ECE220 Final Review - Cramming Carnival

Author: Members of HKN

1 Logical Little Computer 3 Language Lessons

(a) **I/O**: What is the difference between Polling I/O versus Interrupt I/O?

(b) **Stack**: When using a stack, when you pop an item off the stack, is it removed from memory?

(c) **Stack Pt. 2**: If the input stream to a Stack is 123456789, design a set of pushes and Pops such that it is outputted 235641897?

(d) **Subroutines**: What is the purpose of using subroutines in LC3?

(e) **Multiplication**: Write a Multiplication Subroutine in LC3 assembly language.

```
;input R3, R4
;out R0
MULTIPLY
;your code goes here:
```

(f) **More subroutines**: What extra step do you need to take when executing a subroutine inside a subroutine? (Hint: What gets overwritten when JSR is called?)

2 Captivating C Coding

(a) **Variable lifespan**: Explain the "lifespan" of a local variable during a C function call.

(b) **Poopy Pointers**: Explain the importance of having the parameters of this function being pointers.

```
1 void swap(int* a, int* b)
2 {
3     int temp;
4     temp = *a;
5     *a = *b;
6     *b = temp;
7 }
```

(c) **Lovely Linked Lists**: What is the benefit of using a linked list over an array to represent a large, ever-changing list?

(d) **Rambunctious Recursion**: When writing a recursive algorithm, what is the goal of each recursive step? (Hint: The base case represents the simplest form of the problem)

(e) **Arrays**: How are arrays passed into functions in C?

(f) **Static Silliness**: What is printed by the following program?

```
1  static char letters[6] = {'A', 'E', 'F', 'D', 'B', 'C'};
2
3  void mystery () {
4      static int X = 5;
5      int Y = 2;
6      printf ("%c%c", letters[--Y], letters[X--]);
7  }
8  int main() {
9      mystery();
10     mystery();
11     return 0;
12 }
13
```

(g) **Midpoint**: What is wrong with this recursive midpoint function?

```
1 find_midpoint(int a, int b) {
2     if (a == b) { return a; }
3     else if (a < b){
4        return find_midpoint (a+1, b-1);
5     }
6     else if (a > b){
7        return find_midpoint( a-1, b+1);
8     }
9 }
```

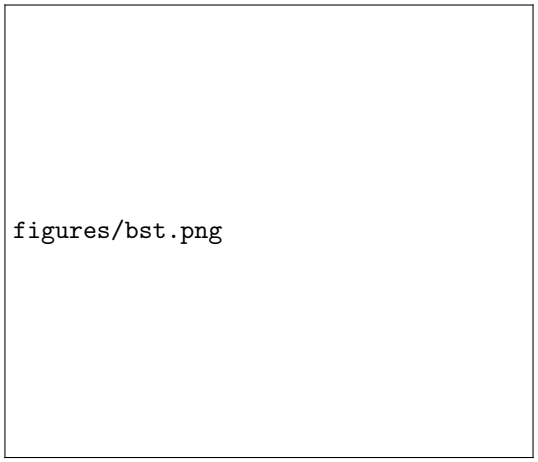
(h) **Function Foo**: What is this function accomplishing? Assume it is called with a valid head to a singly linked list.

```
struct node{
    int num;
    struct node* next;
}
int foo(struct node* head, int bar) {
    if (head->num >= bar) {
        bar = head->num;
    }
    if (head->next == null) {
        return bar;
    }
    else {
        return foo(head->next, bar);
    }
}
```

(i) Which sorting algorithm is this?

```
void sort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j + 1]);
}
```

(j) **Binary Search Trees**: Are these binary search trees?



(k) **C to LC3**: Convert the following C function into LC3 using Callee Setup and Teardown.

```
1 int foo(int a, int b) {
2     int x;
3     x = a + b;
4     return x;
5 }
```

(l) **Recursive Reversal**: What is the output of this program? If there is an error in ReverseArray, identify the line and fix it? (Hint: it might be nice and helpful to print every step of Reverse Array)

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

    if (start < end) {
        // Swap First and Last
        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;
}
```

(m) What is wrong with the following code?

```
double * firstlast(const double values[], int size)
{
    double result[2];
    result[0] = value[0];
    result[1] = value[size-1];
    return result;
}
```

(n) **Alternative Indexing**: Fill in the blanks to make the two arrays the same. Assume row-major order

for array2.

```
int array1[4][2];
int array2[8];
int i, j;
for (i = 0; i < 2; i++) {
    for(j = 0; j < 4; j++) {
    array1[  ][  ] = i + j;
    array2[     ] = i + j;
    }
}
```

(o) 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 >         ) {
            // Fill:
        }
    }
}
```

(p) In MP10, we tasked you with inserting a node into a sorted singly-linked list. To help you get accustomed to poopy pointer arithmetic, we will now attempt to do the same, but with a doubly-linked list. You are given the following struct doubly_node and the function insert_doubly_node(), which takes in the head of a doubly linked list (but you will not have a tail) and a value to insert. Complete the function skeleton below. Assume that the double-pointer head is never NULL, but that the single

pointer *head (the result of dereferencing head once) is not necessarily never NULL.

```
typedef struct doubly_node {
    int value
    doubly_node * next;
    doubly_node * prev;
} doubly_node;


void print_list_d(doubly_node * head)
{
    doubly_node * cur = head;
    while (cur != NULL)
    {
        printf("%d -> ", cur->value);
        cur = cur->next;
    }
}

void insert_doubly_node(doubly_node ** head, int value)
{
    /* edge case 0: linked list is completely empty */
    if(*head ==         )
    {
        *head = (             ) malloc(sizeof(          ));
        (*head)->next =         ;
        (*head)->prev =         ;
        (*head)->value =         ;
        return;
    }

    /* edge case 1: insert AT the head*/
    if (value < (*head)->      )
    {
        doubly_node * new_node = (          ) malloc(          );
        new_node->        = value;
        new_node->next =            ; //new node should be inserted before head
        new_node->          = NULL; //new node previous does not point to anything
        (*head)->        =  new_node; //head's previous points to new_node
        *head =          ; //update the new head of the linked list
        return;
    }

    doubly_node * cur = *head;
    while (cur !=      )
    {
        if (cur->value < value)
        {
            /* last edge case: insert AFTER the tail of the linked list */
            if (cur->next == NULL)
            {
                doubly_node * new_node =                                    ;
                //allocated new_node on the heap

                new_node->value = value;
                new_node->next =            ; //new_node is the tail of the list
                new_node->      = cur; //new_node's previous points to the current
                cur->next =      ; //cur's next is the new node
                return;
            }
        cur =           // advance to the next node
        }


        else
        {
            doubly_node * new_node = (        ) malloc(sizeof(        ));
```

```
                new_node->value =            ;
                new_node->next =             ;
                new_node->         = cur->prev;
                cur->prev->next =        ;
                cur->prev = new_node;
                return;
            }
        }
    }
```

Assuming that insert_doubly_node() and print_list_d() are implemented correctly, what will the output of this code snippet be?

```
int main()
{
    doubly_node * head = NULL;
    insert_doubly_node(&head, 5);
    insert_doubly_node(&head, 2);
    insert_doubly_node(&head, 7)
    insert_doubly_node(&head, 9);
    insert_doubly_node(&head, 1);
    insert_doubly_node(&head, 10);
    insert_doubly_node(&head, 5);
    print_list_d(head);

    return 0;
}
```

3 C++ Programming

(a) **Class**: What is the difference between structs in C and classes in C++?

(b) **Constructor**: What does a constructor do in a class? What does a destructor do in a class? What might happen if we do not have a correct destructor?

(c) **Access specifier**: What is the difference between public, private, and protected in a C++ class?

(d) **Operator overloading**: What does operator overloading do in a C++ class? Give four different examples of operators that can be overloaded.

(e) **Rectangle**: Write the constructors and area function for this rectangle class.

```
1 class Rectangle {
2 private:
3     int width, height;
4 public:
5     Rectangle();
```

```
    Rectangle(int w, int h);
    int const area();
}
```

(f) **Iterator**: What will be printed if we run the following C++ code? (assume we import all needed files)

```
    int main()
{
    vector<int> ar = { 1, 2, 3, 4, 5 };
    vector<int>::iterator ptr = ar.begin();
    ptr++;
    cout << *ptr << endl;
    ptr = ptr + 3;
    cout << *ptr << endl;
    return 0;
}
```

(g) **Template**: What will happen if we run the main function?

```
template <typename T>
T max(T x, T y)
{
    return (x > y)? x : y;
}
int main()
{
    cout << max(3, 7) << endl;
    cout << max(3.0, 7.0) << endl;
    cout << max(3, 7.0) << endl;
    return 0;
}
```

(h) **Copy**: What is the difference between shallow copy and deep copy?

(i) What type of traversal is this?

```
void traversal (node *root) {
    if ( root != NULL) {
       std::cout << root -> data << std::endl;
       traversal (root -> left);
       traversal (root -> right);
    }
 }
```

(j) What is the pre-order, in-order, post-order and level order traversal for the following binary search tree?

figures/bst2.png

(k) How many bytes of memory have been allocated on the heap? Assume that we are in a 32-bit system (recall that an int is 4 bytes, a double is 8 bytes, an int * is 4 bytes, and a char is 1 byte)

```

    int main () {
        int a = 4;
        int d = 6;
        int c = 10;
        int ** b = new int * [5];
        b[0] = &a;
        b[1] = &c;
        b[2] = &d;
        b[3] = new int[6];
```

```
        b[3][0] = 5;
        b[3][2] = 8;
        b[3][1] = 17;
        b[3][5] = 2;
        b[3][4] = 9;
        b[3][3] = 22;
        b[4] = new int(13);
        char * character = new char (k);
}
```

Write the contents of what each pointer points to (if it points to an array, write the contents of the array. If it points to a single int/char, write what the int/char is.)

Fill in the lines after line 10 needed to properly free the memory. Not all lines may be needed.

```
int main () {
    int a = 4;
    int d = 6;
    int c = 10;
    int ** b = new int * [5];
    b[0] = &a;
    b[1] = &c;
    b[2] = &d;
    b[3] = new int[6];
    b[3][0] = 5;
    b[3][2] = 8;
    b[3][1] = 17;
    b[3][5] = 2;
    b[3][4] = 9;
    b[3][3] = 22;
    b[4] = new int(13);
    char * character = new char('k');
    --------
    --------
    --------
    --------
    --------
    --------
}
```

Let's say we have a void pointer p. What is the difference between delete [] p and delete p?

(l) Linked List Problem: Your best friend Gabriel has a C program that stores the name of his favorite Fortune 500 company. He is storing the company name in a linked list format. The structure of the

linked list and the nodes are located below. The end of the list will have the company _node * next point to NULL;

```
typedef struct company_node
{
    int index;
    char letter;
    struct company_node * next;

} company_node;

typedef struct company
{
    int n; //number of entries in list
    company_node * company_head;

} company;
```

Given the above structure, and the fact that the company_node * pointer is 8 bytes (64-bit system), **what is the size in bytes of both struct company and struct company_node?**

Gabriel wants to tell you what his favorite Fortune 500 company is but is unable to speak because he just got his wisdom teeth removed. **Fill in the below program to print the name of the company.** This program takes place in main and has already imported the relevant libraries.

Assume everything has been initialized/malloc'd correctly and the name of the struct was declared with the following line.

Part 1: Printing

```
company * gabriel_company;

printf("The Name of Gabriel's Favorite Fortune 500 Company: ");

company_ * current = gabriel_company->company_head;
for (int i = 0; i < _____; i++) //iteraite through the list
{
    printf("%c", _____); //print the current character
    _____ //move the current node
}
```

Gabriel is quite shy and doesn't want anyone to know what his favorite Fortune 500 company is (except for you, his best friend). **Help him finish the below code to remove the company name from the heap.**

Part 2: Freeing

```
company_node * the_current = gabriel_company->company_head;
company_node the_previous = gabriel_company->company_head;
while(the_current->next != _____){
    the_current = the_current->next;
    free(_____);
    _____; //move previous further
}

free(gabriel_company);
```

Whenever the above code is executed, a memory leak occurs. /textbfExplain what a memory leak is and come up with one line of code that can be added above to fix the memory leak.

(m) Virtual Keywords:

```
#include <iostream>
class Base
{
    public:
        virtual void f()
        {
            std::cout << "base\n";
        }
};

class Derived_One : public Base
{
    public:
        void f() override
        {
            std::cout << "derived_1\n";
        }
};

class Derived_Two : public Base
{
    public:
        void f()
        {
            std::cout << "derived_2\n";
        }
};


int main()
{
    //Define the objects
    Base b;
    Derived_One d_1;
    Derived_Two d_2;

    //assign the objects by reference
    Base& br = b;
    Base& dr_1 = d_1;
    Base& dr_2 = d_2;

    /* PART 1 */
    br.f();
    dr_1.f();
    dr_2.f();

    //Assign the objects by pointer
    Base * bp = &b;
    Base * dp_1 = &d_1;
    Base * dp_2 = &d_2;

```

```
52              /* Part 2 */
53              bp->f();
54              dp_1->f();
55              dp_2->f();
56
57              /* Part 3 */
58              br.Base::f();
59              dr_1.Base::f();
60              dr_2.Base::f();
61
62              return 0
63          }
64
```

Part 1: What is printed by br, dr_1, and dr_2?

Part 2: What is printed by br, dr_1, and dr_2? What is the difference between Part 1 and Part 2?

Part 3: What is printed br, dr_1, and dr_2? What is the difference between the other two parts?