Your work for this must be finished and shown to your lab TA the end of this lab session or the start of your next session.

Download the lab files from the course web page (in lab3.zip) and extract them into a new sub-directory ("lab3") of your "cs221" directory. Make that your current working directory and then run "make dates" to get an intial compile for Q1, and do a test run.

The commands needed to do everything specified in the above paragraph (except downloading the code and extracting its contents) are given below. This assumes you are currently in your home directory, which is the directory you are in after successfully logging in to your account.

```
cd cs221
mkdir lab3
```

Put the extracted files into cs221/lab3 now

```
cd lab3
make dates
./dates
```

Q1. Complete the missing code in "CDate.cc". You do not need to change any other file for this question, but you need to understand what they do, and how they are used. The dot-h file (or "header") contains the class declaration, and it's #include'd in both its associated dot-cc file, and any other files that need to use CDate objects (in this lab, just the dates.cc file).

Read "dates.cc". Note the two different ways to construct a CDate object, and the extra statement that's used when the source is compiled under Windows. Most of the source code we supply will not have these lines included, so if you are running under Windows take note of them.

When you complete the code correctly, you will see the following output:

```
2015/1/1
0/0/0
0/0/0
0/0/0
2000/2/29
0/0/0
2014/12/31
0/0/0
2010/11/30
0/0/0
2012/2/29
2014/9/5
All tests passed.
```

Some helpful links: http://www.cplusplus.com/reference/string/string/compare/ and http://www.cplusplus.com/doc/tutorial/control/#switch
Note: hyperlinks are in write-ups to help you avoid common errors. READ THEM.

Q2. The second half of this lab deals with a recursive data structure called a linked_list. Code that maintains or manipulates recursive structures is usually easier to write recursively. Compile and do an initial test run.

```
make lists
./lists
```

When you complete the functions correctly, the unit tests will all pass. Drawing pictures of the possible configurations that must be handled in each method will help.

Your work for this must be finished and shown to your lab TA the end of this lab session or the start of your next session.

```
/**
 * Delete the last Node in the linked_list
 * PRE: head is the first Node in a linked_list (if NULL, linked_list is empty)
 * POST: the last Node of the linked_list has been removed
 * POST: if the linked_list is now empty, head has been changed
 */
void delete_last_element( Node*& head );

/**
 * Removes an existing Node (with key=oldKey) from the linked_list.
 * PRE: head is the first node in a linked_list (if NULL, linked_list is empty)
 * PRE: oldKey is the value of the key in the Node to be removed
 * POST: if no Node with key=oldKey exists, the linked_list has not changed
 * POST: if a Node with key=oldKey was found, then it was deleted
 * POST: other Nodes with key=oldKey might still be in the linked_list, but
 * POST: if the linked_list is now empty, head has been changed
 */
void remove( Node*& head, int oldKey);

/**
 * Insert a new Node (with key=newKey) after an existing Node (with key=oldKey)
 * If there is no existing Node with key=oldKey, then no action.
 * PRE: head is the first Node in a linked_list (if NULL, linked_list is empty)
 * PRE: oldKey is the value to look for (in the key of an existing Node)
 * PRE: newKey is the value of the key in the new Node (that might be inserted)
 * POST: If no Node with key=oldKey was found, linked_list has not changed
 * POST: Else a new Node (with key=newKey) is right after Node with key=oldKey.
 */
void insert_after( Node* head, int oldKey, int newKey );

/**
 * Create a new linked_list by merging two existing linked_lists.
 * PRE: list1 is the head of a linked_list (if NULL, then it is empty)
 * PRE: list2 is the head of another linked_list (if NULL, then it is empty)
 * POST: A new linked_list is returned containing new Nodes with the keys from
 * the Nodes in list1 and list2, starting with the key of the first Node of
 * list1, then the key of the first Node of list2, etc.
 * When one list is exhausted, the remaining keys come from the other list.
 * For example: [1, 2] and [3, 4, 5] would return [1, 3, 2, 4, 5]
 */
Node* interleave( Node* list1, Node* list2 );
```

When you complete the code correctly, you will see the following :

```
<A> List 1: [3, 2, 1]
<B> List 2: [6, 7, 8, 9, 10]
<C> List 1: [3, 2]
<D> List 1: [3]
<E> List 1: []
<F> List 1: []
<G> List 1: [11, 12]
<H> List 1: [11, 12]
<I> List 1: [11, 12, 12]
<J> List 1: [11, 12]
<K> List 4: [11, 6, 12, 7, 8, 9, 10]
<L> List 4: [6, 6, 7, 7, 8, 8, 9, 9, 10, 10]
<M> List 4: [11, 12]
<N> List 4: [6, 7, 8, 9, 10]
<O> List 4: []
All tests passed.
```

Q3. (Optional) If you used a recursive approach for the interleave method, create another method using an iterative approach. If you used an iterative approach, now use a recursive approach.