Student Review Notes Ed 2.0

Preamble

In the second edition (2.0) of my review notes you will find a further in depth perspective (of my original perspective) on the material presented during the University of Waterloo's summer semester offering of CS 136. These notes are not aimed to replace lectures, tutorials, labs, office hours with your professor or TA, Piazza, or any other source directly provided by the course. It is solely aimed as a study aid that you can refer to for peer perspective on the course material.

These notes were written during the University of Waterloo's summer semester offering of CS 136 in 2013. It covers all the material presented in the course. The material that influenced the contents these notes are as follows:

- Lecture slides designed by the professors.
- "C Programming a Modern Approach – Second Edition", textbook for the course.
- Class averages on assignments (publicly posted on Piazza).
- Other posts on the CS 136 Piazza class.
- Student perspectives.

To preserve the chronological order that the material was presented during lecture, these review notes will follow a similar structure. However, some material in these review notes may be introduced earlier or later. This will only occur if we feel as if material flows better in a separate section than it was originally introduced in.

Practice exercises can be found scattered throughout each section of the review notes. There is a special section at the end dedicated to sample problems based on emphasized course material, struggling assignment questions and other factors.

Prerequisites

Student: Can you give me some pointers for the CS 136 midterm?
Professor: 0xff98120f, 0x000000ff, 0x89ff7192, ...

It is expected that before you begin reading these notes that you have attended every lecture, completed all the assignments and that you have already been introduced to a majority the material presented in these notes.

Disclaimer

We ask that you (our peer) respect all of the guidelines we (your peers) set forth with respect to these review notes. If you are unable to respect any of the guidelines mentioned in the proceeding few paragraphs, please delete this file or close the hard copy of the review notes.

1. We alone reserve the right to alter and distribute these review notes. If you see a mistake or feel as if important content is missing, contact us (information provided below) so that we can make the appropriate correction(s).
2. We will be supporting the accuracy of these review notes for the remainder of the summer semester (2013). After such a time these review notes should be considered obsolete.
3. These review notes are provided absolutely free of charge. If you paid for a hard copy or e-copy of these review notes then you are obligated to contact us so that we can take the appropriate action(s) towards the distributor.

Keep in mind that these notes are in no way guaranteed to be accurate. There may be mistakes, outdated information or tangents that will not be directly related to some of the material presented in the course. These notes have been developed by a few of your peers (yes, students!) during our undergraduate year taking CS 136. No professor, TA or anyone involved in the administration at the University of Waterloo has endorsed these notes.

Special Notices

These review notes would not be possible without the offering of CS 136 at the University of Waterloo and all the hard work the professors, TAs and behind the scenes administration put into this course. This is a truly fantastic course and every student involved in creating these review notes are absolutely loving it. Special mentions to the following students:

- Jacob Pollack, author (**jpollack@uwaterloo.ca**).
- Jacob Willemsma, co-author (**wjwillem@uwaterloo.ca**).

Acknowledgements

Some further acknowledgments to the authors of the CS 136 material and other influencing figures.

- Professors and Teaching Assistants for the summer semester of 2013: Dave Tompkins, Olga Zorin and Patrick Nicholson.
- Design inspired by "Linear Algebra Course Notes" by Dan Wolczuk (with permission).

Table of Contents

Preamble i
Disclaimer ii
Special Notices ii

1 Introduction: Module 01 **1**
1.1 Full Racket 3
1.2 Binary Search Trees 9
1.3 Documentation (Design Recipe) 11

2 Modularization: Module 02 **12**
2.1 The Interface 14
2.2 Using Racket Modules 16

3 C Compilers (and RunC) **17**

4 Functional C: Module 03 **19**
4.1 Typing 20
4.2 Function Definitions 21
4.3 Operators 22
4.4 Scope 24
4.5 Recursion 26
4.6 Function Definitions and Declarations 27
4.7 Interface Files and Standard Modules 28
4.8 Structures 30
4.9 Standard IO Module 31
4.10 Main 32

5 C Memory Model: Module 04 **34**
5.1 Hexadecimal, Decimal, Octal and Binary 34
5.2 More Operators 37
5.3 Memory 38
5.4 Blocks of Memory 40
5.5 Sections of Memory 42
5.6 Overflow 43
5.7 Control Flow 44
5.8 The Call Stack 45
5.9 Pointers 47

6 Imperative C: Module 05 **48**
6.1 The Imperative Paradigm 48
6.2 Control Flow Statements 51
6.3 Mutation 53
6.4 Assignment Operator 54
6.5 Static Storage 55

6.6 Uninitialized Data 56
6.7 Assignment Shortcuts 57
6.8 Examples of Mutation 58
6.9 Looping: Iteration 60

7 Efficiency: Module 06 **62**
7.1 Efficiency 64
7.2 Input Size 65
7.3 Ordered Runtime: Big O 66

1 Introduction: Module 01

The official catalog entry for CS 136 at the University of Waterloo is as follows:

DEFINITION
CS 136

This course builds on the techniques and patterns learned in CS 135 while making the transition to use of an imperative language. It introduces the design and analysis of algorithms, the management of information, and the programming mechanisms and methodologies required in implementations. Topics discussed include iterative and recursive sorting algorithms; lists, stacks, queues, trees, and their application; abstract data types and their implementations.

In the prerequisite course(s) – CS 135 (and CS 115/116), the focus was programming in a functional paradigm in Racket (and in CS 116, Python). However in CS 136, as I hope you are now aware, the focus has shifted to programming in an imperative paradigm in C (C99). Let's look into what it means to program in both paradigms.

DEFINITION
Functional Paradigm

The process of evaluating code as a sequence of Mathematical functions, avoiding mutation and the manipulation of state. It has an emphasis on function definition, function application and recursion.

DEFINITION
Imperative Paradigm

The process of evaluating statements that change a programs state. This is done through mutation and control flow.

There is a third paradigm, declarative programming, however we are not introduced this in the course and hence it will not be referenced again throughout these review notes.

From looking at these paradigms it is fair to conclude that one is the opposite of the other. But do not take my word for it, let's see an example.

EXAMPLE 1

Consider the following two functions that return the sum from $0...n$ of a positive integer integer n.

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

Remark that this is the functional approach to handling this problem. Now consider the following.

```
int sum_iterative( int const n ) {
  int acc_sum = 0;

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

  return acc_sum;
}
```

Remark that this is the imperative approach to handling this problem. It is important we understand both paradigms as it is likely that we will be asked to program in a specific paradigm. These review notes will touch base on this in module 05, for now consider only the functional approach to problems.

1.1 Full Racket

It is true! We are finally old enough (as Dave says) to use full Racket. However there are a few things we will need to keep in mind in order to use it properly. Recall that to enable full Racket you must have the following definition at the top of every Racket file.

```
#lang racket
```

Functions

Keep in mind there is some funky terminology for Racket. We **apply** functions which **consume** arguments and **produce** a value. Consider the following example.

EXAMPLE 1 The following function, f consumes two integers, x and y and produces the sum of x and y.

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

Constants

Recall how to define a constant in Racket.

```
(define age 19)
```

When you are defining a constant in Racket it is an immutable identifier. We will see later on how to mutate an immutable identifier in Racket, however not now. Consider the following example.

EXAMPLE 2 The following function, *get_pennies* consumes one integer, n and produces the value of n pennies.

```
(define penny 1)

(define (get_pennies n)
  (* n penny))
```

Constants are quite useful to improve **readability** and ease of **maintainability**. What if the value of a penny changed from 1 cent to 2 cents and we have over 100 different functions using the value of a penny. All we would need to do is change one constant as oppose to changing the value of a penny in over 100 separate places.

Functions without Parameters

This is indeed possible. However, you will find this feature not very useful until we introduce **side effects**. For now, just be aware of what they are. Consider the following example.

EXAMPLE 3 The following function, *get_lucky_number* consumes nothing. It produces a lucky number.

```
(define (get_lucky_number)
  9)
```

But how do we define a number as lucky? This is up to interpretation however a lucky number in my opinion is 9 (trick question).

Top-Level Expressions

These are rather interesting. If you recall, Racket files will be executed line by line. What would happen if you were to place an expression above a function? Consider the following example.

EXAMPLE 4 First consider what the following Racket file will do. Then open Dr Racket and discover for yourself what it really does.

```
#lang racket

; A top-level expression.
(+ 3 4) ; => 7

; A function definition.
(define (f x y)
  (+ x y))

; Some more top-level expressions.
(f 3 4) ; => 7
(f (f 3 4) (f 3 4)) ; => 14
(+ 2 8) ; => 10
```

You should notice that it outputs 7, 7, 14 and 10 to the console. These are are **top-level expressions** that get executed and output to the console.

Logic in Racket

Recall booleans, symbols, strings, characters, logic operators (and ... or) and conditional statements from CS 135 (or CS 115). These are still very important for the Racket portion of CS 136. If you remember the syntax for a conditional statement, we are now introduced to a newer, more compact syntax. Consider the following.

EXAMPLE 5 The following function will consume a symbol and output $\#t$ or $\#f$ if the symbol is my favorite color.

```
(define favorite-color #\b) ; Char b for blue.
(define favorite-color "blue") ; Blue as a string.
(define favorite-color 'blue) ; Blue as a symbol.

; Older syntax.
(define (guess-color color)
  (cond
    [(symbol=? color favorite-color) #t]
    [else #f]))

; Newer syntax.
(define (guess-color color)
  (if (symbol=? color favorite-color) #t #f))

; .. or solely for this specific function.
(define (guess-color color)
  (symbol=? color favorite-color))
```

Keep in mind that you would think 0 is considered as $\#f$ in Racket, however you are wrong. Anything that is not explicitly $\#f$ is considered true in Racket, including 0.

```
(equal? #f 0) ; => False
(equal? #f 1) ; => False
(equal? #t 0) ; => True
(equal? #t 1) ; => True
```

Structures

No, we are not getting rid of these bad boys anytime soon. In full Racket the syntax for structures become more compact with one little tedious, but important detail. Consider

the following.

EXAMPLE 6 Carefully examine the definition of how the *posn* structure is defined in full Racket. Do you spot anything new?

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

If you noticed, it is no longer $define - struct$, it is now just $struct$. In addition, we must add the $\# : transparent$ keyword at the end of the structure.

EXERCISE Create a structure in Racket without the $\# : transparent$ keyword then define a random *posn* and observe what happens when you execute the Racket file.

Lists

From CS 135 (or CS 115) we should be comfortable using $cons$, $list$, $empty$, $first$, $rest$, $list - ref$, $length$, $append$, $reverse$, $last$ and $drop - right$. Consider the following.

EXAMPLE 7 Examine the use of *cons* and *list*. The function *is_equal* will compare two lists and you can assume it works fine.

```
(define lst1 (cons 1 (cons 2 (cons 3 empty))))
(define lst2 (list 1 2 3))

(define (is_equal lstx lsty)
  (cond
    [(and (empty? lstx) (empty? lsty)) #t]
    [(or (empty? lstx) (empty? lsty)) #f]
    [(equal? (first lstx) (first lsty)) (and (is_equal (rest lstx)
                                                       (rest lsty)))]
    [else #f]))

(equal? #t (is_equal lst1 lst2)) ; => True
lst1 ; => '(1 2 3)
lst2 ; => '(1 2 3)
```

If you recall even further, *member* will check whether an element is a memory of a list. In full Racket *member* now produces the tail of the list if true, otherwise $\# f$. Keep in mind only $\# f$ is false in Racket, meaning the tail of a list is still considered true.

Abstract List Functions

Recall what an **abstract list** function is? They are built-in functions to Racket that perform useful operations on lists. The important feature for abstract list functions is that we can pass parameters to them.

We should be familiar with all the following abstract list functions: $filter$, $build-list$, map, $foldl$ and $foldr$. Remember that for the $build-list$ function, we start building the list at 0. Consider the following examples.

EXAMPLE 8 Examine the use of $filter$, $build-list$, map, $foldl$ and $foldr$ in the following top level expressions.

```
(define lst1 (build-list 5 values)) ; => (list 0 1 2 3 4)
(define lst2 (list -2 -1 0 1 2))

(filter even? lst) ; => (list 0 2 4) remember 0 is even.
(map abs lst2) ; => (list 2 1 0 1 2)
(map + lst1 lst2) ; => (list -2 0 2 4 6)
(foldl + 0 lst1) ; => 15
(foldr + 0 lst1) ; => 15
```

Lambda

The use of *lambda* in Racket is probably one of my favorite language features. It is so powerful, if you do not know what it is your mind will be blown! Consider the problem that, we want to add a small bit of functionality that will only be used once and is exclusive to a specific task. We could create a helper function, however it will get messy. What about an anonymous function? Consider the following example.

EXAMPLE 9 Observe the use of *lambda* as an anonymous function to add functionality that would otherwise require the use of a helper function.

```
(define lst1 (build-list 11 values)) ; => (list 0 ... 10 )

(filter (lambda (x) (and (even? x) (> x 5))) lst1) ; => (list 6 8 10)
```

The power of *lambda* in Racket is endless.

EXERCISE Use the abstract list function $build-list$ to create a list of squares from 0 ... 10. Hint, use *lambda*.

Implicit Locals

We no longer need to explicitly state *local* while we are writing local constants or helper functions. This feature can still however be used but it is not needed nor recommended in full Racket. The *local* special form is now implicit.

1.2 Binary Search Trees

The most useful tree we have seen in CS 135 (and CS 115) which we will continue to use is the **Binary Search Tree** (BST). Here let's make an important definition describing what a valid BST is.

DEFINITION
Binary Search Tree

A node-based binary tree data structure, sometimes referred to as a sorted binary tree, which has the following properties.

- The left subtree of a node contains only keys less than the node's key.
- The right subtree of a node contains only keys greater than the node's key.
- The left and right subtree must also be valid BSTs.
- There cannot be any duplicates.

These properties are also referred to as the **ordering property**.

Remark that an empty BST can be represented by the keyword *empty*, however any sentinel value (such as $\#f$) is accepted. I prefer to stick with *empty* and hence it will be used in any further examples.

In this course, a BST is defined as follows.

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

Questions where we are supposed to create functions that perform some operation on a BST will 99% of the time have some form of recursion. A BST is a recursive data structure and hence recursion is more often than not used to access nodes. Consider the following examples to jog your memory and improve your understanding.

EXAMPLE 1

Write a function *key_exists?* that will consume two arguments, a key and a valid BST and will produce $\#t$ if the key exists, otherwise $\#f$.

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

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

EXAMPLE 2

Write a function *insert_node* that will consume three arguments, a key, a value and a valid BST and will produce the new BST with the key and value added to the BST. If the key already exists, it will overwrite the value of that node.

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

(define (insert_node key val abst)
  (cond
    [(empty? abst) (struct key val empty empty)]
    [(equal? key (bst-node-key abst)) (struct key val
                                              (bst-node-left abst)
                                              (bst-node-right abst))]
    [(> key (bst-node-key abst)) (struct (bst-node-key abst)
                                         (bst-node-val abst)
                                         (bst-node-left abst)
                                         (insert_node key val
                                                      (bst-node-right abst)))]
    [else (struct (bst-node-key abst)
                  (bst-node-val abst)
                  (insert_node key val (bst-node-left abst))
                  (bst-node-right abst))]))
```

EXAMPLE 3

Write a function *sum_vals* that will consume one argument, a valid BST and will produce the sum of all the keys in the BST. You can assume all keys are integers (this is a typical property of a BST).

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

(define (sum_keys abst)
  (cond
    [(empty? abst) 0]
    [else (+ (bst-node-key abst)
             (sum_keys (bst-node-left abst))
             (sum_keys (bst-node-right abst)))]))
```

EXERCISE

Write a function *get_values* that consumes two arguments, a non-empty list of valid keys and a valid BST. It will produce a list of values from smallest key's value to greatest key's value.

1.3 Documentation (Design Recipe)

This is a very tedious but necessary aspect of designing successful software. You may have noticed that not all the questions on the assignments given out by the professors are hand marked and you may be thinking, why bother to include a design recipe? Regardless of hand marking or not, it is required as it does two important things:

- To help us design new functions from scratch.
- To aid in the communication of our function to other developers.

This may not seem useful now, however as we proceed into module 02 it will become more apparent with respect to modularization. Recall from CS 135 (and CS 115) that there were quite a few elements to the design recipe. Let's define what the design recipe is in CS 136.

DEFINITION
Design Recipe

This is the process of communicating information about our function to a client, another developer or our self through smart documentation. It should communicate the following information:

- The contract of the function (ie what it consumes and produces).
- The purpose of the function.
- The pre and post conditions of the function (ie what are the input and output restraints).

The design recipe should be commented out above the function it is describing. Consider the following example.

EXAMPLE 1

Write the design recipe for the Racket equivalent of the *sum* function on page 1 of these review notes.

```
; (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
(define (sum n)
  ...)
```

Well documented code is one of the most important features to have in your projects. Documentation must be written for all functions and helper functions (including locally defined helper functions).

2 Modularization: Module 02

One of the struggles in modern day computing is how do we collectively work a project. Not only that but for a large program with a large collection of functions, how do we break it down into smaller parts. This is where the push for modularization comes in.

DEFINITION **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. Each module should have low coupling and high cohesion.

Without modularization it would be extremely difficult to work as a collective towards a common goal on a project. There are three key pushes for modularizing a project, **re-usability**, **maintainability** and **abstraction**.

DEFINITION **Re-usability**

The process of writing modules that can be taken from one project and applying them to future projects. For example, creating a module that handles MySQL database interaction can be applied to multiple projects that require the ability to access a MySQL database.

DEFINITION **Maintainability**

The process of separating the project into multiple modules with a specific purpose, enabling multiple programmers to work on different aspects of project in parallel. In particular, if one module needs an update then a programming can simply extract that module, update it, then plug it back into the project without impeeding on other development on the project.

DEFINITION **Abstraction**

The process of using a module designed by another company or programmer without actually knowing how it works.

Some of these terms are better understood with an example, consider the following.

EXAMPLE 1

Going into the garage to repair your car radio. Instead of heading to your local Ford dealer and buy a new car, you simply extract the radio from the car and replace it with a new radio.

Solution: Maintainability.

EXAMPLE 2

Putting triple A batteries into your flashlight before you head out camping.

Solution: Abstraction.

EXAMPLE 3

Taking notes in your highschool calculus course and then realizing that Math 137 will spend a lot of time reviewing what you learned in highschool. So instead of taking extensive new notes, you use your old notes from highschool!

Solution: Re-usability, something I did not do :(

EXERCISE

Oops, you spilt milk on your pants. Putting your pants in the washer is an example of...

EXERCISE

Going camping with two of your friends, similar to how Raj, Leonard and Howard did in the Big Bang Theory. If you were to use the telescope you brought to see the meteor shower, this would be an example of...

EXERCISE

Using your iPhone to communicate with friends on iMessage is an example of...

EXERCISE

Putting 4 more GB of RAM into your computer because you forgot you had a 64-bit CPU is an example of...

2.1 The Interface

One of the most important aspect of modularization is developing the interface for a module.

DEFINITION
Interface

Consists of a collection of functions (function definitions) that are accessible outside of the module (public), as well as the appropriate documentation for those given functions. In short, everything a client would need in order to use our module.

But how do we know what a successful modular design should achieve. What standards we should meet in our design. If you recall from the definition of modularization the terms low coupling and high cohesion were used. In successful modular design we aim to achieve both low coupling and high cohesion.

DEFINITION
Low Coupling

There are as few modular inter-dependencies as possible.

EXAMPLE 1

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 any of the functionality. This is an example of high coupling since you cannot extract modules without extracting other modules it depends on.

DEFINITION
High Cohesion

All the functions within a given module are working collectively towards a common goal or purpose.

EXAMPLE 2

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.

Information Hiding and Documentation

It can be important to hide information about your module from the client. Consider you were contracted by TD Canada Trust (a Canadian bank) to implement a module that would handle deposits and withdrawls from an ATM. You will want to hide some of the implementation of your module to avoid letting the client access functions that could alter the intention of your program. This is commonly known as **security**.

In Racket, when we implement a function it is automatically hidden from the client. To allow the client to have access to a given function in our module, we must *provide* the function. Consider the following.

EXAMPLE 3 Implement a module that will let a client guess your favorite color.

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

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

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

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

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

; (fav-color?/helper): Symbol -> Boolean (private)
; Purpose: Consumes a symbol, color and produces #t if the color
;          guessed is my favorite color, otherwise #f.
; PRE: true
; POST: returns a boolean
(define (fav-color?/helper color)
  (symbol=? favorite-color color))

; See interface above (public).
(define (fav-color? color)
  (fav-color?/helper color))
```

Observe that the documentation for the functions that we want the client to use are placed at the top of the file followed by some delimiter. When documenting interfaces in Racket all of the design recipes for functions that we want to *provide* to the client should be at the top. Documentation for other functions, such as helper functions, should be placed respectively with the function definition. You can document multiple functions in the interface by separating their design recipe with a new line.

2.2 Using Racket Modules

To use a Racket module you must use the *require* special form. When it reaches a line with *require*, it will stop executing your code and begin executing the code for the module you are including. Once this is done it will return to your code and continue executing where it left off. Consider the following client module for our favorite color module.

```
#lang Racket ; client.rkt

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

; Guessing my favorite color.
(fav-color? 'green) ; => #f
(fav-color? 'red) ; => #f
(fav-color? 'blue) ; => #t
(fav-color? 'orange) ; => #false

; ERRORs
(fav-color?/helper 'blue) ; => Error!
```

Testing

In full Racket there is no such function called $check-expect$. The way we will create a testing module is by checking if the function with desired inputs is equal to a desired output, similar to $check-expect$.

EXAMPLE 1 Design a testing module for my favorite color module.

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

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

; Performs some testing.
(equal? (fav-color? 'red) #f) ; => Test passes.
(equal? (fav-color? 'blue) #t) ; => Test passes.
```

3 C Compilers (and RunC)

This section is a preface to our introduction to C99 (and henceforth referenced as just C) in CS 136. I feel that it is very important that we understand what goes on behind the scenes when you click "run" in RunC. Understanding what goes on behind the curtains is important when designing successful software. You can skim through this section and still design successful software however I strongly recommend against it.

Lucky for us, RunC is a very useful compiler when learning how to program in C. If you are unfamiliar with C compilers, then to make an appropriate analogy: RunC is very similar to the learning languages in Racket.

DEFINITION
C Compiler

Translates our readable C source code into machine code, a representation of source code that the processor is able to interpret and execute.

One of the most popular C compilers is the GCC compiler. This compiler is available accross many platforms and is extremely robust. GCC is installed on your Ubuntu environment however it must be accessed through the terminal.

In order to "run" a program it must typically undergo three phases. These three phases are known respectively as **preprocessing**, **compiling** and **linking**.

The program is initially given to a preprocessor.

DEFINITION
Preprocessing

Scans all of your source code for commands that start with # (known as **directives**). If a directive is found, it will perform the appropriate action(s).

The modified program is now passed to a compiler.

DEFINITION
Compiling

Turns all of your readable source code into machine instructions, **object code**.

The compiler will output object files for each respective source code files. These object files are then passed to a linker.

DEFINITION
Linking

Grabs all of the object files and any additional libraries needed (such as *stdbool*) and links them into a single executable.

Fortunately with respect to this course, this process is automated in the RunC environment when we click "run". However if we were using a different compiler such as GCC, we would need to perform all three steps manually (typically the preprocessing step is integrated with the compiling step and hence only two steps are typically explicitly referenced).

The bash commands necessary to compile and link your program typically vary, depending on the compiler and operator system you are using. In our Ubuntu environment, the command to compile our code with GCC is as follows:

```
gcc -o -std=c99 file file.c
```

Note the c99 flag. This is required to compile our code using the C99 standard.

A common thing to do when developing support software is to release the object code for your module, or a compiled DLL. The advantage of distributing your module's object file that you are able to provide a client with the necessary tools to use your program without explicitly giving them the source code. These types of files will end with .o or .dll.

EXERCISE

You have been hired to create a module that cracks a safe. Your clients provide you with a dummy safe called safe.o that has functions *unlock* and *withdraw_money*. What should you do with the safe.o while developing your safe cracker?

Integrated Development Environments

It will become a pain to compile all of our source files through the terminal one by one with no ability to debug or easily test our program. An **integrated development environment**, commonly known as an IDE is a program that allows you to work on your source code, compile, link and run your executable with ease. In addition to that, most IDEs have the ability for you to debug your code and view the machine instructions for your program.

The "IDE" we use is called gedit, which is actually only a text editor. RunC is an external tool built into gedit that allows us to compile, link and run our program similar to how an IDE would.

In Summary

For this course we will not be expected to know how to use different C compilers or the phases a compiler takes when compiling our code. However if you understand what goes on behind the curtains it will help you design successful software and debug any problems you may encounter in the future.

Keep in mind that we are expected to compile with RunC. Your source code may compile for other compilers such as GCC, however marmoset will be compiling our source code with RunC.

4 Functional C: Module 03

To ease us into C, we will first start by using it in a Racket-like *functional* style.

In this section, we go through most of the transitions from Racket to C99. Please note that we are in fact using C99, there are quite a few different versions of C! From now on we will not state that we are using C99 as it will remain the same the whole way though.

Another interesting aspect to note is that C, unlike Racket who uses dynamic typing uses *static typing*. Keep in mind the word static comes up for a lot of different reasons.

DEFINITION
Static Typing

Static typing is the process of verifying the type of an input in the source code. C uses this type of typing, visible when declaring anything at all.

DEFINITION
Dynamic Typing

Dynamic typing is the process of verifying the type of an input at runtime (when it compiles). Racket uses this which is why there is no need to signal to DrRacket what types you are using.

EXAMPLE 1

Here's how you would regularly define things in a Dynamic Typing language, in this case Racket.

```
#lang racket

(define x 9)
(define x "Hello")
(define x (list 1 2 3))
```

EXAMPLE 2

Here's how you would regularly define things in a Static Typing language, in this case C.

```
#lang racket

const int x = 9;
const char x[6] = "Hello"; // this is a string, which we will learn later.
const int x[3] = {1 2 3}; // this is an array, which is a little like a list.
```

4.1 Typing

So we've shown you a few types in C that you don't need to know about yet, so why not focus on the ones you need to know, that's what this section is about.

In Racket we had readily available predicate functions such as *integer*? or *cons* that would easily determine the data type of a constant variable (constant and variable should never be in the same sentence).

In C, we don't have or need to have predicates such as those since types are all written in the actual source code, so we have no need to check.

You may have already seen in the last block of code how to declare constants and comment in C, however we'll just quickly go over it again.

In C, any text on a line after // is a comment. Furthermore, any text between /* and */ is also a comment. Use these for writing big block comments, like for your documentation.

In C, you **call** a function, they are **passed arguments** and **return** a value. Furthermore, in C constants are **declared** whereas in Racket they are **defined**. These differences are very important for documentation.

```
// Declaring a constant in C:
int const x = 9;
// Alternatively:
const int x = 9;
```

C99 says that *const* will be applied to the identifier at it's **left**. If there is nothing to its left, then it will apply to the identifier at it's **right**. Though both methods work and it does not matter very much at this point in C, the first method is preferred.

There are a number of different styles to use while naming variables and functions. The two we are suggested to use are either camelcase or the underscore style. Note that you must start an identifier with a letter (a number will not work). As long as you are consistent with your style and it is readable, you should be in the clear.

```
// Variable declared in CamelCase:
int const MaxPaycheckReceived = 245;

// Variable declared in Underscore:
int const Max_Paycheck_Received = 245;
```

4.2 Function Definitions

No newline at end of file Since when we type in C, we use static typing, there are a few more things to keep in mind when declaring a new function. In C you must have a specified return data type and every parameter must have a specified data type. All these specifications are written in the source code. So the following 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)))))
```

... becomes the following 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 n)
{
  return ( 0 == n ) ? 0 : (n + sum(n - 1));
}
```

Note the requirement to write *int* before the function name to specify the output and *int* before the argument n to specify the parameter. *Int* in this case stands for **integer**, which is the only type we'll be using at this point of the course.

4.3 Operators

When we were dealing with Racket +, -, /, * were all types of functions. However in C we call these operators.

Basic Operators

There are a ton of operators in C, all with different varying levels of importance. However, the basic arithmetic operators all follow BEDMAS. If you have any doubt as to the order of operations of your program, do not hesitate to use parentheses to control the flow of your statement. One last different is that C uses $infix$ notation rather than the $prefix$ notation used in Racket. $Infix$ notation is the one you've been using your whole life, so it makes things a little more clear.

EXAMPLE 1 Here are a couple examples of the use of basic 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
```

EXERCISE Create a function in C that takes in an integer and returns the integer cubed.

Logic Operators

Just like in Racket, in C we have various booleans and logic operators. In C, booleans are 0 and 1, where 0 is false and 1 is true. To check equivalence you must use a double equals sign in C.

```
// The value of ...
( 9 == 8 ) // => false, 0
( 9 == 9 ) // => true, 1
( 2 = 9 ) // => error
```

The use of not, and and or in Racket is translated to !, && and || respectively 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, as it can prevent long runtimes.

The ternary operation in C ? works a lot like an if statement. The ternary operation follows a predicate and then returns either the first option if it is true, or moves on the the second option if false. You can also link ternary operations together to work in a way similar to the *cond* statement in Racket.

EXAMPLE 2 Here's how to link ternary operations together to get a *cond* like behaviour.

```
int ternary_operation(int n)
{
  (n == 8) ? 5 : // if n is 8, it will return 5.
  (n == 9) ? 4 : // if n is 9, it will return 4.
  (n == 10) ? 3 : 1; // if n is 10, it will return 3.
  //Else return 1 for all other n.
}
```

Other operators in C include >=, <=, > and <. Once again if you are not sure in the order of precedence these operators take in comparison to one another, use brackets for safety.

4.4 Scope

In C, as with all programming you will ever do. it is very very important to be aware of scope. Scope in C is consistent with Racket with a few new complexities. As of now, we are introduced to three types of scope: **global**, **local** and **block**.

DEFINITION
Global Scope

Global scope is a variable or function that is available to all outsides file sources, and both inside and outside of the file functions.

DEFINITION
Module Scope

Module scope is a variable or function that is only visible within that module (likely just a single file). This is used to secure content that you do not necessarily want uses having access to while using your module. A common way in C to do with is by putting the prefix *static*.

DEFINITION
Block Scope

Block scope is a variable or function that is available only in a select code block (between braces ...). These are for variables and functions that are used during the function call and serve no purpose outside that function.

EXAMPLE 1

Here's a visual representation of all the different types of scope

```
int const g = 9; // Global scope.
static int const f = 9; // Module scope.

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

  {

    int const l = 10; // Block scope.

    return l + f;
  }
}
```

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. That word is *extern*. For example, say we had the following module.

```c
// Module A (.h)

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

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

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

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

To use a we would need to properly call it.

```c
// 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);
}
```

Last thing to know in C is that, you cannot run a top level program with top level operations. If you recall in Racket we had top-level expressions, in C there is no such thing!

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

(a + b); // Error, C cannot evaluate this.
```

EXERCISE Does anything special happen when you define a variable with a static prefix in a block scope? What could this be used for? (Hint, Assignment 5)

4.5 Recursion

By now one has likely become a recursion pro and will be happy to know that recursion works exactly as one would expect in C. We will illustrate this using the implementations from sum from 0 to n using recursion both in Racket and C.

First in Racket...

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

... and in C

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

As one can see, it's pretty straight forward.

4.6 Function Definitions and Declarations

In C, there is a difference between declaring a function and defining a function. When you define a function in C, it contains the body and actual code of your function. However in C you are required to also declare functions before you use them. You can have the definition anywhere in your module, just as you can have the declaration anywhere as well (declarations are most commonly found in interface files).

EXAMPLE 1 Here are a few examples of the do's and don'ts of definitions and declarations

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

// Function definition (includes block scope).
int sum(int 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 n);

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

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

The declaration for *sum_helper* must be placed above where you plan to reference it in your program. Simply writing a declaration for the *sum_helper* function before the *sum* function call will solve that problem.

4.7 Interface Files and Standard Modules

For the first time we can access both files that we have written and those provided by regular C99! This opens up the possibility to do so much more with your code.

Interface Files

In C, we normally separate our written source code in our implementation file (.c) and our interface (.h). The interface file contains all declarations to useful global functions from implementation files, as well as the proper documentation. Creating an interface file makes it very easy to gather all the information we need into a single, easy to read and use file.

EXAMPLE 1 Here's an example of an implementation file and it's complimentary interface file

```c
// 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 n);

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

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

```c
// 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 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.

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.h*, *stdbool.h*, *stdio.h* and *limits.h*. A simple Google search will help you understand what each of them do if you are not sure.

EXERCISE

Write out the interface for a program that requires the ability to output statements (prinf) and takes in three different implementation files of your own.

4.8 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.

EXAMPLE 1 Creating a structure in C.

```c
struct posn {
  int x;
  int y;
}; // Don't forget this semi-colon!!!

struct const posn p = {1, 2};

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

// or alternatively...

struct const 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 use *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.

EXAMPLE 2 Checking if two posn structures are equal.

```c
#include <stdbool.h>

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

EXERCISE Write a structure for a box, with parameters for width, length and height. Using this structure, create a function that will calculate the area of the box and another function that will calculate the volume of the box.

4.9 Standard IO Module

One module in particular is very important and powerful and that is *stdio.h*. The Standard IO module allows one to print statements to the console.

EXAMPLE 1 Here are a couple of examples showing the uses of the stdio.h library.

```
// 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;
}
```

The most important thing to know about printf is the symbol "backslash n" which signifies the creation of a newline. Play around outputting lines with a "backslash n" at the end and without and see what happens.

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.

EXERCISE Write a function that outputs a "nameplate" corresponding to various facts about you on every line. (E.g. Name, UserID, Residence, Program, Favourite food, ...)

4.10 Main

At this point, I hope that you are wondering "well how the heck to I run my code?!?". Well, you are finally big enough to know. The only way in C to run your modules is to have a function called *main*. Every C function must have a main function, else the operating system will not know where to begin running your program! So yeah, it's a little important. Also a very quick and important note is that your function can only ever have one main function. Your code will not compile with more than one main.

DEFINITION
Main

Main is where a program starts it's execution. It is responsible for the high-level organization of the program's functionality and has access to all the command arguments given to the program when it was executed.

EXAMPLE 1

Here is the syntax for the main function. This is very important to understand.

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

  // ... tests.

  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.

EXAMPLE 2

Here's an example testing suite for our sum function written earlier on in this document.

```
// testing module for sum.
```

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

#include "sum.h"

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

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

  printf("done!\n");


  return 0;
}
```

EXERCISE Write your own testing suite for testing the area and volume function from the last section.

EXERCISE Try writing a whole function in your main function, include testing and documentation.

5 C Memory Model: Module 04

This section covers some of my favorite topics in CS. I may get into a lot more detail than we need to know with respect to this course, however every detail is important and will likely be brought up in future CS courses. I will explicitly label what we are responsible for knowing to avoid any confusion.

Computers measure their memory capacity in **bytes** (8 bits). A **bit** is a **bi**nary dig**it** that is either on, 1 or off, 0. These are just units of measurement, similar to milometers, centimeters, kilometers, etc. however they are specific to a purpose with computers.

The need to represent numbers in different bases is important when representing different measurements.

5.1 Hexadecimal, Decimal, Octal and Binary

In Computer Science we will often represent numbers in their base 16, base 10, base 8 and base 2 forms, respectively hexadecimal, decimal, octal and binary. It is important to understand why we do this and what it means to have a number represented in base x. Let's begin with decimal as it should be the most familiar to you.

Decimal

DEFINITION
Decimal

Relating to or denoting a system of numbers based on the number of powers of 10. Base 10 identifiers include 0 ... 9.

Decimal, or alternatively known as base 10, was developed simply because we have 10 fingers accross 2 hands. It is the most popular form of communicating a numerical from one person to another. With these ten identifiers we are able to represent larger numbers such as 1000, 999 and 1337. Consider the following.

EXAMPLE 1

Represent 1204 in its powers of 10.

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

EXERCISE

Represent 1337 in its powers of 10.

Binary

DEFINITION
Binary

Relating to or denoting a system of numbers based on the number of powers of 2. Base 2 identifiers are only 0 and 1.

Binary, or alternatively known as base 2, is used when determining whether something is on (1) or off (0). Bits alone are rarely used in Computer Science as one bit represents such an insignificant quantity. It is common to hear the word byte, which refers to a group of bits.

DEFINITION
Byte

Represents a group of 8 bits. One byte can represent 0 ... 255 distinct values.

EXAMPLE 2

Represent 255 in its powers of 2.

Solution: $255 = (1) \times (2^7) + (1) \times (2^6) + (1) \times (2^5) + (1) \times (2^4) + (1) \times (2^3) + (1) \times (2^2) + (1) \times (2^1) + (1) \times (2^0)$

EXAMPLE 3

Convert 255 from decimal to binary.

Solution: $255 = 11111111$

EXERCISE

Convert 15 from decimal to binary.

Octal

DEFINITION
Octal

Relating to or denoting a system of numbers based on the number of powers of 8. Base 8 identifiers are 0 ... 7.

Octal, or alternatively known as base 8, is occasionally used in Computer Science. Octal became a popular numbering system when IBM introduced 12-bit, 24-bit and 36-bit words. Octal was an ideal abbreviation of binary for these machines because their word size is divisible by three (Wikipedia).

EXAMPLE 4

Represent 36 in its powers of 8.

Solution: $36 = (4) \times (8^1) + (4) \times (8^0)$

EXAMPLE 5

Convert 36 from decimal to octal.

Solution: $36 = 44$

EXERCISE

Convert 3600 from decimal to binary.

Hexadecimal

DEFINITION
Hecadecimal

Relating to or denoting a system of numbers based on the number of powers of 16. Base 10 identifiers are 0 ... 9 and A ... F. Hexadecimal is often denoted by 0x, ie 0xF represents 15 in hexadecimal.

Hexadecimal, or alternatively known as base 16, is used quite often in Computer Science. It is often used when representing memory addresses or the value of 1 byte (one hexadecimal identifier represents 4 bits).

Hexadecimal is extremely important! Let's take a look at a few examples.

EXAMPLE 6

Represent 255 in its powers of 16.

Solution: $255 = (15) \times (16^1) + (15) \times (16^0)$

EXAMPLE 7

Convert 255 from decimal to hexadecimal.

Solution: $255 = 0xFF$

EXERCISE

Convert 255 from decimal to binary. Then convert the binary representation of 255 to hexadecimal. What can you observe?

Observe that in C the following are equivalent.

```
int const x = 0xf; // => 15
int const y = 15; // => 15
```

5.2 More Operators

For the memory model of C we are introducing two new operators. The *sizeof* operator, that will determine the amount of bytes required to store a given datatype (determined at compile-time) and the address of operator, which grabs the address in memory, commonly known as the memory address, of a give identifier.

DEFINITION
Memory Address

Consider all the blocks in memory as an array. The memory address is the index of a given element of that array, where each element has a 4 byte index and a 1 byte value.

Consider the following.

```
int const size_int = sizeof( int ); // => 4
```

... and the address of operator.

```
int const g = 9;
int const g_address = &g; // => address of g, 0x____.
```

EXERCISE What is the size of a *char* and a *bool*?

EXERCISE What is the size of a *int* pointer?

EXERCISE What is the size of a *bool* pointer?

EXERCISE What is the size of a *struct posn* pointer?

EXERCISE Declare 4 constant integers on the stack and output all of their memory addresses to the console. Do you notice a pattern? Do the same but for global constant integers.

5.3 Memory

In every computer there is something called **primary memory** (RAM) and **secondary memory** (hard drive, flash drives, etc.). The one term we will worry about is primary memory.

DEFINITION
Primary Memory

Commonly known as primary storage and main storage, is the only one directly accessed by the CPU. The CPU continuously reads instructions stored there and executes them as required.

Programs are launched into primary memory and disappear when the program is exited or when the computer shuts down. Secondary memory will remain persistent and hence is often used for storage. Primary memory is much faster than secondary memory, however primary memory is much more expensive.

Quantifying Memory

Here is a list of some common units that we should be familiar with:

- Byte = 8 bits, 2^8 possible values.
- KiloByte (KB) = 1024 bytes.
- MegaByte (MB) = 1024 kilobytes.
- GigaByte (GB) = 1024 megabytes.
- Terabyte (TB) = 1024 gigabytes.

CPUs

This is a rather short section barely scratching the surface of CPUs, but it's important to know this for CS 136. You have probably heard the phrases "... my computer has a 32-bit CPU ...", "... yea well my computer has a 64-bit CPU ..." and "... how come I cannot have 8 GB of RAM on a 32-bit CPU ...", but what does it all mean?

The amount of bits your CPU is determines the amount of memory addresses you can have. If you have a 32-bit CPU then you can have up to 2^{32} memory addresses, which is exactly $4,294,967,296$ memory addresses.

If you have a 64-bit CPU then you can have up to 2^{64} memory addresses which is a considerably larger amount than a 32-bit machine. Note that you can have **up to** n amount of memory addresses, not exactly n. For CS 136 we are expected to be using a 32-bit machine, we just wanted to make sure you understand what that really means.

Accessing Memory

The big question is, with all of this memory how do we access it? If you recall, we mentioned the term **memory address**. Now, let us make a formal definition.

DEFINITION **Memory Address**

Is an identifier (4 byte identifier with respect to this course) for one byte of memory.

Remark that a memory address is a number. In fact, it is a number that is often represented in hexadecimal because two hexadecimal digits can easily represent a byte. Depending on your CPU, you can have a lot of memory addresses (and hence a lot of memory) or very few memory addresses (and hence not a lot of memory). The CS 136 Ubuntu environment is a 32-bit machine and hence has 2^{32} *unique* memory addresses.

Consider the following program.

EXAMPLE 1

Recall how we declare a constant in C?

```c
#include <stdio.h>

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.\n", &a ); // => 0x2ff

  return 0;
}
```

Observe that the value of a is 9, however the memory address of a is $0x2ff$. Memory addresses are the computers identifier for where the value of a is stored in memory. It will keep track of the memory address as long as the memory block is in use.

EXERCISE

What do you think the block of memory from example 1 looks like for a?

5.4 Blocks of Memory

This section is tangent from the C memory model with respect to this course, however it is extremely important to be aware of this when you are reading bits from specific memory addresses.

Recall that an integer requires 4 consecutive memory addresses to store its value. Let's presume that we declared a global integer, a, and that the address of operator returns the value $0x2ff$. This means that the bit representation of a is stored in $0x2ff$, $0x300$, $0x301$ and $0x302$.

But which memory address contains what bits? Let us make two important definitions.

DEFINITION
Little Endianness

The ordering of the binary representation of a number from least significant bit to most significant bit.

DEFINITION
Big Endianness

The ordering of the binary representation of a number from most significant bit to least significant bit.

There is a third type of endianness called **mixed or middle endianness**, however it is not as widely used as the other two types of endianness and hence will not be mentioned again.

Big endianness is a lot easier to understand because it is how we represent numbers as human beings. If we return to the example above, the block of memory for a would look as follows assuming big endianness.

$$\begin{aligned} 0x2ff &= [00000000] \\ 0x300 &= [00000000] \\ 0x301 &= [00000000] \\ 0x302 &= [00000101] \end{aligned}$$

Observe that the most significant bits are in the leading memory address while the least significant bits are in the proceeding memory addresses. This is exactly like representing the number one hundred and twelve as 112. On the flip side, let us consider what the blocks of memory would look like assuming little endianness.

$$\begin{aligned} 0x2ff &= [00000101] \\ 0x300 &= [00000000] \\ 0x301 &= [00000000] \\ 0x302 &= [00000000] \end{aligned}$$

Observe that the least significant bits are in the leading memory address while the most significant bits are in the proceeding memory addresses.

There are quite a few advantages and micro performance improvements with each endianness, such as addition and subtraction, however we do not need to be aware of them. When communicating with another machine it is important to be aware of how that machine stores values in memory or you could end up reading a value completely incorrect assuming the wrong endianness.

EXERCISE

If my_num is an integer declared on the stack for a 64-bit machine and initialized to -9 (where its memory address is $0xff$), what does its blocks of memory look like if the CPU uses little endianness?

5.5 Sections of Memory

The sections of memory is one of the most important concepts to understand if you plan to design effective software. We are introduced to four of the five sections of memory: **code**, **read-only**, **global data** and **stack**. The fifth section, **heap**, is not our responsibility for the time being.

When your program is ran, the operating system will allocate a block of memory of size x that your program can use. The smallest memory addresses will be allocated to the code, read-only and global data sections and the largest memory addresses will be allocated to the stack section. Observe that 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.

Consider the following program.

```
int const g1 = 10; // => Memory address of g1 could be 0x10.
int const g2 = 10; // => Memory address of g2 could be 0x14.
char const g3 = 'A'; // => Memory address of g3 could be 0x18.
bool const g4 = true; // => Memory address of g4 could be 0x19.

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

  return 0;
}
```

Observe that the stack is converging with the global data section of memory. If the stack were to grow too big or the global data section were to grow too big then it would result in one section overwriting memory used by another section, commonly known as stack overflow (as global data overflow is much harder to achieve). Cases like these can often be found behind very deep recursion or poor stack overhead management.

5.6 Overflow

What happens if our data type can only hold a 1 byte value but we pass it a 2 byte value? This is known as **overflow** and is a very common mistake that we need to be very aware about! Let us define two important terms.

DEFINITION
Overflow

When the input value is greater than the maximum value determined by the amount of bytes allocated to a given data type.

Similarly,

DEFINITION
Underflow

When the input value is less than the minimum value determined by the amount of bytes allocated to a given data type.

Recall that an integer can take on 2^{32} distinct values and hence a signed integer represents any number, n, where $n \in [-2^{31}, 2^{31} - 1]$. What do you suppose would happen if we were to assign the value of a signed integer to be 2^{31}?

EXAMPLE 1

Consider the following example of integer overflow.

```
int const a = 2147483647; // => 2^31 - 1.
int const b = a + 1; // => 2^31, overflow, an ambiguous number.
```

Depending on your environment the value of b may be **rolled over** and become the smallest possible signed integer, however this behavior is undefined and unpredictable for a signed integer.

EXERCISE

What is the smallest possible positive number that will overflow a *char*?

EXERCISE

Why is overflow for a signed integer considered as undefined behavior?

EXERCISE

Write a function *safe_add* that will safely add two valid integers or return -1 if their sum is an overflow.

5.7 Control Flow

In C we have this fancy term called **control flow**.

DEFINITION
Control Flow

Is used to model how a program is executed.

When your program is ran by the operating system, it 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 starting point of your program.

Remark that you can only have one main function! Moreover, the position (or location) of your program is known as the **program counter** which stores the address of the machine code for the current instruction.

Recall the control flow statement *return*? It is used to **jump** back to where the function was called. It "controls the flow" of your program. There are a lot more control flow statements in C which we will be introduced to in module 05 however for now only worry about *return*.

5.8 The Call Stack

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**.

DEFINITION
Stack Frame

Consists of the arguments passed to a function, local variables/constants declared locally to the function and the return address for the function

Remark that when a global constant is declared it is put into the read-only section of memory and that local constants are put into the stack section of memory. When the stack frame is popped, the local constants disappear however the global constants remain persistent throughout the execution of the program.

Notice that all arguments are copies of their original values. Meaning that if we pass an argument and alter it inside a function it will not alter the original value outside of the function.

EXAMPLE 1

Describe the call stack before any stack frame is popped for the following 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 be illustrated as follows.

```
/**
 * ====================
 * 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 we stated earlier about stack overflow and how it occurs. You should now see why!

Here is a review of all of the memory sections we are responsible for thus far:

```
int const r = 9; // => &r is read-only.
int g = 9; // => &g is global data.

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

  return 0;
}
```

5.9 Pointers

To be continued...

6 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 we mentioned that we will be revisiting the imperative paradigm soon. Well, this is that moment in time.

6.1 The Imperative Paradigm

You can program in an imperative paradigm in both Racket and C. You will find that Racket is designed as more of a functional language whereas C is designed to be both a functional and imperative language.

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 "Correct!")
                                            (set! favorite-color 'green)
                                            #t)]
    [else (printf "Nope, you are wrong!") #f]))
```

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

Furthermore, observe that *printf* is outputting to the console and producing *void*. We say that *printf* has a **side effect**, that side effect being the output to the console.

DEFINITION
Side Effect

A secondary effect, typically not the main intention of the function, occuring behind the scenes.

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.

EXAMPLE 1

What is the side effect of the assignment operation (=)?

Solution: The assignment operator returns the value to the right of the operator and has a side effect that overwrites the identifier in memory to the left of the operator with the value to the right of the operator.

In addition to operations, functions are able to have side effects as well. Consider the following:

EXAMPLE 2

Write a function in Racket that will consume an integer, x and produce x^2. It has a side effect that outputs "You squared me!" every time the function is called.

```
(define (square_x x)
  (printf "You squared me!")
  (* x x))
```

Furthermore, functions are able to have only side effects. These functions are declared with *void*. Consider the following.

EXAMPLE 3

Write a function *hello_world* that outputs "Hello world!" followed by a new line to the console.

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

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

If you recall above we mentioned it is important to document side effects. All side effects must be properly documented in the POST element of the design recipe. Consider the following.

EXAMPLE 4

Write the documentation for example 2.

```
; 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!".
```

EXERCISE Write the documentation for example 3.

In C

Similar to Racket, you can program in an imperative paradigm in C.

In C we have curly braces ({ }). These braces are known as a **compound statement**, or more commonly a C block. A C block behaves similarly to *begin* in Racket with one notable difference. In Racket, *begin* will evaluate every statement one by one and stop after the last statement is evaluated and produce its value, however in C it will do the exact same thing except it requires a control flow statement to inform the C block that it has reached the end of statements we want evaluated. This special control flow statement is called *return*.

EXERCISE Will the following code block produce an error? If no, state the value of x, otherwise state the reason for the error. You can assume all of the necessary libraries are included.

```
int const x = printf( "Hello world!\n" );
```

6.2 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.
```

Alternatively an *if* statement can be setup as follows, however it is extremely bad practice and should be avoided at all costs! Consider the following.

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

Observe, is the *else* applied to the first or the second if statement? In general we should avoid this. Keep in mind you can have a long chain of *if*/*else if*/*else* statements. Consider the following.

EXAMPLE 1 Write a function *sum* that is passed one argument, n and returns the sum from 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 occuring this semester on Piazza with respect to control flow. The error is "reached end of non-void function". It is caused when you are not using control flow statements properly in a non-void function. Consider the following.

```
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.

6.3 Mutation

Thus far all we have been permitted to use are constants. Recall, constants are immutable, meaning once they are initialized they can never be changed throughout the rest of the execution of the program. How can we initialize an identifier to be mutable?

DEFINITION
Variable

An identifier that is mutable, hence it is able to be overwritten in memory with a new value.

Consider the following.

```
int a = 9; // Note the absence of 'const', hence a is a variable.

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

a = 8; // Overwrites a in memory using the assignment operator.

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

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

EXERCISE

Initialize the variable b to 9. Mutate b such that be is equal to 18.

6.4 Assignment Operator

In C the **assignment operator** = (not ==, the "is equivalent to?" operator) is used to assign a value to an identifier. If you are a Math student you will likely find the following confusing.

```
x = y;

y = x;
```

The first line is assigning the value of y to be the new value of x. The third line is assigning the value of x to be the new value of y. Remark that these are not equivalent operations! In variable declarations we say that the variable, *var* is **initialized** to a value, *v*.

The assignment operator performs two actions. In the example above (line 1), x is being assigned to the value of y. However after it overwrites x in memory, it will then return the value of y. Here is an example of a very confusing but valid block of 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!

6.5 Static Storage

Recall that the *static* keyword when used with global constants and variables restricts them to modular scope. But what happens if we declare a *static* local variable?

DEFINITION
Static Storage

Stores local variables in the global data section of memory as oppose to on the stack. If it is a constant then depending on the compiler it may store it in the read-only section of memory.

Consider 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.

EXERCISE

Write a function *avg_age* that is passed an integer, *age* and will return the average of all the ages passed to the function.

6.6 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 using them then there are a few rules you need to be aware of. 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 arbitrary value from a previous stack frame.

You must **always** initialize variables on the stack!

EXERCISE

Declare a variable *weird* on the stack but do not initialize it. Execute your program 3 or 4 times. What is the value of *weird*?

6.7 Assignment Shortcuts

Ha! I bet this section caught someones eye.

As programmers we are extremely lazy. Conveniently, C provides us with "shortcut" operations that allow us to assign common values or perform common mutation to variables. Consider the following list of shortcut operations for mutating variables.

```
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.
```

Observe the prefix and suffix notation for the ++.

EXERCISE Why could the following code block be problematic?

```
// ...

if ( ( sum == 0 ) && ( count++ == 10 ) ) {
  // ...
}

// ...
```

6.8 Examples of Mutation

Now that we have briefly been introduced to what programming in an imperative paradigm is, let's take a look at some classic examples of imperative programming. Consider the following.

EXAMPLE 1 Write a function *swap* 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.

EXAMPLE 2 Write a function *reflect_point* that will reflect a point in the line $y = x$.

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

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

If you noticed in the previous two examples I am using the keyword *const* in a strange 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.

```
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.

EXERCISE

Write functions *pre_inc*, *post_inc*, *pre_dec* and *post_dec* that are passed a pointer to an integer and perform similarly to $++var$, $var++$, $--var$ and $var--$.

6.9 Looping: Iteration

Iteration is the imperative way of performing recursion. There are cases where recursion may seem more "logical" to look at, however we are expected to understand how to think recursively and iteratively. 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 three control flow statements will iterate until the expression is false. Note, a $do - while$ loop will always iterate once before checking whether the expression is false. Consider the following.

EXAMPLE 3 Write a function *make_simplified_chessboard* that will output a chess board of size n in ASCII.

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

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

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

EXERCISE

Write a function *make_hard_chessboard* that is passed two integers, a row and a column size where they are both congruent to 0 (mod 2). This function will return void and output a chess board of size row by col. Represent black spaces with "**" and white spaces with " " (two spaces).

7 Efficiency: Module 06

This section goes briefly into efficiency. Unfortunately in CS 136 we are only introduced to some of the basics and questions such as "why" may be left unanswered.

Consider efficiency. What is the first word that comes to mind?

DEFINITION
Algorithm

Is a step-by-step description on how to solve a problem.

Algorithms are not restricted to computing. You have devised hundreds of subconcious algorithms that you perform daily. Consider the following.

EXAMPLE 1

What is my algorithm for eating cake?

Solution: My algorithm can be written in the following steps.

1. Grab a slice of cake.
2. Grab a fork and knife.
3. Eat it as fast as possible.
4. As I approach licking my plate, get in line for seconds.

EXERCISE

What is the algorithm you use for brushing your teeth?

Consider the following problem.

EXAMPLE 2

Write a Racket function *nodes >?* that will count all of the non-empty nodes in a valid BST and determine whether there are more than k nodes.

Solution: Consider the following algorithms that solve this problem.

1. Calculate the total number of nodes and compare that number to k.
2. Recurse through the BST. At each node recurse to $k - 1$ until it reaches empty or k becomes negative.

Both algorithms solve the same problem however how do we determine which one is better suited to the task? Define better? How do we **compare** algorithms?

The implementation of both algorithms is as follows.

```
; Algorithm 1
(define (m1/nodes>? abst k)
  (define (height-bst abst)
    (cond
      [(empty? abst) 0]
      [else (+ 1 (+ (height-bst (bst-node-left abst))
                    (height-bst (bst-node-right abst))))]))
  (> (height-bst abst) k))

; Algorithm 2
(define (m2/node>? abst k)
  (cond
    [(empty? abst) #f]
    [(< k (bst-node-key abst)) #t]
    [else (or (m2/node>? (bst-node-left abst) (- k (bst-node-key abst)))
              (m2/node>? (bst-node-right abst) (- k (bst-node-key abst))))]))
```

EXERCISE

Think of alternative algorithms and implement them. Then follow the same analysis we take with respect to the provided algorithms.

7.1 Efficiency

The two most common measures of efficiency are **time efficiency** and **space efficiency**.

DEFINITION
Time

The worst case scenario for how much time it takes for an algorithm to solve a given problem.

DEFINITION
Space

The worse case scenario for how much space is required for an algorithm to solve a given problem.

The efficiency of an algorithm may depend on its implementation. All algorithms are measured on their worst case, this will be properly addressed shortly. Unless otherwise specified, the professors will always be referring to time efficiency.

It is important to quantify efficiency. It may seem trivial, use seconds, however you are wrong. Seconds are not useful to a programmer because there are so many factors that could impact seconds such as, was this 1980 or 2040? Was it on a quantum computer or an iMac? What was the operating system and chip manufacturer? ... with respect to Racket we will quantify efficiency as the amount of substitution steps it takes to solve a given problem.

Revisiting our second algorithm in example 2.

```
(define bst-1 (bst-node 5 "" (bst-node 3 "" empty empty)
                             (bst-node 7 "" empty empty)))

(m2/node>? bst-1 10) ; => 47 steps
```

In C one measure may be how many machine instructions are executed. The problem is that the machine instruction count vary from machine to machine and would be an unreliable source of information. To quantify efficiency in C we will count the number of operations executed.

```
sum = 0;                // 1
i = 0;                  // 1

while ( i < 5 ) {       // 6
  sum = sum + i;        // 2 * 5 = 10
  i = i + 1;            // 2 * 5 = 10
}
```

Note that the expression in the loop is executed 6 times. The 6^{th} execution of the expression is to verify that i is now equal to 5.

7.2 Input Size

Recall our second algorithm in example 2. Did you notice that the number of steps depended on the input size? If there are n nodes in the BST, it will require $14n + 2$ steps to solve the problem. From now on we are always interested in measuring the running time based on the size of the input.

DEFINITION
Running Time

The number of steps or operations with respect to the input, n, that an algorithm requires to solve a problem.

We will denote the running time of a function, T as $T(n)$ where n is the input size. Keep in mind there may also be another **attribute** of the input that is important in addition to size.

Analysis

If you recall we mentioned that time and space effieciency are measured with respect to their worst case. Consider the following input to both algorithms.

```
(define bst-1 (bst-node 10 "" (bst-node 8 "" empty empty) empty))

(m1/node>? bst-1 6) ; T(9n + 4) => 22 steps
(m2/node>? bst-1 6) ; T(15n + 5) => 5 steps
```

The **best case** is when only the first node of the BST is visited and the **worst case** is when all of the nodes are smaller than k and hence all the nodes are visited. How should we decide which one is more efficient?

DEFINITION
Worst Case Analysis

Typically, we want to be conservative (pessimistic) and use the worst case. This is the process of determing the efficiency of an algorithm by comparing worst case scenarios.

Comparing the worst case, we see that $T(9n+4)$ is more efficient than $T(15n+5)$. It may also be important to know the **average running time** however we will not be touching this is not touched upon in CS 136 as it requires further analysis of the algorithm with lot's of data.

7.3 Ordered Runtime: Big O

In practice we are not concerned about the difference between the run times of $T(9n+4)$ and $T(15n+5)$. We are interested in the **order** of a running time.

DEFINITION **Dominant Term**

The term that grows the largest as n approaches infinity.

DEFINITION **Order**

The dominant term in the running time without any constant coefficients. It is also known as the growth rate.

The dominant term in both $T(9n+4)$ and $T(15n+5)$ is n and hence it is of order n, denoted as $O(n)$ in **Big O notation**. The orders that we will need to know:

$O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3)$ and $O(2^n)$.

EXAMPLE 1

Consider the following orders.

- 1994 is $O(1)$.
- $1994 + n$ is $O(n)$.
- $10 + n^2 + n \log n + 1994$ is $O(n^2)$.
- $9 + 2^n + n^3$ is $O(2^n)$.

Pay attention to the fact that the dominant term is the order.

To tie order into efficiency, the algorithm with the lowest order is the most efficient. If we were to compare two different implementations $O(n)$ and $O(1)$, $O(1)$ would be the more efficient implementation.

EXERCISE

Compare the graphs for all of the orders we are responsible for knowing.

Big O Arithmetic

When adding two orders, the larger of the two orders will be the result. Consider the following.

EXAMPLE 2

What is the sum of $O(n)$ and $O(n^3)$?

Solution: Notice that the sum is the $max(O(n), O(n^3))$ which equals $O(n^3)$.

When multiplying two orders, the result is the distribution of both orders. Consider the following.

EXAMPLE 3

What is the product of $O(n)$ and $O(n^2)$?

Solution: $O(n) \times O(n^2)$ equals $O(n^3)$.

EXERCISE

Go through all of your assignments and determine the order of each function.