If you call, arguments are copied to the stack frame. If you have a very large structure it will copy it every time you call the function. If you are using recursion, this will create a lot of stack overhead. To avoid this, we pass pointers to structures. First, consider the following.

```
struct struct_1 {
  int x;
  char y;
  char z;
};
```

```
struct struct_2 {
  char y;
  int x;
  char z;
};

printf( "The size of struct_1 is: %d\n", sizeof( struct_1 ) ); // => 8
printf( "The size of struct_2 is: %d\n", sizeof( struct_2 ) ); // => 12
```

This is called structure padding. The processor deals with data types in multiples of 4 bytes. This is to improve efficiency. The bytes not used by the structure will be filled with empty bytes. To pass structures as arguments we do the following.

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

int some_func( struct posn *p ) {
  return ( *p ).x + ( *p ).y;
}

// ... or alternatively using the right arrow selection operation.

int some_func( struct posn *p ) {
  return p->x + p->y;
}
```

5 Imperative C: Module 05

Once module 05 was released there was the "Computer Science equivalent" of a standing ovation in Dave's class. If you recall from the beginning of module 01 I mentioned that we will be revisiting the imperative paradigm soon. Well, this is that moment in time.

Using an **imperative paradigm** is the process of using control flow to manipulate state, using a series of statement to describe how to achieve an outcome[2].

5.1 Imperative Paradigm in Racket

There are three fancy functions that we are introduced to in Racket that allow us to use control flow to manipulate state. These functions are *begin*, *printf* and *set!*. See their respective documentation for further information on those individual functions. Here is an example using all three in a favorite color module.

```
#lang racket ; favorite-color.rkt

(define favorite-color 'blue)

(define (guess-color color)
  (cond
    [(symbol=? color favorite-color) (begin (printf "You guessed correctly!")
                                            (set! favorite-color 'green)
                                            #t)]
    [else (printf "You guessed incorrectly.") #f]))
```

Note the use of the **implicit *begin*** in the else block. *Begin* is similar to *local* in the aspect that they can both be implicitly written in Racket.

Observe that *printf* is outputting to the console and producing *void*. We say that *printf* has a **side effect**, it being that *printf* is outputting to the console. It is extremely important to document side effects in the design recipe. Keep in mind side effects are only in an imperative paradigm - you cannot have a side effect in a functional paradigm.

Note, there can exist functions that only have side effects! These functions are declared with *void*. Example,

```
void hello_world( void ) {
  printf( "Hello world!\n" );
```

[2]There is a fantastic example that Dave illustrates using Kraft Dinner. I hope all of you are lucky enough to get a chance to see his example!

```
  return; // Optional in VOID functions.
}
```

Here is an example of the documentation for a function that will square a number, x and output to the console "You squared me!":

```
; square_x: Int -> Int
; Purpose: Consumes an integer, x and produces the value of
;            x squared.
; PRE: true
; POST: - produces an integer >= 0,
;       - Side Effect: outputs "You squared me!".
(define (square_x x)
  (printf "You squared me!")
  (* x x))
```

5.2 Imperative Paradigm in C

Similar to Racket, you can program imperatively in C. Note there is a key difference between *printf* in Racket and in C. In Racket it produces *void* however in C it returns the number of characters outputted to the console. The return type of *printf* in C is *int*.

In C we have curly braces ({ }). These braces are also known as a compound statement, or more commonly, a C block. A C block behaves similarly to *begin* in Racket with one key difference. In Racket it will evaluate every statement within the compound statement, however in C it requires a control flow statement to tell the block to stop evaluating. This control flow statement is called *return* (which you should be familiar with).

5.3 Control Flow Statements

In an imperative paradigm we are introduced to a few new control flow statements. Those statements being: *if*, *for*, *while*, *switch*, *continue* and *break*. We are responsible for knowing the behavior of each of these control flow statements with the exception of *switch*.

The *if* statements is setup as follows.

```
if (exp) {
  // ... true.
} else if (exp) {
  // ... true.
} else {
  // ... false.
}

// ... or alternatively

if (exp)
  // true.
else
  // false.
```

I am showing the second method of setting up an *if* statement however it is extremely bad practice and should be avoided at all costs! Consider the following.

```
if (exp)
  // ... true.
  if (exp)
    // ... true.
else
  // ... false.
```

But is the *else* applied to the first or the second if statement? In general and with respect to CS 136 - we should avoid this. Keep in mind you can have a long chain of *if*/*else if*/*else* statements. Here is yet another example of a function *sum* that will produce the sum of 0 ... n.

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

// ... or alternatively

int sum( int n ) {
  if ( 0 == n ) {
    return n;
  }

  return n + sum( n - 1 );
}
```

Note that you can have multiple control flow statements. These are used to ”control the flow” of your function. There is an extremely common error this semester on Piazza. This error is ”reached end of non-void function”. This error is caused because you are not including the proper control flow statements. Consider the following.

```c
int do_something( int n ) { // BAD BAD BAD!!!
  if ( 0 <= n ) {
    return n;
  }

  // ... some more code, but oops - no return?!
}

int do_something( int n ) { // OK!
  if ( 0 <= n ) {
    return n;
  }

  return n * -1;
}
```

What would happen if we called the function *do_something* and passed it -1? The function would spit out the error "reached end of non-void function". To avoid this, make the appropriate corrections.

5.4 Mutation

So far all we have been able to use is constants. Recall, constants are immutable, meaning once they are declared they can never be changed throughout the execution of the program. We are finally, emphasis on finally, introduced to variables. Unlike constants, once a variable is **initialized** to a value it can be re-assigned (overwritten in memory) to a new value. Let's consider the following.

```c
int a = 9; // Note the absence of 'const'.

printf( "a = %d\n", a ); // => 9

a = 8; // Mutating a.

printf( "a = %d\n", a); // => 8
```

Remark that we can mutate a as many times as we want throughout our program.

5.5 Assignment Operator

In C the **assignment operator =** (not ==, the equivalent operator) is used to assign a value to an identifier. If you are a Math student you will likely find the following confusing (however if you are reviewing for the midterm I really hope you do not!).

```
x = y;

y = x;
```

The first line is assignment the value of y to x. The third line is assignment the value of x to y. Note these are not equivalent operations! In variable declarations we say that the variable *var* is **initialized** to a value.

The assignment operator performs two actions. In the example above (line 1), x is being assigned to y. After it overwrites x in memory, it will return the value of y. Here is an example of very confusing code.

```
int x = 10;
int y = 11;

printf( "%d\n", ( x = ( y + 1 ) ) ); // => Outputs 22.
```

Remark that the assignment operator is performing a side effect! It is overwriting a value in memory with a new value. These mutations should be documented appropriately.

5.6 Static Storage

Right now you should be aware that *static* gives global constants/variables modular scope. But what happens if we declare a *static* local variable? Static storage stores local variables in the global data section of memory while keeping its scope local to the function that the static local variable was declared in. This may be confusing, however take a look at the following example.

```
int func( void ) {
  static int total_func_called = 0;

  total_func_called = total_func_called + 1;

  return total_func_called;
}

func(); // => 1
func(); // => 2
func(); // => 3
func(); // => 4
```

This function will *return* the amount of times the function has been called. Static storage is very useful as you have probably noticed on assignment five.

5.7 Uninitialized Data

In C you are able to declare variables without an initial value. This is called an uninitialized variable and looks as follows.

```
int i;
```

Uninitialized variables are considered bad practice. However, if you insist on doing it then there are some rules. If you declare a global variable then it will be automatically initialized to 0 (if no value is specified), however if you declare a variable on the stack it will be initialized to some garbage value (some arbitrary value from a previous stack frame). You must **always** initialize variables on the stack.

5.8 Assignment Shortcuts

Ha! I bet this section caught someones eye. As programmers we are extremely lazy. Conveniently, C provides us with ways to assign values to variables. You should know how to use the following however I am listing them here just to recap!

```
int x = 10;
int y = 11;

x = x + 1;
y = y + 1;

// ... is equivalent to.

x += 1;
y += 1;

// ... is equivalent to.

x++; // returns the value of x then increments x by 1.
y++; // returns the value of y then increments y by 1.

// ... or.
```

```
++x; // increments the value of x by 1 then returns the value of x.
++y; // increments the value of y by 1 then returns the value of y.
```

The thing to notice here is the prefix and suffix notation for the ++. This is an operator that will increment the existing value in memory by 1.

5.9 Examples of Mutation

Now that we have learned what imperative programming is, it is time to show you some classic examples of imperative programming. Consider the following function that will swap two values.

```
void swap( int * const x, int * const y ) {
  int const temp = *x;

  *x = *y;
  *y = temp;
}
```

Note the use of pointers and a temporary constant *temp*. Without pointers the swapped values would disappear once the stack frame is popped from the call stack. Let's take a look at another example using structures.

```
void reflect( struct posn * const p ) {
  int const temp = p->x;

  p->x = p->y;
  p->y = temp;
}
```

This function reflects a point in the line $y = x$.

If you noticed in the previous two examples I am using the keyword *const* in a weird place. Here is how it would be read in example 2, "p is a constant pointer pointing to a struct posn". Here are all the possible combinations (important):

```
int *p; // P is a pointer to an integer.

int const *p; // P is a pointer to a constant integer.

int * const p; // P is a constant pointer to an integer.
int const * const p; // P is a constant pointer to a constant integer.
```

The only difference between a constant pointer and a pointer is the value that p points to cannot change. This is exactly what you would expect *const* to do.

5.10 Looping: Iteration

Iteration is the imperative way of doing recursion. There are cases where recursion may seem more "logical" to look at, however iteration is the way CS 136 wants us doing things nowadays. Consider the following introductions of the *while* loop, *do-while* loop and the *for* loop.

```
do {
  // ... some code to be executed indefinitely once.
} while ( exp );

while ( exp ) {
  // ... do something involving the expression.
}

for ( declaration; expression; mutation on the declaration ) {
  // ... do something.
}
```

These two control flow statements will iterate until the expression is false. Note, you can nest loops. Consider the following function that will output a chess board of size n in ASCII.

```
void make_chessboard( int const n ) {
  for ( int r = 0; r < n; r++ ) {
    for ( int c = 0; c < n;  c++ ) {
      const int is_black = ( ( ( c - r ) % 2 ) == 0 );

      if ( is_black ) {
        printf( "**" );
      } else {
        printf( "  " );
      }
    }

    printf( "\n" );
  }
}
```

6 Praise

I tried to keep this section a secret. However, if you have made it this far in my review notes then you are awesome! I wish you best of luck on your midterm. If you have any feedback please do not hesitate to contact me.

Live long and prosper *separates fingers*!

7 Exercises

These exercises are based on material that I feel we as a class struggled with. The professors conveniently posted the assignment averages and question averages of all the students. I picked concepts with the lowest average to create exercises for. **These exercises should not replace incomplete assignment questions - do those first**. Have fun completing these!

There are no test cases or solutions to any of these problems - it is your job to come up with your own test cases. If you need help with any of these questions refer to course material (or Piazza?).

7.1 In Racket

The following exercises are to be completed in Racket. Pay close attention to the restrictions on the questions!

7.1.1 Structures

1. Write a function *reshuffle_bst* that consumes a valid BST. This function should find the smallest node and make it the root node. It should handle all appropriate shuffling to keep the BST ordered. It will produce the new, shuffled BST. Keep in mind a BST is defined as follows. **You must implement this using recursion**.

```
(struct bst-node (key val left right) #:transparent)
```

2. Write a function *bst-height* that will consume a valid BST and produce the height of the BST. You may assume an empty BST has a height of 0.

3. Write a function *bst-full* that will consume a valid BST and produce #t if the BST is full, meaning every node in the tree has two children that are either both non-empty or both empty, otherwise #f.

4. Write a function *posn-equal?* that will consume two valid *posn*s and produce #t if they are they same points on the euclidean plane, otherwise #f.

7.1.2 Lists

1. Write a function *convert-to-assoc-lst* that will consume a non-empty list. It will produce an associative list starting from index 0 treating every element of the consumed list as a value. **You cannot use recursion**. Example:

```
(equal? (convert-to-assoc-lst (list 1 2)) '((0 1) (1 2))) ; => #t
```

2. Write a function *to-base-x* that will consume two integers, n and b where n is a number and b is a base. It will produce all the digits in proper order of n converted to base b. Example:

```
(equal? (to-base-x 9 2) '(1 0 1)) ; => #t
```

3. Write a function that consumes a non-empty list and produces the same list reversed. **You cannot use abstract list functions, you must use recursion**.

7.1.3 Other

1. For this question you are expected to use mutation (*set!*). You will have a lot of freedom on the implementation for this question so be creative. First, write a function *give-backpack* that consumes nothing. It has a side effect that if you do not already have a backpack it will create one for you and output "Congratulations, a free backpack!", otherwise it will output to the console "You fool, you cannot wear two backpacks!" and remove your current backpack.

The second function you need to make is called *give-me-something*. It consumes nothing and produces nothing. If you have a backpack, it will add a random object to your backpack (a number) and output what it added followed by a new line, otherwise it will output "You do not have a backpack!".

The contracts are as follows.

```
; give-backpack: Void -> Void

; give-me-something: Void -> Void
```

7.2 In C

The following exercises are to be completed in C (C99). Pay close attention to the restrictions on the questions!

7.2.1 Structures

1. Write the declaration for a function *minimal_posn_overhead*. It is passed one argument, a struct *posn*. Write it in such a way that stack overhead is reduced as much as possible.

2. Write a function *is_posn_equal* that is passed two arguments that are *struct posns*. This function will return true if they are the same point on the euclidean plane, otherwise false.

3. Write a function *is_orthogonal_set* that is passed two arguments. The first and second arguments are both *struct Vectors*. It will return true if both vectors form an orthogonal set in R^2, otherwise false. Where a *struct Vector* is defined as.

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

4. Write a function *key_exists* that is passed two arguments, an integer, *i* and a valid BST, *abst*. This function will return true if the key exists, otherwise false. Keep in mind that an empty bst points to NULL and the BST structure is defined as follows. **You must implement this using recursion**.

```
struct bst_node {
  int key;
  int val;
  struct bst *left;
  struct bst *right;
};
```

The declaration of the function is as follows.

```
bool key_exists( int const i , struct bst * const pbst );
```

7.2.2 Loops

1. Write a function *make_chessboard* that is passed two arguments, x1 and x2 where x1 and x2 are both integers. It has a side effect that will output a chessboard of size x1 by x2 (width by length). Black squares are "**" and white squares are " " (note the double space). It will return void. **You must implement this using loops**.

7.2.3 Memory Model

1. Figure out where each constant, variable and function declaration is stored in memory. For every constant and variable declared locally to main, determine there exact hexadecimal memory address assuming the address of s1 = 0x300.

```
int const g1 = 9;
int g2 = 1337;
static g3 = 7331;

int main( void ) {
  int const s1 = 100;

  int *ps = &s1;

  char c1 = 'J';
  char c2 = 'P';

  char *pc = &c1;

  return 0;
}
```

2. Draw the memory block (assuming little-endianness) of the following 32-bit integer declaration. Hint, be careful this is a signed integer!

```
int const g = -255;
```

7.2.4 Pointers

1. Write a function *max_val* that is passed three arguments. The first two arguments are integers i, n and the third argument is a function, f. Max val will return max(f(i), f(n)).

2. Write functions *pinc*, *pdec* and *sinc*, *sdec* that will perform the exact same as ++var, –var and var++, var–. Hint, use pointers.

7.2.5 Bitwise

1. Write a function *flip_bits* that is passed one argument. This argument is a character that must be >= 0. This function will flip all 8 bits in the character and return an integer with the new binary representation of the character. **You cannot use the built in C function XOR, you must use iteration and other bitwise operators (bit shifters, & and | are permitted)**. Hint, your mask *could* be 0x80.

```
assert( flip_bits( 9 ) == 11110110 ); // Note 9 is 1001 in binary.
```

7.2.6 Errors

1. What does ”reached end of non-void function” mean?

2. What does ”bus error” mean?

3. Consider the following function. Why is the following solution wrong?

```
/**
 * sum( n ): Consumes one argument, n and returns the sum
 *           from 0 ... n.
 * PRE: n >= 0
 * POST: returns an integer >= 0
 */
int sum( int n ) {
    return ( ( 0 == n ) ? n : sum( n - 1 ) );
}
```

4. Consider the following function. Identify why the solution did not receive full marks?

```
#include <stdio.h>

/**
 * sum( void ): Consumes no arguments. Returns the amount of times
 *              this function has been called.
 * PRE: true
 * POST: - returns an integer >= 0
 */
int func_counter( void ) {
    static int counter = 0;

    counter += 1;

    printf( "This function has been called %d times.\n", counter );

    return counter;
}
```

4. Consider the following function. Identify the error.

```
#include <stdio.h>

/**
 * sum( n ): Consumes one argument, n and returns the sum
 *            from 0 ... n.
 * PRE: n >= 0
 * POST: returns an integer >= 0
 */
int sum( int n ) {
  return ( is_zero( n ) ? n : ( n + sum( n - 1 ) ) );
}

static int is_zero( int n ) {
  return ( 0 == n );
}
```

5. Consider the following function. Identify the error.

```
#include <stdio.h>

int main( void ) {
  int i = 9;
  int k = 10;
  int * const p = &i;

  printf( "i = %d\n", *p );

  pi = &j;

  printf( "j = %d\n", *p );

  return 0;
}
```

7.2.7 Word Problems

1. What is the difference between a function declaration and a definition?

2. What is static storage?

3. What is the difference between the read-only and global data section of memory?

4. What is the difference between a standard library and a library you create?

5. Convert 0xff from hexadecimal to base 2 (binary) and base 10 (decimal). What observations can you make of the binary representation.

6. What is the size of a memory address on a 32-bit machine? How about a 64-bit machine?

If I were to declare an integer on a 16-bit machine, what would its size be?

7. Why is C++ a pun?

8. List four of the five sections of memory.

9. Explain in detail the steps that occur behind the scenes when you hit ”Run” in gedit using RunC.

10. What happens if I do not initialize a variable on the stack?