Nick Troccoli
CS 106X

Section #4
Week 5

Section Handout #4: Classes, Pointers, and Dynamic Memory

Based on handouts by various current and past CS106B/X instructors and TAs.

Extra practice problems: CodeStepByStep – pointers, Circle (classes); Textbook – chapter 6 (classes), 11 (pointers)

1. Pointer Trace *(pointers, dynamic memory, tracing)*

The following code produces 5 lines of output. Write each line of output as it would appear on the console. For this problem, assume that the variables in main are stored at the following memory addresses:

- main's a variable is stored at address 0xaa00
- main's b variable is stored at address 0xbb00
- main's c variable is stored at address 0xcc00
- main's d variable is stored at address 0xdd00
- main's e variable is stored at address 0xee00
- any memory dynamically allocated on the heap will be at address 0x555500

```
int pointerTrace(int a, int& b, int* c) {
    b++;
    a += *c;
    cout << b << " " << *c << " " << a << " " << c << endl;
    c = &a;
    return a - b;
}
int main() {
    int a = 4;
    int* b = new int(8);
    int c = -3;
    int d;
    int* e = &a;
    d = pointerTrace(a, *b, &c);
    pointerTrace(c, d, b);
    pointerTrace(*b, *e, &d);
    cout << a << " " << b << " " << *b << " " << c << " " << e << " " << e << " " << *e << endl;
    cout << &a << " " << &b << " " << &c << " " << &d << " " << &e << endl;
    return 0;
}
```

Feel free to tear off the last page of this handout and use one of the Stack/Heap diagrams on that page to help you trace through this code.

Line 1: ____

Line 2: ____

Line 3: ____

Line 4: ____

Line 5: ____

2. Very Complex Pointer Trace *(pointers, dynamic memory, tracing)*

Trace through the execution of a call to `gryffindor()`, as defined below, and draw the program's memory at the designated point in the code. Indicate which variables are on the stack and which are on the heap, and indicate orphaned memory. Indicate with a question mark (?) memory that we don't know the values of.

```
void gryffindor() {
    Hogwarts *triwizard = new Hogwarts[3];
    triwizard[1].wizard = 3;
    triwizard[1].potter = nullptr;
    triwizard[0] = triwizard[1];
    triwizard[2].potter = hufflepuff(triwizard);
    //DRAW THE MEMORY AS IT LOOKS AFTER THE NEXT LINE
    triwizard[2].potter->quaffle = 4;
}

Quidditch * hufflepuff(Hogwarts * cedric) {
    Quidditch *seeker = &(cedric->harry);
    seeker->snitch = new int;
    *(seeker->snitch) = 2;
    cedric = new Hogwarts;
    cedric->harry.quaffle = 6;
    cedric->potter = seeker;
    cedric->potter->quaffle = 8;
    cedric->potter->snitch = &(cedric->potter->bludger[1]);
    seeker->bludger[0] = 4;
    return seeker;
}
```

```
struct Quidditch {
    int quaffle;
    int *snitch;
    int bludger[2];
};

struct Hogwarts {
    int wizard;
    Quidditch harry;
    Quidditch *potter;
};
```

Draw your answer on one of the Stack/Heap diagrams on the last page of this handout.

3. Cleaning Up After Yourself *(pointers, dynamic memory, debugging)*

Whenever you dynamically allocate an array with `new[]`, you need to deallocate it using `delete[]`. It's important when you do so that you only deallocate the array exactly once – deallocating an array zero times causes a memory leak, while deallocating an array multiple times usually causes the program to crash. (Interestingly, deallocating memory twice is called a "double free" and can lead to security vulnerabilities in your code! Take CS155 for details.)

Below are three code snippets. Trace through each snippet and determine whether all memory allocated with `new[]` is correctly deallocated exactly once. If there are any other errors in the program, make sure to report them as well.

```
int main() {
   int* baratheon = new int[3];
   int* targaryen = new int[5];
   baratheon = targaryen;
   targaryen = baratheon;
   delete[] baratheon;
   delete[] targareon;
}
```

```
int main() {
   int* stark = new int[6];
   int* lannister = new int[3];
   delete[] stark;
   stark = lannister;
   delete[] stark;
}
```

```
int main() {
   int* tyrell = new int[137];
   int* arryn = tyrell;
   delete[] tyrell;
   delete[] arryn;
}
```

4. Unit Tests *(testing)*

A major component of writing classes (and code in general) is thinking from the client's side and knowing what functionality you want to support before actually coding it. Writing unit tests prior to writing actual code should help you design your code from the client's perspective and identify potential edgecases early on.

Before writing the `Date` class in problem 5, identify at least four units of functionality that your class should support and write a unit test for each to ensure that your future code will work when called from the client's side. Good candidates for tests include potential edgecases as well as standard functionality. As discussed in lecture and on assignment #4 (in `DNADetectiveTests.cpp`), use the following syntax for your `Date` tests:

```
ADD_TEST("Put your description of the test here") {
    /* body of the test */
    /* add your test code here */
    expect(someCondition); // replace with the functionality you expect to have
}
```

For problems 5 and 6, recall our classes checklist from lecture:

- ❒ **Specify instance variables.** What information is inside this new variable type?
- ❒ **Specify public methods.** What can this variable type do for others?
- ❒ **Specify constructor(s).** How do you create a new variable of this type?

5. First Date *(classes)*

Write a class named Date that stores information about a month and a day. You may ignore leap years and don't store the year in your object. Your class must implement the following public members:

member name	description
`Date(int m, int d)`	Constructor: constructs a new Date representing the given month and day.
`daysInMonth()`	Returns the number of days in the month stored by your date object.
`getDay()`	Returns the day.
`getMonth()`	Returns the month.
`nextDay()`	Advances the Date to the next day, wrapping to the next month and/or year if necessary.
`toString()`	Returns a string representation such as "10/22".

Specifically, write the contents of the .h and .cpp files required to implement this functionality. You are free to add any private member variables or methods that you think are necessary.

```
// .h file
#pragma once
using namespace std;
class Date {
public:
    // you fill this in
private:
    // add whatever you need
};
```

```
// .cpp file
#include "Date.h"

// you fill in the rest
```

See the next page for the bonus part for this problem.

Bonus for Date: add functionality to your Date class to throw errors for invalid month/day input, or to store the year and account for leap years. A year is a leap year if the following three properties are true: (1) the year is divisible by 4, and (2) the year is NOT divisible by 100, unless (3) the year is also divisible by 400. (Interestingly, this is why the year 2000 was a leap year, even though 1900 wasn't and 2100 won't be.)

6. IntArrayList *(classes, pointers, dynamic memory)*

So far in 106X, we've been using the Vector class from what's typically called the "client side"—we call its functions without dealing with how they're actually implemented. Here, we'll turn the tables and think about how it works on the inside, using our knowledge of classes, pointers, and memory allocation all at once.

Write a class `IntArrayList` that implements a growable array of integers. This will be very similar to the basic functionality of a `Vector<int>`. In the same fashion that a Vector works, you should implement this by using an array of `int`s to store the elements that have been added to the collection. Typically the array is larger than the data added so far, so that it has some extra slots in which to put new elements later. We'll use the term **capacity** to describe the actual length of the internal array, and the term **size** to describe the number of elements currently in the list.

Start with an array of length (capacity) 10 by default, and grow it as needed. When growing the internal array, you should double its capacity (why is this a better choice than increasing the capacity by 1?). Do not leak any memory. You may assume that any indices passed in are valid indices (i.e. are in-bounds).

Your class must implement the following public members:

type	member name	description	runtime
-	`IntArrayList()`	Constructor: Constructs a new IntArrayList object with an initial capacity of 10.	O(1)
-	`~IntArrayList()`	Destructor: (automatically called when this object goes out of scope) Frees any remaining dynamically-allocated memory associated with this IntArrayList. Do we need to write a destructor? If so, why?	O(N)
`void`	`add(int *value*)`	Appends the given value to the end of this list.	**O( )**
`void`	`insert(int *index*, int *value*)`	Inserts the given value at the specified index in this list. Shifts elements to the right to make room for the new element.	**O( )**
`void`	`clear()`	Removes all of the elements from this list. Should this call `remove`?	**O( )**
`int`	`get(int *index*)`	Returns the element located at the specified index in this list.	O(1)
`void`	`set(int *index*, int *value*)`	Replaces the element at the specified position in this list with the specified element.	O(1)
`int`	`size()`	Returns the number of elements currently in this list.	O(1)
`bool`	`isEmpty()`	Returns true if this list contains no elements.	O(1)
`void`	`remove(int *index*)`	Removes the element at the specified position in this list. Shifts elements left to cover the removed element and to maintain list continuity.	**O( )**
`string`	`toString()`	Returns a string representation of this list, e.g. "{a, b, c}".	O(N)

Your final code should comprise both `IntArrayList.h` and `IntArrayList.cpp`, except for `#include`, `using`, and `#pragma once` statements, which you may ignore here. Mark functions that do not modify the state of the `IntArrayList` as `const` when you write them. Define whatever private member variables or methods that you need. Also, fill in the empty slots in the runtime column above with the optimal **average-case** runtimes of those functions.

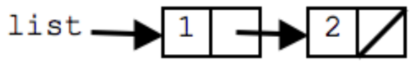
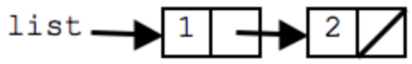
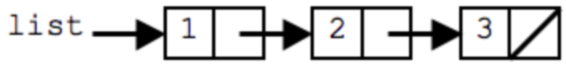
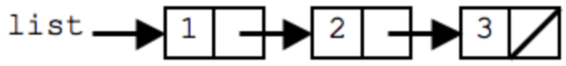
Bonus: add error checking by throwing exceptions when invalid index parameters are provided.

For problems #7 and 8, we'll be working with the ListNode structure, shown below. Draw step-by-step pictures when working through these problems!

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

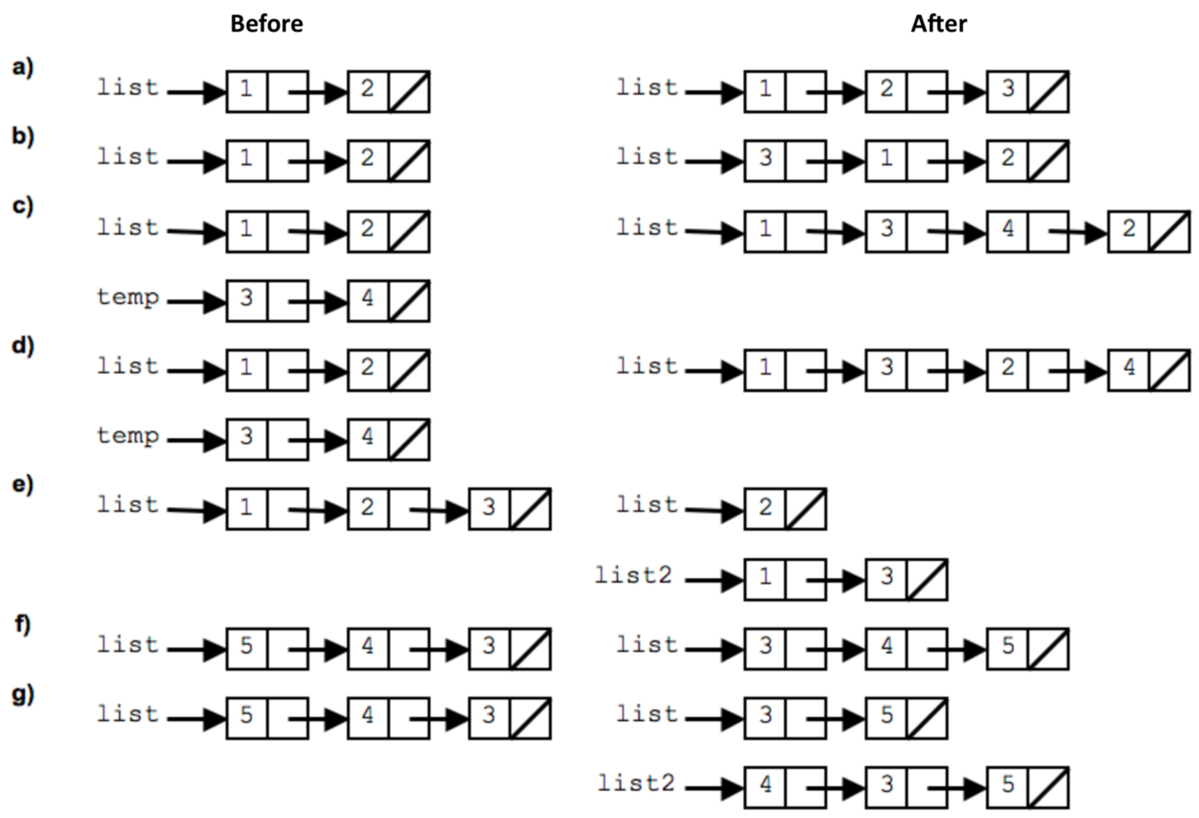
7. Linked Nodes *(pointers)*

Draw a picture of what the given nodes would look like after the code executes.

	Before / code	After
a)	 `ListNode newNode = {3, nullptr};` `list->next = &newNode;`	
b)	 `ListNode newNode = {3, list->next};` `list->next = &newNode;`	
c)	 `ListNode newNode = {4, list->next->next};` `list = &newNode;`	
d)	 `list->next->next = nullptr;`	

8. Linked Nodes 2 *(pointers)*

Write the code that will produce the given "after" result from the given "before" starting point by modifying links between the nodes shown and/or creating new nodes as needed. There may be more than one way to write the code, but do NOT change any existing node's data field value. If a variable does not appear in the "after" picture, it doesn't matter what value it has after the changes are made. Note that the variables `list`, `list2`, `temp`, etc. shown below are *pointers* to `ListNode`s.



Stack	**Heap**

Stack

Heap