ECE220 Final Review - Cramming Carnival Solutions

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?
Polling I/O is when the program continually checks on the status of the input device (i.e. if a button has been pressed on the keyboard). Interrupt I/O is where the I/O device controls the flow, the device generates an "interrupt signal" which indicates that the device is ready with a new operation. The program gets "stuck waiting" in Polling I/O, but in Interrupt I/O the program continues until the interrupt signal is received.

(b) **Stack**: When using a stack, when you pop an item off the stack, is it removed from memory?
When you pop an item off a stack, it is not actually removed from memory, instead the stack pointer is moved "down" to point to the next item on the stack.

(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?

Operation	Stack After Operation
push	1
push	21
pop	1
push	31
pop	1
push	41
push	541
pop	41
push	641
pop	41
pop	1
pop	empty
push	7
push	87
pop	7
push	97
pop	7
pop	empty
done	

Table 1: Stack state after each operation

(d) **Subroutines**: What is the purpose of using subroutines in LC3?
Subroutines allow you to break the code down into usable and resusable chunks that will be used multiple times so that you do not have to rewrite the same block of code each time. Example is the addition or multiplication subroutine for the stack calculator.

(e) **Multiplication**: Write a Multiplication Subroutine in LC3 assembly language.
;input R3, R4

```
;out R0
MULTIPLY
;your code goes here:

ST R4, SaveR4        ; Callee-Save R4
AND R0, R0, #0       ; Clear R0
ADD R4, R4, #0       ; Check R4's condition code
BRz DONE             ; if R4 is 0, return 0
BRn NEG

POS                  ; R4 is positive
ADD R0, R0, R3       ; Add R3 to R0
ADD R4, R4, #-1      ; Decrement R4
BRp POS              ; Complete R4 number of loops
BRnzp DONE           ; R0 has R4*R3

NEG                  ; R4 is negative
ADD R0, R0, R3       ; Add R3 to R0
ADD R4, R4, #1       ; Increment R4 (since R4 is negative)
BRn NEG              ; Loop abs(R4) times
NOT R0, R0           ;
ADD R0, R0, #1       ; Negate R0 (Since R0 has -R4*R3)

DONE
LD R4, SaveR4        ; Callee-Restore R4
RET
SaveR4 .BLKW #1      ; Save this spot for R4
```

(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?)
Save the value of R7 in memory so that you know which address to go back to in the original subroutine.

2 Captivating C Coding

(a) **Variable lifespan**: Explain the "lifespan" of a local variable during a C function call.
When a function gets called, the local variables are placed onto the runtime stack and are then used during the execution of the function. When the function returns control back to the main function the runtime stack pointer goes below the local variables so they are removed from the stack.

(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 }
```

Having parameters as pointers allow the data pointed to to actually change in the function. If the values were passed in as just their values, only the run time stack variables would be edited so that the operation would not be saved. Note how this function doesn't return anything.

(c) **Lovely Linked Lists**: What is the benefit of using a linked list over an array to represent a large, ever-changing list?
It is significantly easier to both add and remove items in a linked list. Simply moving around some pointers makes it so that you can change the items easy.

(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)
The goal of each recursive step is to make the problem slightly simpler. An example is factorial, you calculate number * factorial(number-1). By decrementing the number each time, we make the problem easier, until we get to the base case.

(e) **Arrays**: How are arrays passed into functions in C?
Arrays are passed as a pointer to the first item in the array.

(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
```

"ECEB" is printed; On the first call to mystery(), Y is set to 2 and X is statically set to 5. The notation "–Y" means to decrement, then use the result. "X–" means the opposite. So the first call to mystery() will print letters[1], letters[5], or "EC". Then the next call Y will be set back to 2. So it will print letters[1], letters[4], or "EB". Notice that X does not get set to 5 because it is declared as static, and does not get put on the stack, instead it is put in a different section of memory.

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

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

There are two main things wrong. The first, is that were is no return type for this function, we need to make this return a float. The second, is that we need to deal with numbers that are odd-length apart, for example 4 and 7. To do this we can add: else if (a+1 == b) return ((a+b)/2); as another condition to check.

(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);
    }
}
```

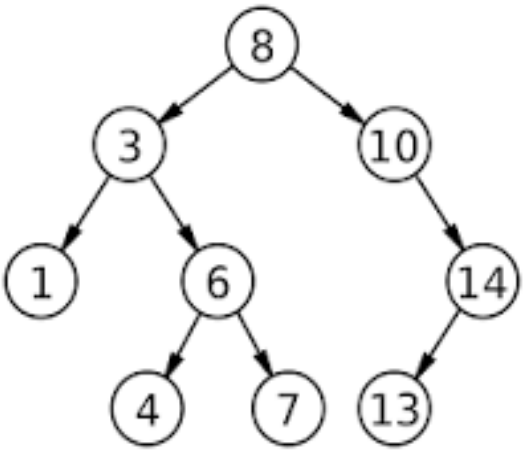
This function recursively returns the largest number of the linked list.

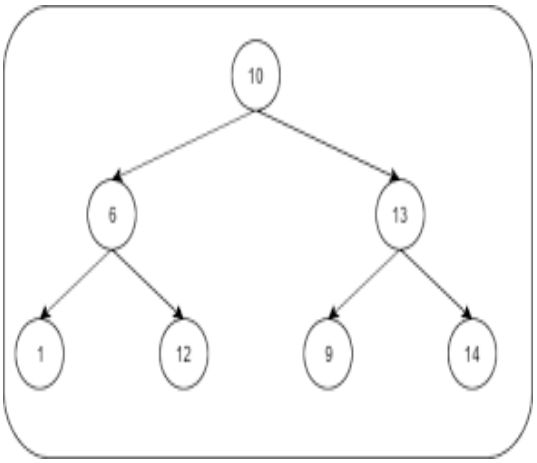
(i) Which sorting algorithm is this?

```
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]);
}
```

This is bubble sort.

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





Yes, No

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

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

```
ADD R6, R6, #-2          ; Allocate spot for return value,now R6 points to return address
STR R7, R6, #0                                                    ; Push R7 (Return Address)
ADD R6, R6, #-1
STR R5, R6, #0                                          ; Store R5 (Caller's Frame Pointer)
ADD R5, R6, #-1                                                 ;Set frame pointer for Callee

ADD R6, R6, #-1                                      ;Allocate memory for local variables (x)
LDR R1, R5, #4                                                                    ; Load a
LDR R2, R5, #5                                                                    ; Load b
ADD R0, R1, R2
STR R0, R5, #0                                                      ; Store result into x
STR R0, R5, #3                                    ; Store result into return value slot

ADD R6, R5, #1                                                    ; Pop local variables
LDR R5, R6, #0                                                    ; Pop Frame Poitner
ADD R6, R6, #1
LDR R7, R6, #0                                                    ; Pop Return Address
ADD R6, R6, #1
RET
```

(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;
}
```

The program is basically just flipping the first and last elements, and then flipping the first and second to last and so on. So in the end, the only thing that actually changes is the last element goes to the front

and the rest is shifted up. To fix this we need to change the recursive call to ReverseArray(array+1, size-2)

(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;
}
```

The return value is a pointer to a local array that will go out of scope when the function ends, leading to an error.

(n) **Alternative Indexing**: Fill in the blanks to make the programs work properly.

```
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;
    }
}
```

array1[i][j]
array2[(4*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:
        }
    }
}
```

&(all_students[0])
&best_student
best_student->GPA
(*best)->GPA
*best = &(all[i])

(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.

```
1   typedef struct doubly_node {
2       int value
3       doubly_node * next;
4       doubly_node * prev;
5   } doubly_node;
6
7
8   void print_list_d(doubly_node * head)
9   {
10      doubly_node * cur = head;
11      while (cur != NULL)
12      {
13          printf("%d -> ", cur->value);
14          cur = cur->next;
15      }
16  }
17
18  void insert_doubly_node(doubly_node ** head, int value)
19  {
20      /* edge case 0: linked list is completely empty */
21      if(*head ==   NULL      )
22      {
23          *head = ( doubly_node *  ) malloc(sizeof( doubly_node  ));
24          (*head)->next =  NULL   ;
25          (*head)->prev =   NULL   ;
26          (*head)->value =  value   ;
27          return;
28      }
29
30      /* edge case 1: insert AT the head*/
31      if (value < (*head)->value      )
32      {
33          doubly_node * new_node = (doubly_node *   ) malloc(  sizeof(doubly_node) );
34          new_node->value = value;
35          new_node->next = *head; //new node should be inserted before head
36          new_node->prev = NULL; //new node previous does not point to anything
37          (*head)->prev  = new_node; //head's previous points to new_node
38          *head = new_node; //update the new head of the linked list
39          return;
40      }
41
42      doubly_node * cur = *head;
43      while (cur != NULL     )
44      {
45          if (cur->value < value)
46          {
47              /* last edge case: insert AFTER the tail of the linked list */
48              if (cur->next == NULL)
49              {
50                  doubly_node * new_node =   (doubly_node *) malloc(sizeof(
    doubly_node));                                  ;
51                  //allocated new_node on the heap
52
53                  new_node->value = value;
54                  new_node->next = NULL; //new_node is the tail of the list
55                  new_node->prev = cur; //new_node's previous points to the current
56                  cur->next = new_node; //cur's next is the new node
57                  return;
58              }
59              cur = cur->next;       // advance to the next node
60          }
61
62
63          else
```

```
64              {
65                  doubly_node * new_node = (doubly_node *  ) malloc(sizeof(doubly_node
        ));
66                  new_node->value = value;
67                  new_node->next = cur;
68                  new_node->prev= cur->prev;
69                  cur->prev->next =  new_node;
70                  cur->prev = new_node;
71                  return;
72              }
73          }
74      }
```

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

```
1       int main()
2       {
3           doubly_node * head = NULL;
4           insert_doubly_node(&head, 5);
5           insert_doubly_node(&head, 2);
6           insert_doubly_node(&head, 7)
7           insert_doubly_node(&head, 9);
8           insert_doubly_node(&head, 1);
9           insert_doubly_node(&head, 10);
10          insert_doubly_node(&head, 5);
11          print_list_d(head);
12
13          return 0;
14      }
```

Answer:

```
1 1 -> 2 -> 5 -> 5 -> 7 -> 9 -> 10 ->
```

3 C++ Programming

(a) **Class**: What is the difference between structs in C and classes in C++?
Both structs in C and classes in C++ have the ability to define the data inside of the class/struct. However, classes in C++ have two extra things. Classes have the ability to control "security" in which you can determine who can access certain parts. Classes also allow you can to define functions that are specific to the class.

(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?
A constructor is a special member function that creates and initializes a new object. A destructor is a special member function that deletes an object and free's all dynamically allocated memory. If destructor is not implement or implemented incorrectly, there might be memory leak since there are memories not being freed

(c) **Access specifier**: What is the difference between public, private, and protected in a C++ class?
Public members can be accessed by anyone who accesses the class. Private members can only be accessed by member functions. Protected members are similar to private as it can only be accessed by member function, but it can be used in member functions of classes derived from the class (Inheritance).

(d) What does operator overloading do in a C++ class? Give four different examples of operators that can be overloaded.
Overloading an operator allows you to redefine the built in operators, including +, -, *, ¡, ¿, =, to allow them to work with your user defined classes

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

```
class Rectangle {
private:
    int width, height;
public:
    Rectangle();
    Rectangle(int w, int h);
    int const area();
}
```

Answer:

```
Rectangle::Rectangle() {
    width = 0;
    height = 0;
}

Rectangle::Rectangle(int w, int h) {
    width = w;
    height = h;
}

int Rectangle::area() const {
    return width * height;
}
```

Why initialize width and height to 0 and not 1?. Also we could use → to reference member variables

(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;
```

```
9       return 0;
10  }
```

2 and 5 will be printed. When initializing the iterator to begin(), it is at the first position. ++ will move the iterator to the second position and printed 2. + 3 will move the iterator to 3 positions later, which is at 5

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

```
1  template <typename T>
2  T max(T x, T y)
3  {
4      return (x > y)? x : y;
5  }
6  int main()
7  {
8      cout << max(3, 7) << endl;
9      cout << max(3.0, 7.0) << endl;
10     cout << max(3, 7.0) << endl;
11     return 0;
12 }
```

Nothing will be printed because an error is caused at line 10. A template can be used so that we can use one function with different types, so the max function works at lines 8 and 9 since the function can both accept int and float. However, at line 10, we pass one int and one float into the function, which are different types. This generates a compiler error because of the ambiguity.

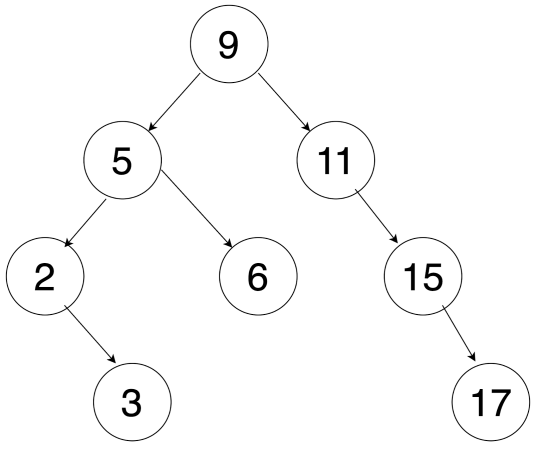
(h) **Copy**: What is the difference between shallow copy and deep copy?
Shallow copy only copies the variables of the original object. If one of the variables is a pointer, it will only copy the pointer but not the data it points to, so they are still pointing to the same data. But deep copy will not only copy the variables but allocate similar memory with the same value as the original object. So the data the pointers point to will also be copied in newly allocated memory

(i) What type of traversal is this?

```
1  void traversal (node *root) {
2      if ( root != NULL) {
3         std::cout << root -> data << std::endl;
4         traversal (root -> left);
5         traversal (root -> right);
6      }
7   }
8
```

This is an example of Pre-Order traversal. This algorithm will recursively iterate through the nodes in the tree, starting at the root and traversing to the left and the right subtrees respectively.

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



Pre-Order: 9, 5, 2, 3, 6, 11, 15, 17
In-Order: 2, 3, 5, 6, 9, 11, 15, 17
Post-Order: 3, 2, 6, 5, 17, 15, 11, 9
Level-Order: 9, 5, 11, 2, 6, 15, 3, 17

(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)

```
1
2     int main () {
3         int a = 4;
4         int d = 6;
5         int c = 10;
6         int ** b = new int * [5];
7         b[0] = &a;
8         b[1] = &c;
9         b[2] = &d;
10        b[3] = new int[6];
11        b[3][0] = 5;
12        b[3][2] = 8;
13        b[3][1] = 17;
14        b[3][5] = 2;
15        b[3][4] = 9;
16        b[3][3] = 22;
17        b[4] = new int(13);
18        char * character = new char (k);
19 }
20
21
22
```

20 + 24 + 4 + 1 = 49 bytes

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.)
4
10
6
5 17 8 22 9 2
13
k

Fill in the lines after line 17 needed to properly free the memory. Not all lines may be needed.

```
1  int main () {
2      int a = 4;
3      int d = 6;
4      int c = 10;
5      int ** b = new int * [5];
6      b[0] = &a;
7      b[1] = &c;
8      b[2] = &d;
9      b[3] = new int[6];
10     b[3][0] = 5;
11     b[3][2] = 8;
12     b[3][1] = 17;
13     b[3][5] = 2;
14     b[3][4] = 9;
15     b[3][3] = 22;
16     b[4] = new int(13);
17     char * character = new char('k');
18     delete [] b[3];
19     delete b[4];
20     delete [] b;
```

```
    delete character;
}
```

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

delete [] will free a contiguous array of memory, while delete frees only a single allocated block of memory of a particular data type. (e.g, an array of ints vs. a single int)

(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?**
company: 4 + 8 = 12 bytes
company_node: 4 + 1 + 8 = 13 bytes

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_node * current = gabriel_company->company_head;
for (int i = 0; i < gabriel_company->n; i++) // iterate through the list
{
    printf("%c", current->letter); //print the current character
    current = current->next; //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 != NULL  ){
    the_current = the_current->next;
    free(the_previous);
    the_previous = the_current ; //move previous further
}

free(gabriel_company);

```

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

A memory leak is when we forget to free allocated memory or lose access to a region of allocated memory due to a pointer value being changed. We can fix this by revising line 8 as follows:

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

```

(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;
```

```
36
37              //assign the objects by reference
38              Base& br = b;
39              Base& dr_1 = d_1;
40              Base& dr_2 = d_2;
41
42              /* PART 1 */
43              br.f();
44              dr_1.f();
45              dr_2.f();
46
47              //Assign the objects by pointer
48              Base * bp = &b;
49              Base * dp_1 = &d_1;
50              Base * dp_2 = &d_2;
51
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?
base
derived_1
derived_2

Part 2: What is printed by br, dr_1, and dr_2? What is the difference between Part 1 and Part 2?
base
derived_1
derived_2

Note: In Part 1, we accessed the functions through objects by reference. In Part 2, we accessed the functions through objects by pointers.

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

Note: In Part 1 and Part 2, we default to the virtual functions despite the objects being a type "Base". With Part 3, we explicitly ask for the parent version of the function through Base::f().