CSCI 211 **NAME** _____
March 5, 2010 **Minutes used:** _____

First Midterm Exam

- This test contains 9 questions worth a total of 88 points.
- Questions 1-4 have short answers and are 6 points each.
- Question 5-6 are a bit longer and are worth 8 points.
- Questions 7-9 require you to write code. They are worth 16 points each.
- You have 50 minutes to complete this exam.
- You may **not** use your text, notes, or any other reference material.
- The test with answers will be posted on the class webpage after the test.
- No electronic devices (music, phone, calculator, etc).
- **Do not turn this page until instructed to do so.**

Test Taking Advice

- The amount of space after a question does not always indicate how long the answer should be. Sometimes I add space so questions fit well on pages.

- Some questions have multiple parts such as "Explain your answer." Make sure you answer all the parts of each question.

- If you can't answer a question, move on and come back to it later. I often hear something like this, "I spent 40 minutes working on this 10 point problem and left 30 points worth of problems blank."

- If you have time left over, use it to review your answers. Students who turn tests in early often make trivial mistakes that they would catch if they went back over their answers. Some of my questions are difficult, go back and make sure you understood the question.

- If you don't understand a question ask me during the test. It is too late to ask for clarification after the exam.

1. (6 points) Define 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. (6 points) Which of the two following code fragments is better? Why?

```
// I think size is greater than 0 so this will work
for (int i = size; i != 0; i--)
      cout << i << endl;
```

```
assert(size > 0);
for (int i = size; i != 0; i--)
      cout << i << endl;
```

The second is better. In the second, if size is less than 0 the assert statement will print an error message and terminate the program.

In the first if size starts less than 0 the loop become infinite and the program will never terminate (actually it will terminate when i underflows (runs out of bits and the smallest negative number becomes the largest positive number) and then counts down to 0, but this process is long enough that we can call it "an infinite loop").

3. (6 points) When the following program runs it prints 76 and 86. Fill in the necessary blanks with the * and & operators so that the program prints 86 and 86. Not all of the blanks need an operator. Hint: you should think why I called the one variable *ptr*.

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

int main()
{

    int ___i = 76;
    int __*_ptr;

    ___ptr = _&_i;

    _*__ptr += 10;

    cout << ___i << endl;
    cout << _*__ptr << endl;

}
```

4. (6 points)The following function works most of the time, but sometimes it fails. Describe the problem. What situation causes it to fail? The List class definition is on the last page.

```
bool List::delete_head(int &value)
{
    value = m_head->m_value;
    Node *tmp = m_head;
    m_head = m_head->m_next;
    delete tmp;
    return true;
}
```

This function does not work if the list is empty (if m_head == NULL). When the list is empty and m_head == NULL, the right-hand-side of the first line (m_head->m_value) causes a segmentation fault because it is dereferencing a NULL pointer (which is out of the program's memory segment).

5. (8 points) **What does the following program print?** This question is a little bit tricky. Explain your answer or you will not get any points.

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

bool is_true()
{
    cout << "is_true()" << endl;          each time called “is_true()” is printed
    return true;
}
bool is_false()
{
    cout << "is_false()" << endl;         each time called “is_false()” is printed
    return false;
}


int main()
{
  if (is_true() && is_false())       is_true() returns true, is_false() is called
      cout << "1 true" << endl;
  else cout << "1 false" << endl;    prints false:  (true && false) is false

  if (is_false() && is_true())       is_false() returns false, don't call is_false()
      cout << "2 true" << endl;
  else cout << "2 false" << endl;    print false:   (false && true) is false

  if (is_true() || is_false())       is_true() returns true, don't call is_false()
      cout << "3 true" << endl;
  else cout << "3 false" << endl;    prints true:   (true || ANYTHING) is true

  if (is_false() || is_true())       is_false() returns false, must call is_true()
      cout << "4 true" << endl;
  else cout << "4 false" << endl;    prints true:   (false || true) is true
}
```

This program demonstrates the short circuit evaluations of expressions. Consider (A && B), if A is false, don't need to evaluate B because (A && B) is false. If A is true must evaluate B. Consider (A || B). If A is true no need to evaluate B (A || anything) is true. if A is false, must evaluate B.

```
is_true()
is_false()
1 false
is_false()
2 false
is_true()
3 true
is_false()
is_true()
4 true
```

6. (8 points) **What does the following program print?** This question is very tricky, be careful! Explain your answers or you will not get any points.

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

void f(int value)
{
    value += 1;
}
void g(int &value)
{
    value += 10;
}
void h(int *value)
{
    *value += 100;
}

int main()
{
    int i = 42;

    f(i);                       i is passed by value and thus not changed by f()
    cout << i << endl;          prints: 42

    g(i);                       i is passed by reference and thus change by g()
    cout << i << endl;          prints: 52

    h(&i);                      i is passed by address to h() and thus changed by h()
    cout << i << endl;          prints: 152

}
```

For the following questions, use the code on the last page. Tear off last page for easy reference.

7. (16 points) Write the function bool List::is_sorted_backwards() that returns true if the list is sorted backwards (from *largest* number to *smallest* number) and false if it is not sorted. Assume that an empty list is sorted.

```
bool List::is_sorted_backwards()
{
    // special case that the list is empty
    if (m_head == NULL)
        return true;

    Node *ptr = m_head;
    while (ptr != NULL && ptr->m_next != NULL)
    {
        // if any pair of neighbors in the list are out of order, then the
        // entire list is not sorted
        if (ptr->m_value < ptr->m_next->m_value)
            return false;
        ptr = ptr->m_next;
    }
    // all pairs of neighbors are in order, so the entire list is sorted
    return true;
}

// here is a shorter version of the above code
// this code handles the empty list case (special case code above isn't necessary)
bool List::is_sorted_backwards()
{
    for (Node *ptr = m_head; ptr && ptr->m_next; ptr = ptr->m_next)
    {
        if (ptr->m_value < ptr->m_next->m_value)
            return false;
    }
    return true;
}
```

8. (16 points) Write the function void 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. Insert the number into the list even if it is already in the list. Assume this list **is** sorted when the function is called.

```
void 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);
    }
    else
    {
        Node *ptr = m_head;
        // search for the node the new number should be inserted after
        while (ptr->m_next != NULL && ptr->m_next->m_value < value)
        {
           ptr = ptr->m_next;
        }
        assert(ptr != NULL);
        ptr->m_next = new Node(value, ptr->m_next);
     }
}
```

9. (16 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
    Node *ptr = m_head;
    while(ptr != NULL)
    {
        // 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;
        }
        ptr = ptr->m_next;
    }
}

The following is another solution
void List::remove_duplicates()
{
    Node *ptr = m_head;
    while(ptr != NULL && ptr->m_next != NULL)
    {
        if (ptr->m_value == ptr->m_next->m_value)
        {
            Node *tmp = ptr->m_next;
            ptr->m_next = ptr->m_next->m_next;
            delete tmp;
        }
        // the "else" below is very important
        // without it, this function would not remove all duplicates
        else ptr = ptr->m_next;
    }
}
```

Use the following class definitions for questions 7,8, & 9. You **may not** alter or add to this class definitions.

You may tear this page off so it is easier to reference.

```
class List
{
   public:
     List() {m_head = NULL;}
     void insert(int value); // inserts value at front of list
     bool delete_head(int &value) ;
     bool is_sorted_backwards();
     void insert_sorted(int value)
     void remove_duplicates();

   private:
     class Node
     {
         public:
           Node (int value, Node *next) {m_value = value; m_next = next;}
           int m_value;
           Node *m_next;
     };
     Node *m_head;
};
```