Lab 2 – Data Types

* 4 data types
  + Decimal (normal base 10)
  + Binary
    - 0b
    - Base 2
  + Octal
    - 0o or just 0
    - Base 8
  + Hexadecimal
    - 0x
    - Base 16
    - A-F 10-15
* Converting, I should know this
* Binary addition
  + 0+0 = 0
  + 0+1 = 1
  + 1+0 = 1
  + 1+1 = 0 (only this has carry out = 1)
  + 1+1+1 = 1 (carry out = 1)
* Signed vs unsigned
  + Signed magnitude = left most bit is the sign
  + 1’s complement = just flip bits (neg nums reps. by inverse)
  + 2’s complement = flip the bits and add 1 ; the good one
    - Ex: 3 bits -> max is 3 (011), min is -4 (100)
* Detect Overflow
  + Only reliable way
    - If carry in does not equal carry out for left most bit, there is overflow
      * So one is 1 and other is 0
      * If same, NO OVERFLOW
* Bitwise Operators
  + NOT ~
  + AND &
  + OR |
  + XOR ^ (exclusive or)
  + A NOR B = NOT (A OR B) = ~(A|B) \*DeMorgan’s
  + A NAND B = NOT (A AND B) = ~(A & B) \*Demorgan’s
  + (A NOT XOR B) = ~(A ^ B)
* Masking
  + To SET a bit, make it a 1
  + To CLEAR a bit, make it a 0
  + With masks, we can
    - Extract specified bits (make rest 0)
    - Clear specified bits (make them all 0)
    - Set specified bit(s)
    - Flip specified bit(s)
* Shifting
  + Left shifting (<<)
  + Right shifting
    - Signed (>>)
    - Unsigned (>>>)
  + \*equivalent to multiplying or dividing by two!
  + Ex: 0111 << 1 === 1110 (you fill in the gaps by the left most bit, ie. If num was negative your replace with 1’s, if positive, replace with 0’s)
* ASCII – American standard code for information interchange
  + Fuck it

Lab 3 – IEEE-754, Digital logic

* Numeric chars to nums
  + ‘0’ == 48
* Convert upper to lower case
  + Add or subtract 32 (upper is first)
  + Change bit 5 appropriately
* IEEE-754
  + Formula: (-1)^S \* 1.M \* 2^(E-127)
  + Sign (1 bit \*bit 31) Exponent (8 bits \*bits 30 to 23) Mantissa (23 bits \*bits 0-22)
* Edge Cases
  + If M==0
    - If E==0
      * 0
    - If E is between not equal to 0 and 255
      * Powers of 2
    - If E == 255
      * Infinity
  + If M!=0
    - If E==0
      * Non-normalized, use a diff formula
        + (-1)^S \* 0.M \* 2^(-126)
    - If E is between not equal to 0 and 255
      * Regular nums
    - If E == 255
      * NaN
* Logic Gates
  + Abstraction for reps groups of transistors
  + 7 gates
    - AND (weird cup sideways
    - NAND like and but circle at tip
    - OR – shark
    - NOR – shark with circle at tip
    - NOT – triangle with circle at tip
    - XOR – shark with line at bottom
    - XNOR – XOR with circle not at tip
  + Convert truth table to sum of products
    - Create a term for each case that results in 1
      * Each term is an AND
      * E.g A=0, B=1, C=0 becomes A’BC’
    - OR all the terms together once finished with above steps
  + Sum of products to truth table
    - Use many AND gates and connect with OR at end
    - So is 8 rows in table, use 8 ANDS and 1 OR
  + Convert a Boolean expression
    - Make a truth table first
    - Test each inputs
    - Build a circuit
  + Convert from a circuit
    - Just look at it piece by piece
    - Start small then go big
    - Build truth table

Lab 4 – transistors, gates, K-maps

* Transitors
  + P-type
    - Have the circle outside
    - Normally closed, so if, 1’d, the floating or undefined.
    - 0’d makes it flow
  + N-type
    - No circle
    - 1 makes power go
    - 0 results in floating
    - Normally open
  + Opposite effect N type vs P type
* DeMorgan’s Law
  + (not A and not B) = not (A or B)
  + (not A or not B) = not (A and B)
* Decoders
  + If there are n inputs bits, then there are 2^n outputs
    - Ex: 3 bits equal 8 possible outputs from 0-7
* Multiplexer (MUX)
  + Selects between inputs via selector
  + If there are 2^n inputs, then there are n selector bits
  + ALWAYS one output
* Logic Conversion
  + Simplifying a circuit to a more simple circuit
  + USE Kmaps
* K maps
  + Simplify Booleans expressions by grouping related terms
  + Karnaugh Maps
  + First, get the truth table
    - In general, vertical axis has two vars, C and C’
    - Horizontal axis has 4 vars, every combo of A and B
      * Ex: AB, A’B, A’B’, AB’
        + CAN ONLY CHANGE ONE VARIABLE AT A TIME FOR AXIS FROM LEFT TO RIGHT
    - Put in out in correct block
  + Make grouping
    - Rules
      * Groups must be rectangular (not necessarily perfect squares, basically groups of 1, 2, or 4, if 8 get help)
      * May wrap around edges
      * Can only contain 1’s or X’s (try not to use X’s)
      * All 1’s must be contained in one group
      * Groups must be as large as possible
      * Size of group must be power of 2. Can have a group of size 1.
  + Write simplified expressions
    - Pull out simplified expression based on K-map groupings
    - OR all terms together at very end
    - If alike, keep. If contradict, remove
      * Ex: (ABC + A’BC) 🡪 BC

Lab 5 – state machines, sequential logic

* Combinational logic
  + Outputs depend only on current inputs
    - Ex: logic gates (7), transitors
* Sequential logic:
  + Outputs depend on both current and previous (memory, clocks, etc.)
  + Latches
* Latches
  + RS latch
    - Basis of sequential logic
    - Remembers 1 bit
      * Ex: if Sn and Rn are both 1, remember
        + 0 and 0 not allowed
  + Gated D latch
    - R-latch with modified interface
    - Output = input only when enable is 1
      * Enable means access it for input to change output like a lock
* Level-triggered logic
  + Ex: RS and D latches
* Edge-triggered logic
  + Clocks
  + Rising-edge triggered logic
    - Output can only change when clock changes from 0 to 1 (transition phase)
  + Falling-edge
    - Other way
  + D Flip-flops
    - Rising edge
    - Left D-latch updates when clock is 0
    - Right D-latch updates when clock is one
    - Made up of 2 d latches
    - Think sandy’s house from spongebob
    - SanD wears flip flops
  + Registers
    - Combine multiple D-flip flops to create a register
    - One D-flip flop for each bit
      * 8 D-flip flops for rep 8 bit num (16 d latches) (or 32 rs latches)
    - Stores multiple bits
    - Need n-bit input or output lines like D-flip flop
* State Machines
  + Has different states that require conditions to be met to go to next states
  + One circle for each state
  + Ex: dancing crab
  + Transition requires some actions like storing power for instance
* One-hot
  + Only one bit is active at a time
  + So for 3 states, need 3 bits
* Binary-encoded
  + Multiple bits are allowed to be “on” or 1’d
  + So if 3 states, need two bits (could also rep max 4 states)
* Address space
  + Number of memory addresses that exist
  + Computer memory is made up of distinct memory addresses, each containing some amount of data.
* Addressability
  + The amount of data stored at any given memory address

Lab 6 – intro to LC-3

* Computer is just a complex state machine
* Look at datapath chart
* Tri-state buffer
  + Choose whether to let a value through
  + Used for GATE
* The bus
  + The fucking line
  + Only one value on wire at time or short circuit
  + 16 bit wire
* Control signals
  + All the LD, GATES, and MUXes
    - LD – write enable for a register
    - Gate – buffer that allows to go onto bus. Basically tsa checkpoints
    - MUX – selector for a multiplexer
* Clock
  + Clock cycles
    - Alternate between 0/1
    - One clock is connected to all registers, keeping sync
    - Ensure no short circuits
    - When register has its WRITE ENABLE on, next clock edge save current input
    - Basically go before MEM is 1, go through MEM is 2, and leave is another. Either 1 3 or 5 clock cycles
* Microcontroller
  + Finite state machine (FSM)
    - The brain
    - Ensures that correct operations take place, while preventing short circuits
    - Every instruction has a series of states to execute it
      * A set of outputs and the next state to go
* ALU
  + The math one
  + Two inputs, two registers SR1 and SR2
  + 00 is add
  + 01 is AND
  + 10 is NOT
  + 11 is PASS
  + On the chart
  + Drives output onto bus
  + Condition codes (CC)
    - It is only set or accessed or used when writing to general purpose registers, basically saving result in a register
* What is Memory?
  + Registers vs Memory (RAM)
    - Registers
      * Quick data storage on processor
    - Random Access Memory
      * Memory like a really big array
      * Contains instructions for programs that are currently being executes
      * Stores any data a program may need
      * When opening complex, there is a bunch of random instructions. That reps the memory
      * Two sides of memory
        + Address side
        + Data side (or data @ address)
* Loading from Memory
  + Place address where data is located in MAR (memory address register)
  + Memory is read at MDR (Memory Data Register)
  + Two entrances, one exit
* Storing into Memory
  + Data from a register is placed into MDR
  + Address of where to store data is placed into MAR
  + Data is stored in memory at specified address by MAR
* PC
  + Program counter – tracks address of NEXT instruction to execute, always one step ahead essentially
  + Update
    - PC+offset
    - PC + 1
    - Given a value via the bus
* IR instruction register
  + Hold value of the current instruction
  + Look at ops handout
  + Reads the 16 bit number to do shit
* Execute instructions
  + Macrostates or stages
    - 1. Fetch (3 clock cycles)
      * Load next instruction from memory into instruction register (IR)
      * Increment PC
    - 2. Decode (1 clock cycle)
      * Microcontroller looks at instruction to determine instruction to do
    - 3. Execute (varied)
      * The steps to do shits

Lab 7 – LC-3 Datapath

* You should know what fetch to fucker
  + Start at pc, do shit there, ride the bus to MAR, go through memory, MDR, back on bus, end at IR
* Decode
  + If needed, LC-3 takes care of it really, from IR along with NZP condition codes if needed, go to FSM
* Execute
  + Differs
* LC-3 ALL OPERATIONS
  + ADD
    - Opcode: 0001
    - Two formats (reg and immediate or imm)
      * Bit 5 specifies format
      * Adds two args and stress result in destination registers (DR)
        + DR = SR1 + SR2
        + Ex: ADD R0, R1, R2
    - ADD immediate
      * Same thing but third spot is add number
      * Use # to clarify using decimal and not hex
      * DR = SR + imm5 (bc 5 bits)
      * Max is add 15 or min is minus 16
      * Ex: ADD R0, R1, #8
  + AND
    - 0101
    - Two formats
    - DR = SR1 & SR2
    - Ex: AND R0, R1, R2
    - Can use to test condition codes
    - AND immediate
      * Good for clearing register
      * DR = SR & imm5
      * Ex: AND R3, R3, #0
  + NOT
    - 1001
    - DR = ~SR
    - Flips all bits
    - Ex: NOT R1, R0
  + All three above set condition codes
  + LD
    - 0010
    - Basic load
      * Loads value at memory address PC incremented (or PC\*) + PCOffset9 and puts it in destination register
      * We get that bread
    - DR = mem[PC\* + PCOffset9]
    - PCOffset9 is a 9-bit 2’s complement value
    - Ex: LD R0, #5
      * Reads value at location in memory addressed by PC\* + 5
  + ST
    - 0011
    - Stores instruction and is similar to LD
    - Stores value in the source register at memory address PC\* + PCOffset9
    - mem[PC\* + PCOffset9] = SR
    - ST R0, #5
      * Stores what’s in R0 at mem address PC\* + 5
  + LDR
    - 0110
    - load base + offset instruction
    - calculate its address by adding a register’s value to a 6-bit 2’s complement numbers that comes from the instruction
    - DR = mem[BaseR + offset6]
    - Sets condition codes
    - LDR R0, R5, #0
    - Good with arrays
      * Ex: if R5 were the address of the start of the array, this instruction would put first element of array into R0.
  + STR
    - 0111
    - Store Base + Offset
    - Stores value from source register at memory location given by sum of base register’s value and the immediate offset.
    - Mem[BaseR + offset6] = SR
    - STR R0, R5, #6
      * So basically R5 + 6 is the address where you will store the data that is R0 in memory
  + LDI
    - I for I do it again
      * Or indirect
    - 1010
    - DR = mem[mem[PC\* + PCOffset9]]
      * Or DR = mem[LD]
    - Do it again
      * The small circle around memory
    - Sets condition codes
      * All loads set condition codes EXCEPT for LEA
  + STI
    - Same
    - mem[mem[PC\* + PCOffset9]] = SR
    - yeah
  + LEA
    - It exists
    - 1110
    - Load effective address
    - Good for labels
    - DR = PC\* + PCOffset9
    - This one does NOT go through memory
  + BR
    - Go brrr
    - Branch, not break
    - 0000
    - Reads condition codes like an if statement and test for nzp
    - Ex: BRzp ESCAPE would check if such register is 0 or positive. If so, branch jump to there. What directly follows this is not zero or positive.
  + NOP
    - Fuck this one
    - Has a 0 in each of NZP
    - Fuck it
  + Basically any instruction that updates register file except the loser LEA
* Labels
  + Kind of like variable names
  + Can use as a parameter in assembly code
  + Can hold data values or memory addresses
  + Not stored anywhere in memory, instead assembler calculates the right numerical offset to get to desired label
    - To calculate offset yourself, find difference between memory address of label and incremented address of instruction, which will be value stored in PC when it executes
* LC-3 Pseudo-ops
  + Not instruction but directions to assembler
  + .orig – tells assembler to put block of code at desired address, start code here
    - .orig x3000
  + .stringz – assembler will put a string of characters in memory followed by “\0” null terminator
    - Keep in mind due to length is word length + 1
      * .stringz “something”
  + .blkw – allocated memory for specified label
    - .blkw 10 (reserves 10 memory spaces)
  + .fill – puts value in specified memory location
    - .fill x4040 (put something there)
  + .end – end of block, closing tag for .orig

Lab 8 – assembly debugging with complx

* Common assembly techniques
  + Look at lab 8
  + Subtraction/negation
    - NOT R3, R1 ; R3 = ~R1
    - ADD R3, R3, #1 ; add 1 to make -R1
    - ADD R3, R2, R3 ; R3 = R2 + (-R1) = R2-R1
  + Copy a value to another register
    - ADD R2, R1, 0 ; R2 = R1
  + If statements
    - if (Rx cc 0) {
      * //if code
      * } else {
      * //else code
      * }
    - End if code
  + Branch to the if code if Rx meets condition, otherwise go to else
  + Can extend to multiple conditions for if-else-if
  + Ex:
    - ADD Rx, Rx, 0
    - BR(cc) if code
    - BR else code
  + At the end of each segment, go to the END if code
  + Ex:
    - If\_code:
      * ;some stuff
      * BR END\_IF
    - Else code
      * ;some stuff
      * BR END\_IF
  + While loops
  + While(Rx cc 0) {
    - While code
    - }
    - Can be used to make for loops
  + Ex:
  + While\_loop:
  + ADD Rx, Rx, 0
  + BR (!cc) END\_WHILE CODE
  + ; WHILE CODE here
  + ;recheck condition when done
  + BR WHILE\_LOOP
* TRAPs
  + Subroutines built into LC-3 to simplify instructions
  + Like normal instructions but are aliases to TRAP calls
    - Ex: HALT == TRAP x25
  + Has corresponding 8-bits TRAP Vector
  + Ex:
    - HALT (x25) stops running program
    - OUT (x21) takes character in ASCII in R0 and prints it to console
    - PUTS (x22) given mem address in R0, print characters until NULL terminator (‘\0’)
    - GETC (x20) takes character input from console and stores it in R0
* Complx
  + Debugger

Lab 9 – Subroutines and The Stack

* Assembly subroutines
  + Another name for function but not really
* JSR
  + Jump to subroutine
  + Saves PC\* value into R7 and then set PC\* to the target
  + R7 = PC\*
  + PC = PC\* + PCOffset11
* JSRR
  + Jump to subroutine, register
  + Same as JSR, but uses a base register instead of a label/offset
* THE stack, look at tl03 solution

Lab 10 – intro to C

* Hell incarnate
* Compilation overview
  + Step 1. Preprocessor
    - Runs before the compiler
  + Step 2 compiler
    - Convert C into assembly
  + Step 3 Assembler
    - Assembly
  + Step 4 Linker
    - Link multiple files, subroutines, etc. together
* Macros
  + Ex:
    - #define MACRO\_NAME(ARGUMENTS) TEXT\_REPLACEMENT
    - #define MULT(A,B) ((A) \* (B))
    - #define SUPER\_SQUARE\_A(A,B) (((A)\*(A) + (B)\*(B)) \* 1.0f)
    - #define PI 3.141593
  + So if preprocessor sees MULT(5,7) somewhere in C file, it will replace it with ((5\*7)) PI will be replaced with 3.141593
* Header Files
  + In C, you can’t use a function/variable before you declare it
  + Header files contain function declarations and global variables
  + Must declare in header first
  + Has a .h extension
  + Like the “interface” of what a file exposes
  + Shouldn’t include function implementations ever
* #include
  + C does not have an “import” system
  + Preprocessor copies all C code from filename and replaces the include statement with that code
  + #include-ing a header file is like copy-pasting the declarations you need
  + Generally, never #include a .c file
  + #include <filename> for system header files
  + #include “filename” for header files you write
* Include guards
  + What happens if I #include the same header file twice
    - You get a compiler error due to multiple declarations
    - Easily happen if a file includes two other files that share a dependency
  + Can solve this by putting this in our headers file to ensure each header file only gets included once
  + #ifndef <HEADER\_FILE\_NAME) \_H
  + #define <HEADER\_FILE\_NAME>\_H
  + Contents of header file
  + #endif
* C Data types and sizes
  + Unlike in Java, the exact sizes of number types is not specified in the C standard
  + There are minimums in the standard as well as some conventions
  + Exact sizes are implementation-specific and may vary between compilers or systems-you should never assume a given type is a given size
    - Except for char, which is 8 bits
  + Can use sizeof() op to determine the size of a C type in bytes
  + Ex:
    - Char ALWAYS 8 bits, 1 byte
    - Everything else is subjective

Lab 11 – storage, and pointers, and arrays, and strings

* Memory regions in C
  + Like LC-3
  + Top is the Stack – holds local variables and arguments stored here
    - Like Lc-3
  + Heap – dynamically allocated memory
    - Related to allocating a new object
    - For like malloc stuff
  + Data – global variables are stored here
    - Like labels in assembly
  + Code
    - Program text
  + Ex:
* #include <stdio.h>
* int \*getNthElement(int n) {
* static int nums[] = {48, 45, 69, 2, 12, 61, 4, 58, 42, 82};
* char letter = 'U';
* int myNum = nums[n];
* return &myNum;
* }
* int count = 8348;
* int main(void) {
* int \*x = getNthElement(6);
* printf("%d\n", \*x);
* return 0;
* }
* Blank **[1]** corresponds to where **getNthElement** is stored - code
* Blank **[2]** corresponds to where **nums** is stored - data
* Blank **[3]** corresponds to where **count** is stored -data
* Blank **[4]** corresponds to where **x**is stored -stack
* Blank **[5]** corresponds to where **n**is stored – stack
* Need to explicitly need malloc for heap usage, if myNum involved heap, then myNum would be on the heap
* Global Variables

Ex:

int x = 4;

int main(void) {

int y = 3;

return 0;

}

* x is a global variable stored in the “data” section of memory
* y is a local variable stored on the stack (specifically on the stack frame for main)
* Type Qualifies: static
  + Three different ways
    - Static defined function (like private in java) not visible outside of its C file
    - Static defined global variable (like private in java) not visible outside of its C file
    - Static defined local variable do not lose values between function calls
      * Local vars normally stored on the stack, so we instead put it on code
* Type qualifiers: const and extern
  + Const: defines a variable as constant
    - Like final in java
    - Ex: const int x = 5;
  + Extern: tells compiler that the variable has been defined in another file
    - Ex: extern int x;
    - It’s other.c defined as
      * Int x = 5;
* Pointers
  + Variables that contain a memory address
    - Have a type
    - Refers to the type of the data AT that memory address
    - Also a special case: void pointers point to a memory address but there is no known type for the data at the address
    - Denoted by an \* following the type
  + Ex:
    - Char \*x; //declare that x is a pointer to char
    - Char \*\*y; //declares that y is a pointer to a pointer to a char
    - Void \*z; //declares that z is a pointer to an unspecified type
  + & used to find the address of a code element
    - Char I = 97; //I stores value 97 or ‘a’
    - Char \*x = &I; //x stores the address of I
    - Char \*\*y = &x; //y stores the address of the variable x
    - Void \*z = &I; //z stores address of variable i
* Dereferencing pointers
  + Pointer points at or refers to memory value at address contained in pointer
  + Can use operator \* to dereference pointer, to get the value
  + So
    - Char c = 97;
    - Char \*x = &c; x stores memory address of c
    - X == 97; //address in x is not 97
    - \*x == 97; //yes, the value at x is 97
    - \*x = 0; the value at address x is set to 0 or null
    - X = 0; x is now a null pointer and will segfault if deferenced again or \*x again
* Arrays
  + Like assembly, fixed-sized of same typed elements in a sequence
  + Laid out consecutively in memory
    - Int arr[5]; //arr is an array of 5 ints
    - Char arr[] = {‘A’, 66, ‘C’}; // arr is an array of 3 chars
    - Int \*arr[3]; //arr is an array of 3 pointers to int
  + Unlike in Java, cannot count on uninitialized data to be NULL or 0, simply un-initialized
  + Like in assembly, need to know set length and first element
    - Need pointer to first element
* Arrays decay to pointer: can implicitly convert an array into a pointer to the memory address of its first element
  + Arr == &arr[0]
  + arr[0] = 7; == \*arr = 7;
* but they are not the same:
  + arrays are fixed-size, and they refer to specific region of memory. Cannot make an array point elsewhere once it is initialized.
    - We can reassign where it points to in memory
* Recall: sizes of C types
  + C has a special sizeof(type\_t) operator that return the size of a type
  + Expressed in bytes
* Pointer arithmetic
  + Way that C treats adding and subtracting to/from pointers; add offset times size of the pointer type
    - Int \*y;
    - Y + 2 evaluates to y + 2\*sizeof(int)
    - Y[2]
    - Also applies to arrays
      * Ex: int arr[4];
      * Arr[3] = 12; and \*(arr + 3) = 12 are equivalent
* Strings
  + In C, strings make use of all concepts above
  + Basically an array of chars with a null terminator at end ‘\0’
  + Keep in mind when looking at length

Lab 12 – more C

* Structs
  + A struct (structure) is a data type that is used to group together (aggregate) multiple values
  + Looks similar to classes in Java; however they do not have methods – like enums
  + Use . operator to access the vars inside a struct
  + If given a pointer to a struct, USE THE -> operator to simultaneously dereference and access it
    - Ex:
      * struct Dog { //stored in Data segment
      * char name[20];
      * int isGood; //we use ints for Boolean values in C
      * };
      * How this looks in memory is that each char in the name takes up 1 byte or 1 address block, but 20 are set aside. So if address starts at x4000, name covers for x4000-x4013 bc hex, yes there is such thing as x400C
      * Then x4014 has isGood which is an int, and if sizeof(int) is 4, then the next available block for another var would be at x4018
* C Pitfalls
  + Keep in mind that local vars are stored on stack.
  + If you want to permanently change the value in the array in a struct Dog for instance
    - Ex: struct Dog dogs[10];
  + Then you need to call the address of dog and dogs and copy by an 🡪 to dereference to actually edit it
    - Ex:
      * Int main(void) {
        + Struct Dog \*dog = &dogs[5];
        + Strncpy(dog->name, “Doggo”, 6);
        + Dog->isGood = 1;
  + Ex:

Int main (void) {

Int x = 2109;

Func(x);

Printf(“%d\n”, x);

}

Void func(int num) {

Num = num + 1;

}

* It won’t print out 2110, bc the param of the func call is not \*x, meaning we did not pass by reference, but instead pass by value. We want actually affect.

Lab 14 – type declarations

* Ex:
  + Int (\*matrix)[8]; // matrix is a pointer to an array of 8 ints
  + Int (\*add)(int, int); // add is a pointer to a function that take in two int params that return an int
* Derived types
  + All types have:
    - Exactly one base type: int, char, struct coord, etc.
    - Zero or more derived types:
      * \* - “pointer to…”
      * [] – “array of…”
      * [n] – “array of n …”
      * …(…)(args) – “function taking args and returning …”
* You know how to read. Start at name read far right until ) then far left until ( then repeat until done

Ex:

* int \*(\*foo[5])(char\*);
  + “foo is an array of 5 pointers to a function taking a char pointer as a parameter in and returns a pointer to an int”

Lab 15 – dynamic memory

* Dynamic memory allocation
* The heap!
  + Sometimes we write programs expecting user input or input from a source we don’t control like a file
  + The heap holds dynamically allocated variables that have space allocated at runtime, like creating a new object
* Malloc
  + If we want to dynamically allocate memory, we can malloc()
  + Takes in one arg: how many bytes to allocated from the heap
  + Returns a pointer to the allocated memory, or NULL if there’s an error
  + Ex:
    - Int \*myInt = malloc(sizeof(int));
      * MyInt now points to a block of four bytes on the heap, if int is 4 bytes big. MyInt becomes a pointer and the stuff it points to is on the heap
* Free()
  + C’s version of garbage collection
  + Free() tells C that it can give the memory to other purposes
  + Never use a freed pointer! Just don’t
  + Ex: free(myInt); //the arg it takes is a pointer to a spot in heap
* Calloc()
  + Malloc but immediately zeroes out the memory
    - Two args, num of elements and size of each element,( good for arrays!)
  + int \*array = calloc(array\_len, sizeof(int))
* Realloc()
  + If we want to grow or shrink the size of a block on the heap, such as resize array
  + Also preserve the data present
  + Ex: int \*first5 = realloc(numbers, 5\*sizeof(int));
  + First5 is a pointer now to the first 5 ints of the “array” numbers
  + Note that numbers pointer is NO LONGER VALID
* Uses of malloc
  + Malloc and friends are our ways of interacting with dynamic memory or heap
    - Ex:
      * Dynamically allocated arrays (1D, or multi-D)
      * Dynamic data structures like linked lists
      * Large data structures that won’t fit on stack
      * When data needs to survive after the function end.

Lab 16 – Malloc error handling

* Error handling
  + int \*get\_array(void) {  
     int \*ptr = calloc(4, sizeof(int));  
     if (!ptr) return NULL;  
     ptr[2] = 7;  
     return ptr;  
    }  
    Solution: always check the result of malloc & friends! If it returns NULL, handle it appropriately
  + int \*get\_array(void) {  
     int \*ptr = malloc(4\*sizeof(int));  
     if (!ptr) return NULL;  
     ptr[2] = 7;  
     int \*new\_ptr = realloc(ptr, 6\*sizeof(int));  
     if (!new\_ptr) { free(ptr); return NULL; }  
     ptr = new\_ptr;  
     ptr[5] = 12;  
     return ptr;  
    } Solution: always assign the result of realloc to a new pointer!
  + struct dinosaur \*create\_dino(char \*name) {  
     struct dinosaur \*dino = malloc(sizeof(struct dinosaur));  
     if (!dino) return NULL;  
     dino->name = name;  
     dino->coolness = 11;  
     return dino;  
    }  
    Solution: If you want your struct’s data to be on the heap, you must deep copy any user-provided pointers (including   
    strings) first
  + struct dinosaur \*create\_dino(char \*name) {  
     struct dinosaur \*dino = malloc(sizeof(struct dinosaur));  
     if (!dino) return NULL;  
     dino->name = malloc(strlen(name) + 1);  
     if (!dino->name) { free(dino); return NULL; }  
     strcpy(dino->name, name);  
     dino->coolness = 11;  
     return dino;  
    }