C++ Classes

1. Destructors

A destructor is a member function with the same name as its class prefixed by a ~ (tilde). Destructors are usually used to deallocate memory and do other cleanup for a class object when the object is destroyed. It is automatically called when a class object passes out of the scope of the variable.

```cpp
{ // beginning of scope
    int i; // declare variable i
    // constructor function is called automatically at this point
    ...
} // end of scope
// variable i no longer exists here
// destructor function is called automatically at this point
```

The LinkedList.h file is shown next:

```cpp
#include <iostream>

class LinkedList{
private:
    struct ListNode {
        int data;
        ListNode *next;
    };

    ListNode *head;
public:
    // constructors are automatically called when the object is created
    LinkedList();

    // destructor is automatically called when the object passes out of scope
    ~LinkedList(); // a destructor takes no arguments and has no return type
};
```
The LinkedList.cpp file is shown next:

```cpp
#include <iostream>
#include "LinkedList.h"

LinkedList::LinkedList(){
    head = NULL;
}

LinkedList::~LinkedList(){
    ListNode *ptr = head;
    ListNode *temp;

    while(ptr != NULL){
        temp = ptr;
        ptr = ptr->next;
        delete temp;  // release the memory pointed to by temp
    }
}
```
2. **Exercises** (Problems with an asterisk are more difficult)

Experiments with memory leaks using the delete command and the destructor member function.

Use Applications/Utilities/Activity Monitor on the Macs to monitor the memory usage.

Ctrl-Alt-Del on the PCs to monitor the memory usage.

1. Know exactly when a destructor is called by adding three print statements in your code: one right before the destructor is called, one inside the destructor and one right after the destructor is called.

2. Using your LinkedList class without the destructor, write a program that keeps allocating nodes for the linked list (you don’t actually need to insert the nodes into the linked list) until you get a memory overflow error. Count how many nodes you need to allocated before a memory error occurs. Let \( \text{maxn} \) be this number. You might want to make the ListNode much larger so that \( \text{maxn} \) doesn’t have to be so large.

   Don’t print out \( \text{maxn} \) after each count because it will take too long. Print out \( \text{maxn} \) once every 100,000, e.g. if \((\text{maxn} \% 100000 == 0) \) cout << \text{maxn} << endl;

3. Think of a way to test the delete command. Insert \( \text{maxn} \) nodes into a new linked list, but after each insert, also call DeleteNode. Inside the DeleteNode function, DO NOT use the delete command. You should get a memory overflow error.

4. Same as number 3 but use the delete command inside the DeleteNode function. You should not get a memory overflow error.

5. Think of a way to test your destructor. One way is to declare \( \text{maxn} \) number of LinkedList, and for each one declared, you then exit the scope. DO NOT include a destructor for the LinkedList class. You should get a memory overflow error.

6. Same as number 5 but include a destructor for the LinkedList class. You should not get a memory overflow error.