1. (5 points) What is the difference between a class and an object?
   A class is a template for an object; it defines the member functions and the member variables. A class has no memory associated with it. An object is an instantiation of a class like in “int i” i is an instantiation of an integer. There is memory associated with an object. There can be many objects of the same class.

2. (5 points) The following is a legal and working program. What does it print? You must explain your answer for any credit. Be careful, this is a trick question.
   void find_max(int nums[], int size, int &max)
   {
     max = nums[0];
     for (int i = 1; i < size; i++)
     {
       if (nums[i] > max)
       {
         max = nums[i];
       }
     }
   }
   void find_min(int nums[], int size, int min)
   {
     min = nums[0];
     for (int i = 1; i < size; i++)
     {
       if (nums[i] < min)
       {
         min = nums[i];
       }
     }
   }
   int main()
   {
     int values[] = {-42, 17, 34, 42, 99};
     // this initialized the array

     int the_max = 0;
     int the_min = 0;
     find_max(values, 5, the_max);
     find_min(values, 5, the_min);
     cout << "max = " << the_max << " min = " << the_min << endl;
     return 0; // normal exit
   }

   max = 99, min = 0
   The find_max() function takes a reference to the integer max, so when it puts 99 in max it changes the_max in main(). The find_min() function takes a integer value for min, so when it puts -42 in min that value is NOT put in the_min.
3. (5 points) What does the following code print? You must explain your answer for any credit.

```cpp
#include <iostream>
using namespace std;

int main()
{
    int *i = new int;
    int *j = new int;
    *i = 42;
    *j = 42;
    if (i == j)
    {
        cout << "i == j" << endl;
    }
    else
    {
        cout << "i != j" << endl;
    }

double *x = (double *) i;
    cout << "*x = " << *x << endl;
}
```

It prints:
```
i != j
x = garbage number
```

i and j are pointers that point to different locations in memory. Thus they are not equal and “i != j” is printed.

x is a pointer to a double, but we assign it the address of an integer. *x dereferences the address held in x assuming it is a double (because x is a pointer to a double). Since that address in memory really holds and integer, a garbage number is printed.
The problem is that main.cpp includes foo.h twice (once directly (#include “foo.h”) and once indirectly via bar.h (main.cpp includes bar.h and bar.h includes foo.h)) and thus it appears to the compiler that class Foo is defined twice. The solution is to put the following code at the very top of foo.h

```cpp
#ifndef FOO_H
#define FOO_H

class Foo
{
public:
    Foo(int value);
    void print();
private:
    int m_value;
};
#endif
```

When it is compiled the following error is printed:

```
In file included from bar.h:2,
    from main.cpp:3:
foo.h:3: error: redefinition of 'class Foo'
foo.h:3: error: previous definition of 'class Foo'
make: *** [main.o] Error 1
```

What is the problem? Describe the best way to fix it (don’t worry about the exact syntax, just describe what you do to prevent this from happening).

The problem is that main.cpp includes foo.h twice (once directly (#include “foo.h”) and once indirectly via bar.h (main.cpp includes bar.h and bar.h includes foo.h)) and thus it appears to the compiler that class Foo is defined twice. The solution is to put the following code at the very top of foo.h

```cpp
#ifndef FOO_H
#define FOO_H
```
5. (10 points) Given class Foo and the following code, indicate which lines of code in main() cause Foo() and/or ~Foo() to execute. For example, if you think that the line “Foo g;” causes Foo() to be executed, put Foo() on the blank line to the left of “Foo g.”

```cpp
class Foo
{
    public:
    Foo() { cout << "Foo::Foo() called" << endl; }  
    ~Foo() { cout << "Foo::~Foo() called" << endl; } 
    void print() { cout << "Foo::print_foo() called" << endl; }
};

void print_foo(Foo *my_foo)
{
    my_foo->print();
}

int main()
{
    Foo g;        // declaring a new Foo object
    Foo *f;
    Foo f = new Foo();     // dynamically instantiating a Foo object
    print_foo(new Foo());  // dynamically instantiating a Foo object
    ~Foo()          // deleting a dynamically instantiated Foo
    ~Foo()          // local variable g goes out of scope
}
```

6. (15 points) Write the function List::find_element() that finds the element with the given index in the list. For example, if the index parameter is 1, the function should find the 1st element in the list. If the index is 2, it should find the 2nd element in the list. If the index is n, it should find the nth element. Return false if the list does not have that many elements.

```cpp
bool List::find_element(int index, int &value)
{
    for (Node *ptr = m_head; ptr; ptr = ptr->m_next)
    {
        if (index == 1)
        {
            value = ptr->m_value;
            return true;
        }
        index--;
    }

    return false;
}
```
7. (15 points) Write the function bool List::insert_sorted(int value) that inserts the given value into the list in such a way that the list is ordered from smallest to largest. If the number is already in the list, do not insert it and return false. If the number is not in the list, insert it and return true.

bool List::insert_sorted(int value)
{
    // if list is empty or new element belongs at front of list
    if (m_head == NULL || value < m_head->m_value)
    {
        m_head = new Node(value, m_head);
        return true;
    }
    else
    {
        Node *ptr = m_head;
        while (ptr->m_next != NULL && ptr->m_next->m_value < value)
        {
            ptr = ptr->m_next;
        }
        assert(ptr != NULL);
        // check to see if number already in tree
        if (ptr->m_value == value)
        {   return false;  }
        ptr->m_next = new Node(value, ptr->m_next);
        return true;
    }
}

8. (20 points) Write a function that removes all duplicate entries in the list. Assume that the list is ordered from smallest to largest, thus all duplicates will be next to each other. For example, if the list contained {1,2,2,2,3,3,4,5,5,5,5,5,6} before calling remove_duplicates(), it should contain {1,2,3,4,5,6} after. Do nothing if the list is empty or contains no duplicates.

void List::remove_duplicates()
{
    // consider all nodes in the list
    for (Node *ptr = m_head; ptr != NULL; ptr = ptr->m_next)
    {
        // while there is a next node
        // and next node has same value as the current node, remove next node
        while (ptr->m_next != 0 && ptr->m_value == ptr->m_next->m_value)
        {
            Node *tmp = ptr->m_next;
            ptr->m_next = ptr->m_next->m_next;
            delete tmp;
        }
    }
}