Lab 11

CSE 165: Object Oriented Programming

Spring 2022

100 points

1. Counting Classes (15 points)

Create a class named `Counter` that contains an `int ID` and a `static int count`. The default constructor should begin as follows:

```
Counter( ) : ID(count++)
```

It should also print its ID and that it's being created. The destructor should print that it's being destroyed and its ID.

Do not decrement the count value in the destructor.

Submit your class in a file named Counter.h. Your class will be tested with the countingClasses.cpp file.

Expected output:

```
0 created
1 created
0 destroyed
1 destroyed
```

2. New and Delete Operators (15 points)

Add to your Counter class overloaded operators `new` and `delete` so that each will print "new" and "delete" respectively and then will allocate and delete memory using your own implementation, which may use malloc/free or the global new/delete operators.

Your code will be tested with the overloadedNewDelete.cpp file.

Expected output:

```
new
0 created
new
1 created
new
2 created
new
3 created
new
4 created
0
1
2
3
4
0 destroyed
delete
1 destroyed
delete
2 destroyed
delete
3 destroyed
delete
4 destroyed
delete
```

3. New[ ] and Delete[ ] operators (20 points)

Now overwrite the array versions of `new` and `delete` for your `Counter` class.

They should print "new N counters starting at S" and "delete counters", where:

- `N` is the number of objects being created
- `S` is the counter value of the first object being created in the array

As in the previous exercise, your overloaded operators should also correctly allocate and free memory as expected.

Your code will be tested with the arrayNewDelete.cpp file.

Expected output:

```
new 8 counters starting at 0
new 6 counters starting at 5
delete counters
delete counters
```

4. Self-deleting Objects (20 points)

A class can delete itself when it detects that it is no loner needed.

We are now going to use class `Shared` to implement the simplest version of what is called a "smart pointer", a way to achieve automatic memory management of objects allocated dynamically.

Create class named `Obj` deriving from `Shared`. It should print "New" and "Delete" from its constructors and destructor correctly so that the given code works. Save your class in a file named Obj.h and submit that file.

Use the files Shared.h, X.h, and selfDelete.cpp in order to test your code.

Expected output:

```
New Obj1
Delete Obj1
New Obj2
Obj2 is still here!
Delete Obj2
```

5. Graph Example (30 points)

In this exercise you are going to implement a graph object in an efficient way. Recall that a graph can be thought of as a collection of vertices that are connected to other vertices by edges.

There are many different ways to represent a graph in a computer program. In this exercise we will represent a graph as a vector of vertices, where pairs of adjacent vertices form an edge. For example, suppose we have 4 vertices, labelled: 1, 2, 3, and 4. Then consider the list:

```
1, 2, 1, 3, 1, 4, 2, 3
```

Edges are formed by reading the list two items at a time, so in the above graph vertex 1 is connected to 2, 3, and 4, vertex 2 is connected to vertex 3.

So we will be able to specify any graph as a vector of Vertex objects. One obvious observation is that many of the vertices listed above are repeats of the same vertex. For example, 1 appears in the list three times but all of the occurrences refer to the same vertex. When we implement the graph we may choose to create separate copies for each occurrence of a vertex in the list. This approach is inefficient so instead, we will simply store a pointer to each vertex.

The problem with this approach is related to freeing the memory taken up by our graph. We have a list of pointers to Vertex, with repeated entries. If we were to successfully delete the graph, we would have to call the delete operator once for every unique entry. We would not be able to simply call delete on each entry because when we get to the third entry of our example graph above, we would be attempting to delete 1, which would be already deleted before, so it would result in a Segmentation Fault.

One way to deal with this situation is to use a smart pointer, like the one that was introduced in the previous exercise. Use the files Shared.h, Vertex.h, Graph.h, and efficientGraphs.cpp, which implement the idea of using a smart pointer to be able to delete the graph. Parts of Graph.h have been omitted. Your task is to fill in the missing parts and upload the updated version of Graph.h.

Expected output:

```
Creating v1
Creating v2
Creating v3
Creating v4
Creating v5
v1 - v1
v1 - v3
v1 - v4
v1 - v2
v1 - v3
v1 - v4
v2 - v3
v2 - v4
v3 - v4
v5 - v1
v5 - v3
Deleting v2
Deleting v4
Deleting v1
Deleting v5
Deleting v3
```