ECE 220 Final Exam

Joseph Ravichandran, Kanad Sarkar, Srijan Chakraborty

HKN Services

- Offer review sessions for most required ECE and PHYS
- HKN offers peer-to-peer tutoring for ECE 220 (As well as most required ECE, CS, MATH and PHYS courses)
- https://hkn.illinois.edu/service/
  - Scroll to tutoring and pick anyone for one-on-one tutoring
  - Contact us directly! All netIDs provided!

- Part 1: LC-3
  - Assembly language programming & process
  - Memory-mapped I/O: input from keyboard, output to monitor
  - TRAPs & Subroutines
  - Stacks
- Part 2: C
  - Built-in data types, operators, scope
  - Functions & run-time stack
  - Pointers & arrays
  - Recursion: searching, sorting, backtracking
  - I/O: streams and buffers, read from / write to file
  - User-defined data types: enum, struct
  - Dynamic memory allocation
  - Linked data structures: linked list (stack, queue) & trees
- Part 3: C++
  - Class (encapsulation, inheritance, abstraction)
  - Virtual function, operator overload, template (polymorphism)
  - Pass by value /(const) reference / address

THINGS YOU SHOULD HAVE MEMORIZED (OR HAVE ON YOUR CHEAT SHEET)

- Basic LC –3 (datapath, basic functions, traps, I/O, etc.)
- The Stack setup and teardown slides
- File I/O in C
- The four qualities of object oriented programming
- The difference between bubble sort and insertion sort

LC3 I/O

- KBSR
- KBDR

Functions in C

- The function prototype or declaration:
  - Name (identifier)
  - Return type or output
  - Arguments or inputs and their types
  - If not void, MUST return something
- Provides abstraction

`Example: int isPrime(int n)`

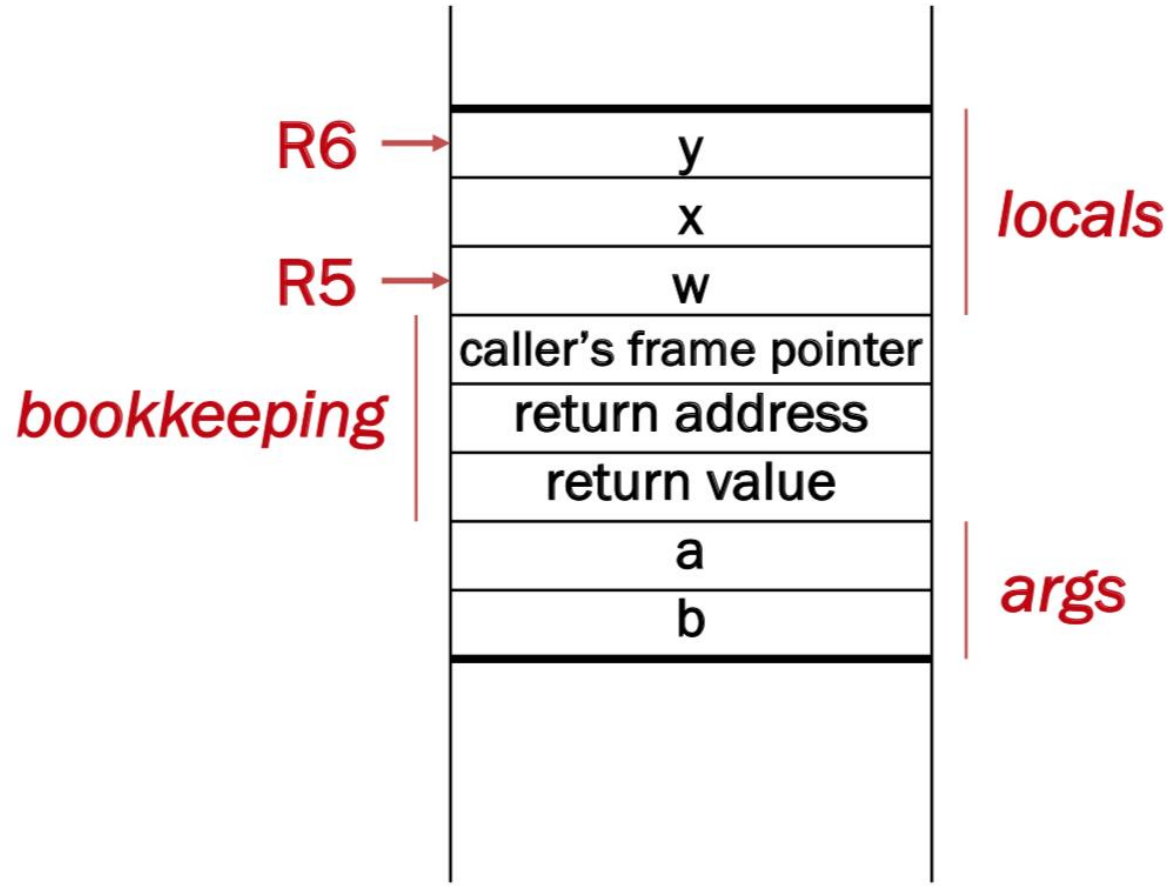
- Hide low-level details
  - Give high-level structure to program, easier to understand overall program flow
  - enable separable, independent development
  - reuse code

The C Runtime Stack

Used by the compiler to keep track of variables and memory

- R5 – **Frame Pointer**. It points to the beginning of a region of activation record that stores local variables for the current function.
- R6 – **Stack Pointer**. It points to the top most occupied location on the stack.
- Arguments are pushed to the stack **RIGHT TO LEFT**
- Local variables are pushed to the stack in the order declared

```
int func(int a, int b)
{
   int w, x, y;
   .
   .
   .
   return y;
}
```



Stack Build-Up and Tear-Down

Caller function

1. *caller setup*: push callee's arguments onto stack
2. pass control to callee (invoke function)

Callee function

3. *callee setup*: push bookkeeping info and local variables onto stack
4. execute function
5. *callee teardown*: pop local variables, caller's frame pointer, and return address from stack
6. return to caller

Caller function

7. *caller teardown*: pop callee's return value and arguments from stack

Register Usage

R5: Stack Frame Pointer

R6: Stack Top Pointer

R7: Return Address

Callee Setup in 4 steps!

```
ADD R6, R6, #-4      ; Allocate space for linkage and 1 local variable (to ensure R5 is valid)
STR R5, R6, #1       ; Save old value of R5
ADD R5, R6, #0       ; Set R5 to new frame base
STR R7, R5, #2       ; Save return address
```

What would happen if we did not add space for 1 local variable? In other words, R5 was pointing to a location above R6?

R5 would be pointing to memory outside of the stack, and the stack data structure's integrity would be ruined.

Callee Teardown in 4 steps!

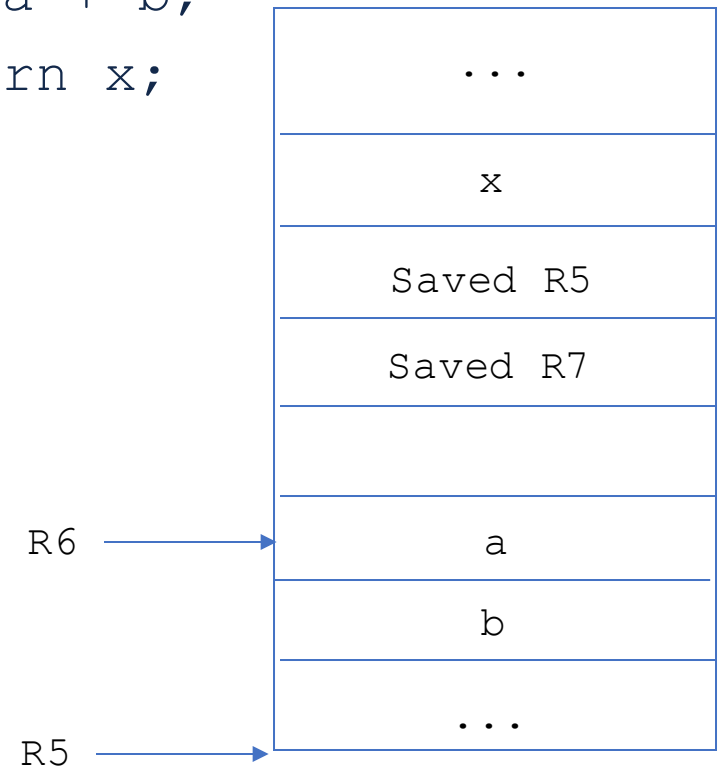
```
ADD R6, R5, #3      ; Have R6 point to return space (3 below R5)
STR R0, R6, #0      ; Push return value into return spot (If R0 has value)
LDR R5, R6, #-2     ; Push old stack frame back into R5
LDR R7, R6, #-1     ; Load old return address back into R7
```

Basic trick is pop R6 4 times in one instruction, then reach at the rest of the required variable

When coding, don't forget to RET after done in JSR

Callee Example

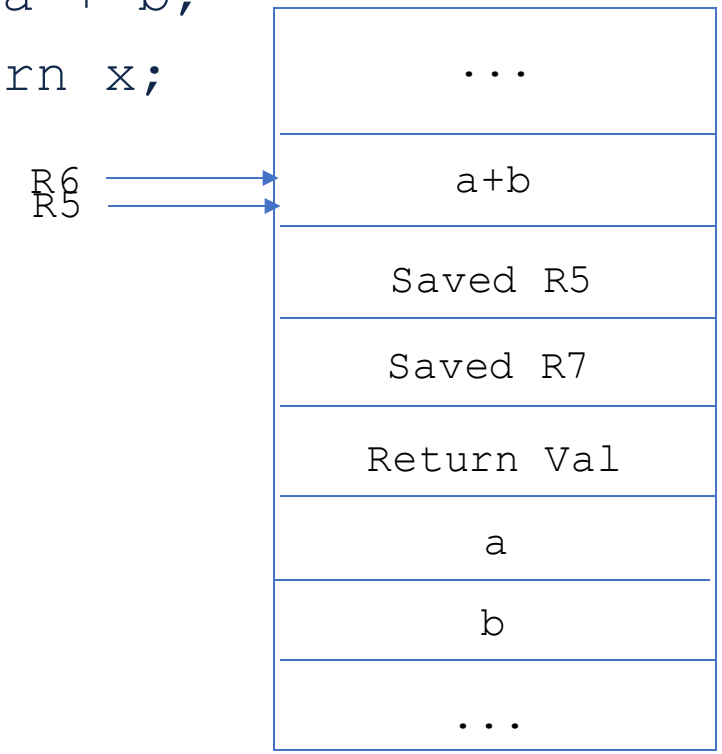
```
int foo (int a, int b) {
      int x;
      x = a + b;
      return x;
}
```




```
; Bookkeeping creation
ADD R6, R6, #-4; Make space on stack
STR R5, R6, #1 ; Store R5
ADD R5, R6, #0 ; Set R5 to new frame
STR R7, R5, #2 ; Store return address

; Calculation
LDR R1, R5, #4 ; Load a into R1
LDR R2, R5, #5 ; Load b into R2
ADD R0, R1, R2 ; Store result into R0
STR R0, R5, #0 ; Store R0 in x

; Teardown frame & return
STR R0, R5, #3 ; Store R0 as ret val
LDR R7, R5, #2 ; Restore R7
LDR R5, R5, #1 ; Restore R5
ADD R6, R6, #3 ; Teardown stack,
                                    leaving return
value
```

Callee Example

```
int foo (int a, int b) {
      int x;
      x = a + b;
      return x;
}
```




```
; Bookkeeping creation
ADD R6, R6, #-4; Make space on stack
STR R5, R6, #1 ; Store R5
ADD R5, R6, #0 ; Set R5 to new frame
STR R7, R5, #2 ; Store return address

; Calculation
LDR R1, R5, #4 ; Load a into R1
LDR R2, R5, #5 ; Load b into R2
ADD R0, R1, R2 ; Store result into R0
STR R0, R5, #0 ; Store R0 in x

; Teardown frame & return
STR R0, R5, #3 ; Store R0 as ret val
LDR R7, R5, #2 ; Restore R7
LDR R5, R5, #1 ; Restore R5
ADD R6, R6, #3 ; Teardown stack,
                                    leaving return
value
```

Caller Example

```
int main () {
     int x;
     int result;
     result = foo(x);
}

int foo (int a) {
     ...
}
```

```
LDR R0, R5, #0 ; Load x from stack frame of main

ADD R6, R6, #-1
STR R0, R6, #0 ; Push R0 onto the stack

JSR foo ; Jump to foo

; Note: After the call to foo R6 has been
; decremented by 1!

LDR R0, R6, #0 ; Read the return value

ADD R6, R6, #2 ; Pop the parameters & return value
                 from function call

STR R0, R5, #1 ; Store returned value into result
...
```

POINTERS!!!!

- This slide was sponsored by POINTER GANG

Pointers

- Dereference Operator: *
  - Returns the data that the pointer points to
- Address Of Operator: &
  - Returns the address in memory of the object applied on
- Shorthand Dereference & access operator: ->
  - pointer->member is equivalent to *(pointer).member
  - Good for use with struct pointers
- Value is an LC3 address (x3000, xCAFE, xBABE)

Pointers

- Pass by pointer VS pass by value
  - Former allows you to change actual object in memory by dereferencing the pointer, latter is just a bitwise copy
- Pointer math depends on size of the pointer type
  - If char* a is x3000, a + 3 is x3003
  - If int* a is x3000, a + 3 is x300c

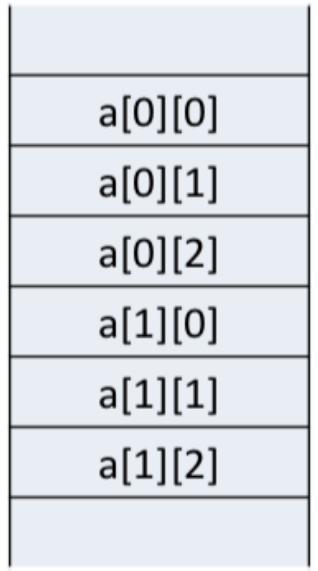
Arrays

- Pointer to several blocks of memory.
- If int a[#], a is a pointer to the FIRST element
- arr[x] operator is same as *(arr + x)
  - Basically gets you to starting address of object at x
- Stored sequentially in contiguous memory
- When passed to function, only pointer to first element is passed
- Arrays cannot be passed by value

Multi Dimensional Arrays in C

int a [2][3];

	Column 1	Column 2	Column 3
Row 1	a[0][0]	a[0][1]	a[0][2]
Row 2	a[1][0]	a[1][1]	a[1][2]



Stored in memory in the Row Major Format

`i*(number_of_columns) + j = element at i,j`

Can be applied to higher dimensions!

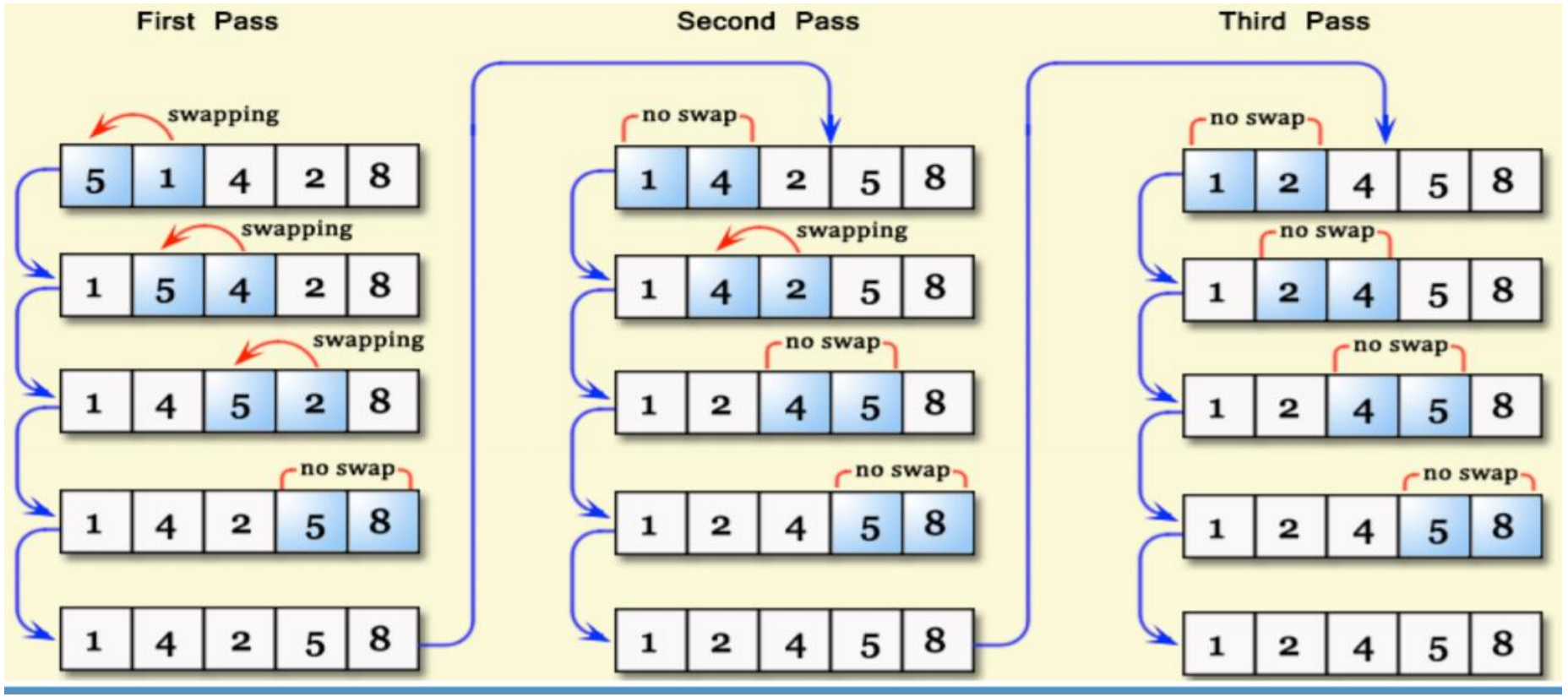
Searches

- Linear Search: iterates from the start of the array till the item is found.
- Binary Search:
    1. Find the middle and check if it is the item
    2. Search first half if desired item is smaller than middle, else check second half
    3. Repeat 1 and 2 until found

Sorting

Bubblesort: Most basic (and slow) algorithm

(Check EVERY element for EVERY spot)

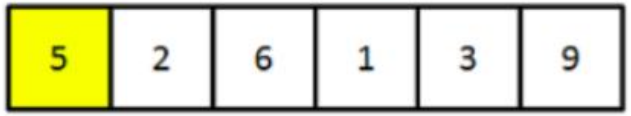


Insertion Sort

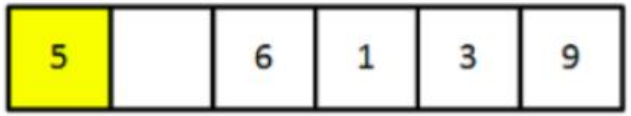
1) remove item from array, insert it at the proper location in the sorted part by shifting other items;

2) repeat this process until the end of array is reach.

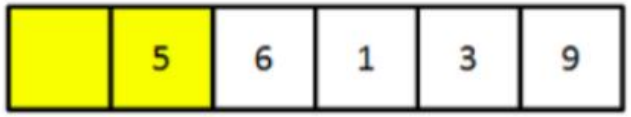
Step 1
Assume first item is "sorted"



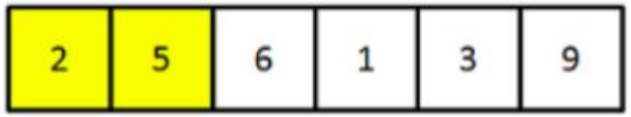
Step 2
Identify the value to compare



Step 3
Since 5 > 2, shift 5 over to create space for 2 in the sorted section



Step 4
Insert 2 into the empty space in the sorted section



Recursion

- Whenever a function calls itself
- Builds a runtime stack frame every call
- Always include a base case
- Recursive case should make problem smaller

Recur_Function(2)
Recur_Function(1)
Recur_Function(0)
Main

Recursion and the Idea of Backtracking

- Recursion: Decompose a bigger task into smaller tasks and combine them using known rule or trivial cases
- Recursion + Backtracking: Guess to create smaller tasks, detect when impossible; guess again
- Look at solve_sudoku in mp7 and N_queens example in lecture slides

Structs

Allow user to define a new type consists of a combination of fundamental data types (aggregate data type)

Example:

```
struct StudentStruct {
        char Name[100];
        int UIN;
        float GPA;
};
```

To access a member of a struct, use the "." operator:

```
struct StudentStruct my_struct;
my_struct.UIN = 123456789;
```

To access a member of a struct pointer, use the "->" operator:

```
struct StudentStruct *my_struct;
my_struct->UIN = 123456789;
```

Typedef

Allows you to refer to a struct without having to specify ‘struct’ keyword each time

Example 1 (Out of line):

```
struct StudentStruct {
      ...
}

typedef struct StudentStruct Student;

// Allows you to use ‘Student’ as an alias to ‘struct StudentStruct’
```

Typedef

Allows you to refer to a struct without having to specify ‘struct’ keyword each time

Example 2 (Inline typedef):

```
typedef struct StudentStruct {
      ...
} Student;

// Allows you to use ‘Student’ as an alias to ‘struct StudentStruct’
```

File I/O in C

FILE* fopen(char* filename, char* mode)

//mode: “r”, “w”, “a”, ...

success-> returns a pointer to FILE

failure-> returns NULL

int fclose(FILE* stream)

success-> returns 0
failure-> returns EOF

int fprintf(FILE* stream, const char* format, ...)

success-> returns the number of characters written failure-> returns a negative number

int fscanf(FILE* stream, consta char* format, ...)

success-> returns the number of items read; 0, if pattern doesn’t match

failure-> returns EOF

int fgetc(FILE* stream)

success-> returns the next character
failure-> returns EOF and sets end-of-file indicator

int fputc(FILE* stream)

success-> write the character to file and returns the character written

failure-> returns EOF and sets end-of-file indicator

char* fgets(char* string, int, num, FILE* stream)

success-> returns a pointer to string

failure-> returns NULL

int fputs(const char* string, FILE* stream)

success-> writes string to file and returns a positive value

failure-> returns EOF and sets the end-of-file indicator

DYNAMIC MEMORY

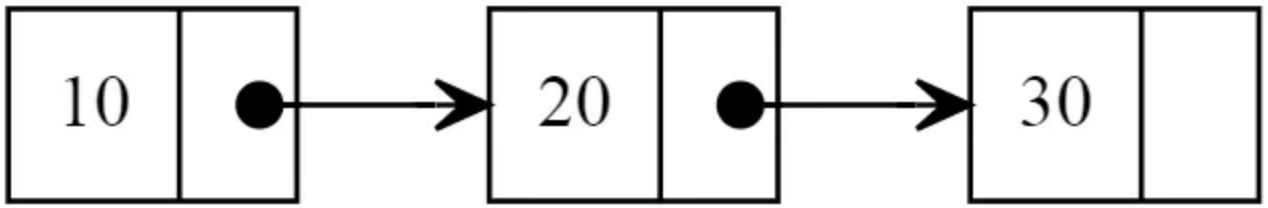
- The way memory works is that we have the runtime stack and a heap
- If we want something to persist, we put it on the heap
- Use malloc family of functions to allocate space on heap

```
char *ptr=(char*)malloc(sizeof(char)*15);
```

- Useful for making memory you must fill/access later (dynamically!)
- Use free() to delete memory on heap
  - free can ONLY be used on a pointer returning memory on the heap
- Everything allocated on heap MUST BE DELETED (using free)
  - Else we have leaked memory

LINKED LISTS

- Whole new way to store data by taking advantage of structs and pointers!
- Consists of "nodes" with data and a pointer to the next node
- "Nodes" are represented with a struct! With a data member and a "Node" pointer member
- Can't access anything directly (no access to whole data structure), BUT can iterate node-by-node
- Keep track from the pointer to the first node ("head" pointer)



Linked List

Linked List Structure

Built of units called "nodes" with data and a pointer to next node in list

All we really need is a permanent storage to the first node, in non-circular lists last node points to NULL

To iterate, we hold a temporary node pointer, and use it to go to the next node in the list
e.g:
ListNode* temp = head;
while(temp != NULL)
    temp = temp->next;

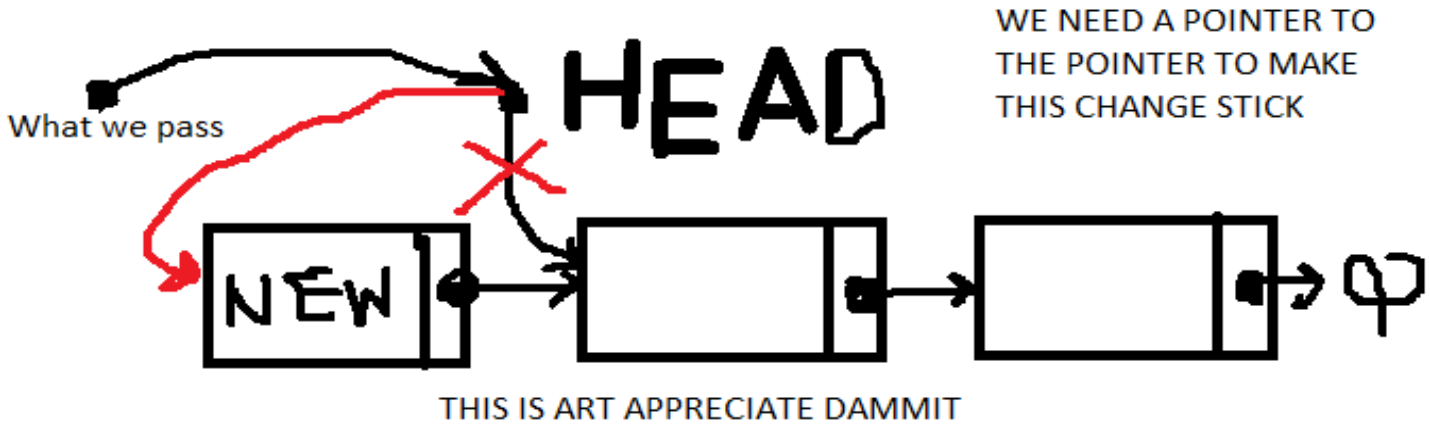
In our list itself, we only need to hold the first node, anything else depends on implementation

e.g.: In this example we happen to hold the last node too.

```
1   typedef struct ListNode {
2     struct ListNode* next;
3     int data; //Just happens to be int
4   }
5
6   typedef struct List {
7       int count;
8       ListNode* first;
9       ListNode* last;
10  } List;
```

A NOTE ON PASSING A DOUBLE HEAD POINTER

- IMPORTANT: If we'd need to change a head node, pass a double pointer to the head node to our function so we could change the head node itself if needed
- e.g: void insertAtFront(ListNode** head, int data);

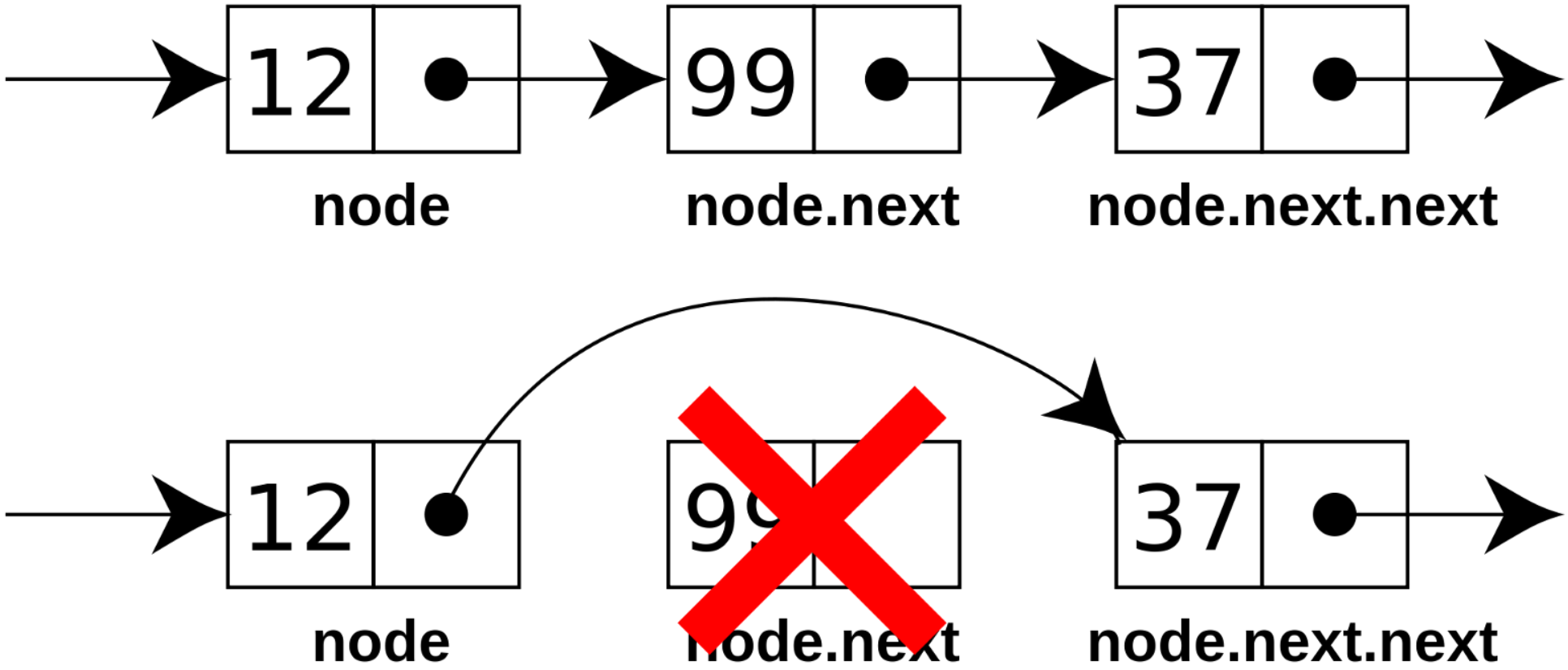


HOW TO DO (SINGLY) LINKED LISTS

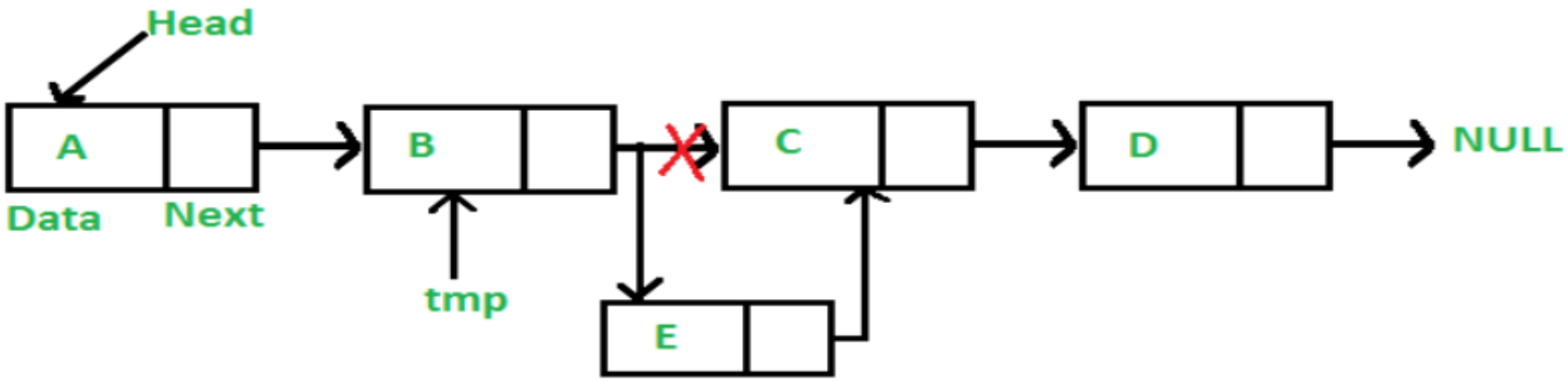
- To iterate:
  - Make a temporary ListNode* to hold the current node we're on
  - Just repeatedly call temp = temp->next such that we get to the next node
- To remove:
  - Go up to the previous node, store its next (your target node) then set the prev's next equal to the next's next; free your (stored) target.
- To insert:
  - Go up to previous node, set new node's next to prev's next, set prev's next equal to new node.

^^^SEARCH ALL ALGORITHMS ONLINE,

Linked List Visuals: Remove



Linked List Visuals: Insert



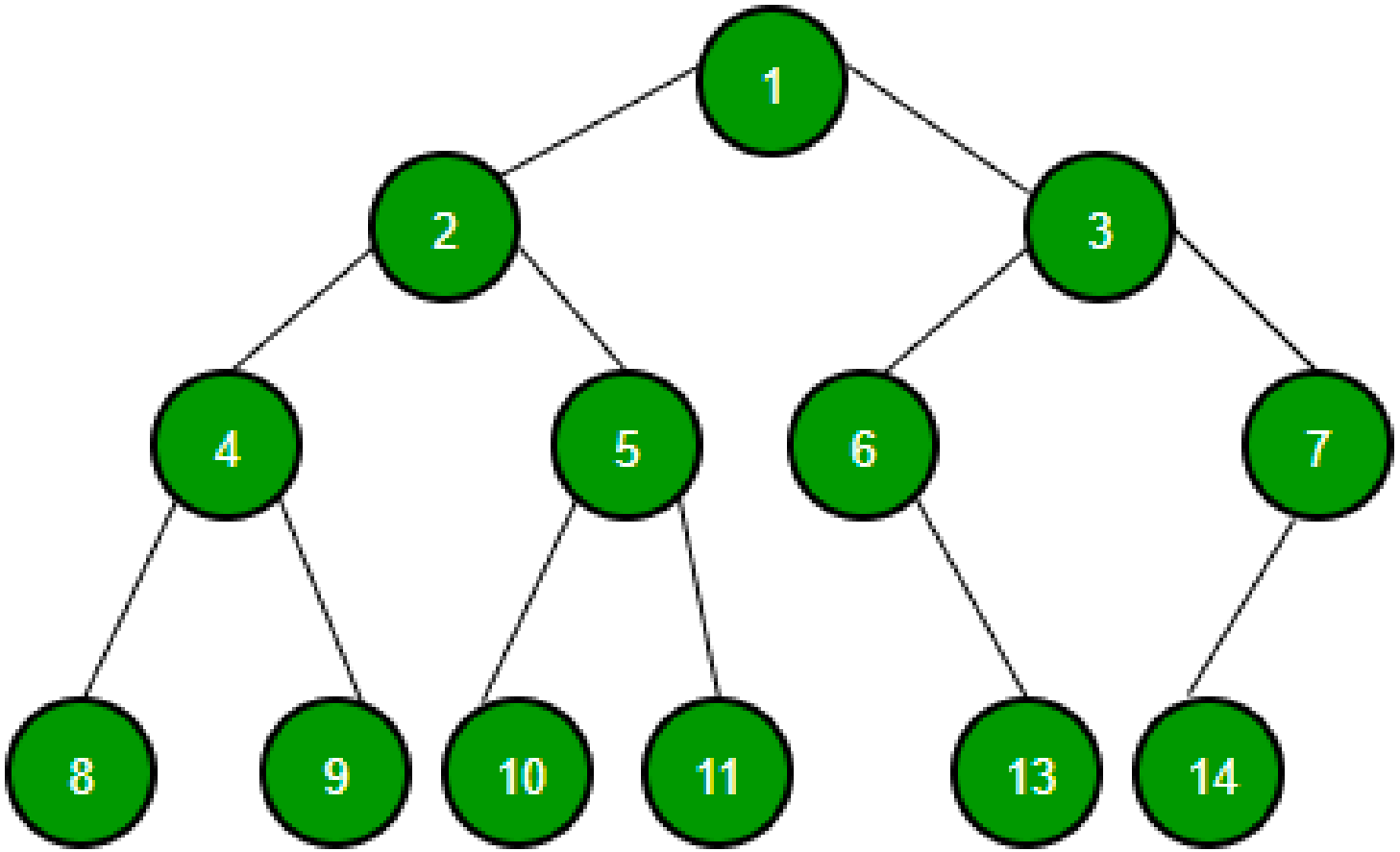
What's Wrong with this Linked List?

```
typedef struct ListNode {
     ListNode next;
     int data;
} ListNode;

ListNode add_to_list (ListNode next, int data_in) {
     ListNode new_node;
     new_node.next = next;
     new_node.data = data_in;
     return new_node;
}
```

What's Wrong with this Linked List?

```
typedef struct ListNode {
      ListNode next;
      int data;
} ListNode;

ListNode add_to_list (ListNode next, int data_in) {
      ListNode new_node;
      new_node.next = next;
      new_node.data = data_in;
      return new_node;
}
```

TREES!



C++

- Best way to make sense is as a change in programming philosophy
- From C-based open structure to object oriented programming
- Think of programs as "objects" interacting and doing things
- Object Oriented Programming has 4 pillars:
  - Encapsulation
  - Abstraction
  - Inheritance
  - Polymorphism

Classes, Objects (Encapsulation, and Abstraction)

- Encapsulation: My class is an organized box, from which I give you just what's necessary, nothing more or less (organization tool)
  - Why we make private members and helper functions, and public functions
- Abstraction: If someone hands me something and says it works, I have faith it works and don't care how (Think JSR from LC3)
- Class has members (data, like in structs!) and functions that use/act upon members
- An object is an instantiation of a class (Analogy: Think of class as a blueprint and object as a house)
- Encapsulation means I keep direct access to members private and the functions you need public

Basic Class Structure

```
1   class exClass{ //example Class
2   public: //public modifer
3     exClass(); //constructor
4     exClass(int i); //another constructor
5     ~exClass(); //destructor
6     void setMem(int i); //function
7     int getMem(); //fucntion
8   private: //private modifier
9     int member; //example member
10  };
```

- We keep members private because of the idea of Encapsulation!
  - I give you access to just what you need
  - Give you no access to the "guts" of my class, controlled access
  - Also the basic reason why we organize into .h and .cpp files
  - (You only see .h and what everything in it does)
  - You can only get/set/modify on my rules, and because of **ABSTRACTION** you have faith that I implemented correctly

Constructors and Destructors

- So if I give you nothing to set with, how do I initially set the members of an object?
- CONSTRUCTORS!
- Constructors have no return type, but are defined like any other function (pass arguments, set members depending on how we define)
- If you don't make a constructor, C++ gives your class a default constructor that sets all members to default and all pointers to NULL
- DESTRUCTOR is how we free members put on the heap (usually pointers)

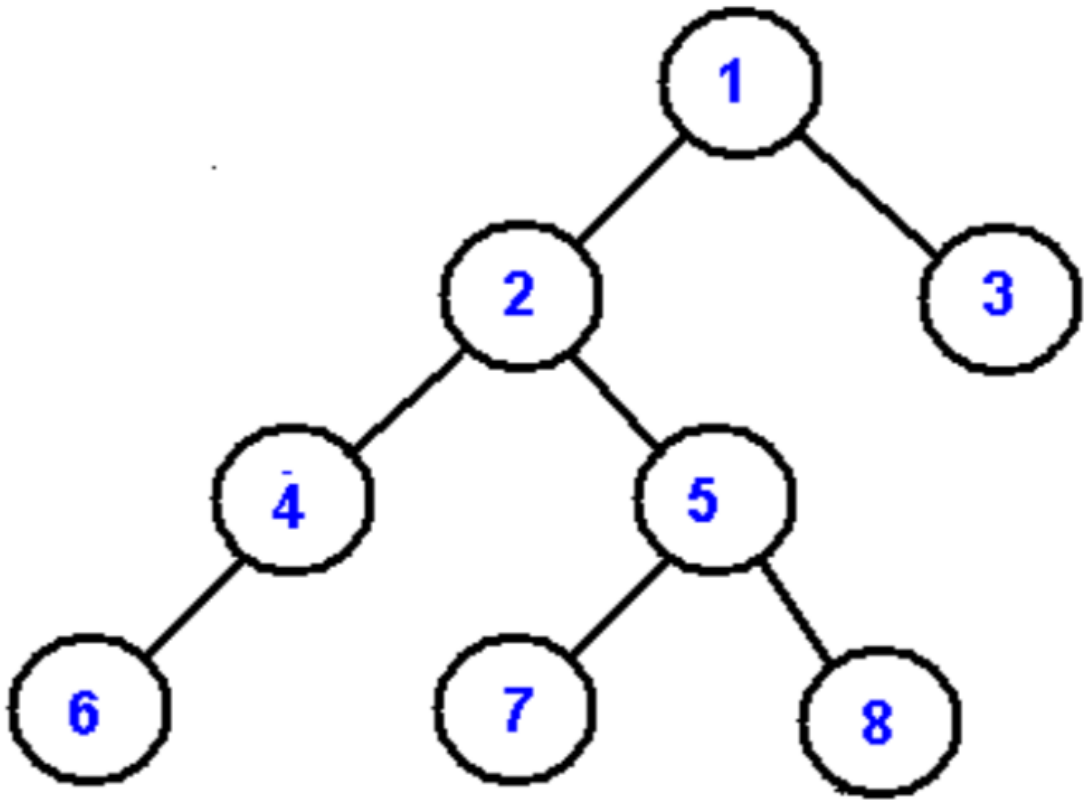
Operator Overloading

- In C++, classes can redefine operators! (+, -, /, =, ==)
- Example:

```
void operator+(complex c, complex d)
    {
        complex temp;
        temp.real = c.real + d.real;
        temp.imaginary = c.imaginary + d.imaginary;
        cout<<"Real sum is      : "<<temp.real<<endl;
        cout<<"Imaginary sum is : "<<temp.imaginary<<en
    }
```

- Basically like any function BUT called directly

Q4. Is the following tree a binary search tree? Explain your reason.



Concept Question 1

```
char word[10];
char *cptr;
cptr = word; //assign cptr to point to word
```

- How does C pass arrays?
- In LC-3, how many bytes of memory are needed to store an integer pointer (int *int_ptr)?

Concept Question 2

What is printed by this program?

```
#include <stdint.h>
#include <stdio.h>
static char letters[6] = {'A', 'E', 'F', 'D', 'B', 'C'};

void mystery () {
        static int32_t X = 5;
        static int32_t Y;
        Y = 2;
        printf ("%c%c", letters[--Y], letters[X--]);
}

int main () {
        mystery ();
        mystery ();
        return 0;
}
```

Concept Question 2

What is printed by this program?

```
#include <stdint.h>
#include <stdio.h>
static char letters[6] = {'A', 'E', 'F', 'D', 'B', 'C'};

void mystery () {
        static int32_t X = 5;
        static int32_t Y;
        Y = 2;
        printf ("%c%c", letters[--Y], letters[X--]);
}

int main () {
        mystery ();
        mystery ();
        return 0;
}
```

ECEB

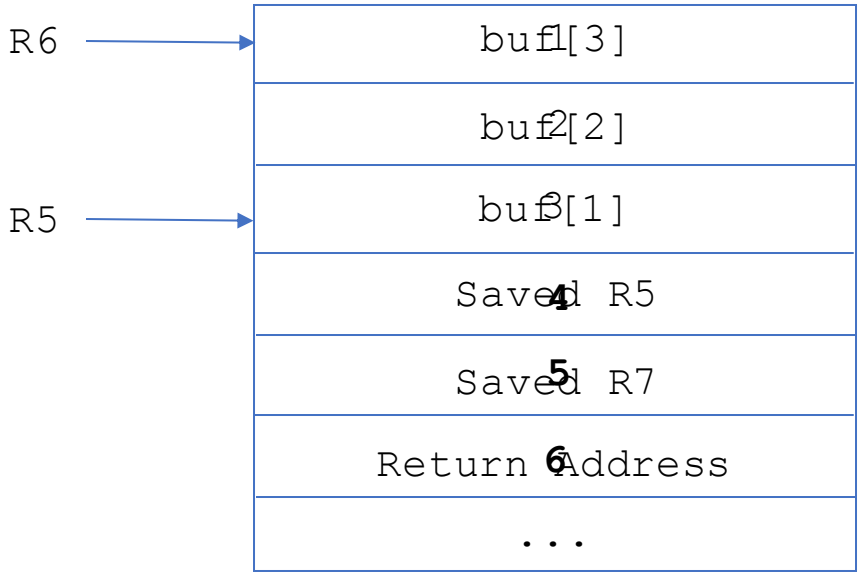
Concept Question 3 (Challenge)

```
int foo () {
    char buf[3];
    scanf("%s", &buf);
    return 0;
}
```

What happens if we pass “123456” into this program?

Concept Question 3 (Challenge)

```
int foo () {
     char buf[3];
     scanf("%s", &buf);
     return 0;
}
```



What happens if we pass “123456” into this program?

Concept Question 4

What is the output of the program?
What is wrong with the function ReverseArray?

```
#include <stdio.h>

void ReverseArray(int array[], int
size) {
	int start = 0, end = size - 1,
temp;
	if (start < end) {
		temp = array[start];
		array[start] = array[end];
		array[end] = temp;
		ReverseArray(array, size-
1);
	}
}
```

```
int main(){
	int array[5], i;
	for (i = 0; i<5; i++){
		array[i] = i;
	}

	ReverseArray(array, 5);
	printf("Reversed Array: ");
	for (i = 0; i<5; i++){
		printf("%d ", array[i]);
	}
	printf("\n");
	return 0;
}
```

Concept Question 4

What is the output of the program?
What is wrong with the function ReverseArray?

```
#include <stdio.h>

void ReverseArray(int array[], int
size) {
	int start = 0, end = size - 1,
temp;
	if (start < end) {
		temp = array[start];
		array[start] = array[end];
		array[end] = temp;
		ReverseArray(array, size-
1);
	}
}
```

0 1 2 3 4
4 1 2 3 0
3 1 2 4 0
2 1 3 4 0
1 2 3 4 0

Concept Question 4

What is the output of the program?
What is wrong with the function ReverseArray?

```
#include <stdio.h>
void ReverseArray(int array[], int
size) {
	int start = 0, end = size - 1,
temp;
	if (start < end) {
		temp = array[start];
		array[start] = array[end];
		array[end] = temp;
		ReverseArray(array+1,
size-2);
	}
}
```

Concept Question 4

What is the output of the program?
What is wrong with the function ReverseArray?

```
#include <stdio.h>
void ReverseArray(int array[], int
size) {
	int start = 0, end = size - 1,
temp;
	if (start < end) {
		temp = array[start];
		array[start] = array[end];
		array[end] = temp;
		ReverseArray(array+1,
size-2);
	}
}
```

0 1 2 3 4 (size = 5)
4 1 2 3 0 (size = 3)
4 3 2 1 0 (size = 1)

Concept Question 5

What is wrong with this recursive function?

```
int find_midpoint(int a, int b) {
    if (a == b) { return a; }
    else { return find_midpoint(a+1,b-1); }
}
```

Concept Question 5

What is wrong with this recursive function?

```
find_midpoint(0, 6)
find_midpoint(1, 5)
find_midpoint(2, 4)
find_midpoint(3, 3)
Return 3
```

```
int find_midpoint(int a, int b) {
    if (a == b) { return a; }
    else { return find_midpoint(a+1,b-1); }
}
```

Concept Question 5

What is wrong with this recursive function?

```
int find_midpoint(int a, int b) {
    if (a == b) { return a; }
    else { return find_midpoint(a+1,b-1); }
}
```

```
find_midpoint(0, 7)
find_midpoint(1, 6)
find_midpoint(2, 5)
find_midpoint(3, 4)
find_midpoint(4, 3)
find_midpoint(5, 2)
find_midpoint(6, 1)
find_midpoint(7, 0)
find_midpoint(8, -1)
find_midpoint(9, -2)
...
```

Concept Question 6

- Which of the following LC3 assembly sections will depend on the number of parameters passed to the function `example`?

In Caller:

```
; Section 1: Prepare for call

JSR EXAMPLE

; Section 2: Cleanup after call
```

In Callee:

```
EXAMPLE
; Section 3: Setup Stack Frame

; (Execute code)

; Section 4: Teardown stack

RET
```

Concept Question 6

- Which of the following LC3 assembly sections will depend on the number of parameters passed to the function `example`?

In Caller:

```
; Section 1: Prepare for call

JSR EXAMPLE

; Section 2: Cleanup after call
```

In Callee:

```
EXAMPLE
; Section 3: Setup Stack Frame

; (Execute code)

; Section 4: Teardown stack

RET
```

Concept Question 7

```
; Takes two positive integers and returns the product
MULT  AND R1, R1, #0
      ADD R3, R3, #0
      BRz DONE
LOOP  ADD R1, R1, R2
      ADD R3, R3, #-1
      BRp LOOP
DONE  RET
```

Which registers are caller saved?

Which registers are callee saved?

Concept Question 7

```
; Takes two positive integers and returns the product
MULT  AND R1, R1, #0
      ADD R3, R3, #0
      BRz DONE
LOOP  ADD R1, R1, R2
      ADD R3, R3, #-1
      BRp LOOP
DONE  RET
```

Which registers are caller saved? R1, R3, R7

Which registers are callee saved? R0, R2, R4, R5, R6

Concept Question 8

What is wrong with this program?

```
int array[4][2];
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 4; j++) {
        array[j][i] = i+j;
    }
}
```

Concept Question 8

What is wrong with this program?

```
int array[4][2];
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 4; j++) {
        array[i][j] = i+j;
    }
}
```

Concept Question 8

Fill in the blank such that array contains the same memory as it did on the previous slide

```
int array[8];
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 4; j++) {
        ____________________ = i+j;
    }
}
```

Concept Question 8

Fill in the blank such that array contains the same memory as it did on the previous slide

```
int array[8];
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 4; j++) {
        array[(4*i)+j] = i+j;
    }
}
```

Concept Question 9

Fill in the blanks, assuming `fname` is a file containing an integer on the first line

```
int read_int_from_file (const char* fname) {
      int x, status;
      FILE* f = ___________________;
      if (f == NULL) { return -1; }
      status = ____________________;
      _________;
      if (______ != 0) { return x; }
      else { return -1; }
}
```

Concept Question 9

Fill in the blanks, assuming `fname` is a file containing an integer on the first line

```
int read_int_from_file (const char* fname) {
      int x, status;
      FILE* f = fopen(fname, "r");
      if (f == NULL) { return -1; }
      status = fscanf(f, "%d", &x);
      fclose(f);
      if (status != 0) { return x; }
      else { return -1; }
}
```

Concept Question 10

Fill in the blanks to find the student with the highest GPA and store a pointer to them in `best_student`

```
typedef struct StudentStruct {
        int UIN;
        float GPA;
} Student;

int main () {
        Student all_students[5];
        // Load data into all students:
        load_students(all_students, 5);

        // Find the student with the highest GPA:
        Student* best_student = ________________;
        find_best(all_students, 5, ____________);
        printf("Best GPA:%f\n", _________________);
}
```

```
void find_best(Student* all, int
num_students, Student** best) {

    for (int i = 0; i < num_students; i++)
    {
        if (all[i].GPA > _____________) {
            ___________________;
        }
    }
}
```

Concept Question 10

Fill in the blanks to find the student with the highest GPA and store a pointer to them in `best_student`

```
typedef struct StudentStruct {
        int UIN;
        float GPA;
} Student;

int main () {
        Student all_students[5];
        // Load data into all students:
        load_students(all_students, 5);

        // Find the student with the highest GPA:
        Student* best_student = &(all_students[0]);
        find_best(all_students, 5, &best_student);
        printf("Best GPA:%f\n", best_student->GPA);
}
```

```
void find_best(Student* all, int
num_students, Student** best) {

    for (int i = 0; i < num_students; i++)
    {
        if (all[i].GPA > (*best)->GPA) {
            *best = &(all[i]);
        }
    }
}
```

Concept Question 10

Fill in the blanks to find the student with the highest GPA and store a pointer to them in `best_student`

```
typedef struct StudentStruct {
        int UIN;
        float GPA;
} Student;

int main () {
        Student all_students[5];
        // Load data into all students:
        load_students(all_students, 5);

        // Find the student with the highest GPA:
        Student* best_student = &(all_students[0]);
        find_best(all_students, 5, &best_student);
        printf("Best GPA:%f\n", best_student->GPA);
}
```

```
void find_best(Student* all, int
num_students, Student** best) {

    for (int i = 0; i < num_students; i++)
    {
        if (all[i].GPA > (*best)->GPA) {
            *best = &(all[i]);
        }
    }
}
```

Critical Thinking: Why do we need this line? What if we simply said `best_student = NULL`?

Concept Question 11

How many times will the function `recursive_func` be called?

```
int main () {
     recursive_func(5);
}

void recursive_func (unsigned int a) {
     printf(”a is %d! \n”, a);
     if (a < 0) { return; }
     recursive_func(a – 1);
}
```

Concept Question 11

How many times will the function `recursive_func` be called?

```
int main () {
     recursive_func(5);
}

void recursive_func (unsigned int a) {
     printf("a is %d! \n", a);
     if (a < 0) { return; }
     recursive_func(a - 1);
}
```

Concept Question 11

How many times will the function `recursive_func` be called?

```
int main () {
     recursive_func(5);
}

void recursive_func (unsigned int a) {
     printf("a is %d! \n", a);
     if (a < 0) { return; }
     recursive_func(a - 1);
}
```

`a` is unsigned, so it will never be less than 0!

Concept Question 11

How many times will the function `recursive_func` be called?

```
int main () {
     recursive_func(5);
}

void recursive_func (unsigned int a) {
     printf("a is %d! \n", a);
     if (a < 0) { return; }
     recursive_func(a - 1);
}
```

`a` is unsigned, so it will never be less than 0!

The recursion will only end when we run out of memory!

Concept Question Codes

- All the code has been posted online at this link:
- https://pastebin.com/pyzMNmb9

TIPS

- Make sure you understand all MPs.
- Attempt the coding portion before the concept portion.
- Atleast one question should be based off a lecture example or lab.
- Check your pointers!