# CSCI-1200 Computer Science II — Spring 2006
# Lab 12 — Linked Lists

Please download two files from the course web site. The first is the header file defining the `cs2list` class plus the node and iterator classes. The second is a main program to test this class. You will need to add to both. The program will not compile until you've added the code for Checkpoint 1.

```
http://www.cs.rpi.edu/academics/courses/spring06/cs2/labs/12_linked_lists/cs2list.h
http://www.cs.rpi.edu/academics/courses/spring06/cs2/labs/12_linked_lists/test_cs2list.cpp
```

Recall that thinking about writing linked-list code is best done visually. For each checkpoint, draw picture showing the linked list before and after each operation. Draw separate pictures for each special case you must handle.

Debugging linked list code can be challenging. Reasons for this include the difficulty of visualizing what's happening and the fact that the impact of errors (incorrectly assigned pointers, usually!) often appears later in the code and in a different place from where the mistake was actually made. Three methods can be used in debugging, often in combination, to find and fix the errors. The first is writing functions to print the contents of a list. Calls to this function should be inserted in many places in the code and compared to the output you expect. The second is testing the functions on small examples and special cases. The third is using a debugger to step through the contents of a linked list, following the pointers. Try to apply these techniques as you work toward solutions to the following checkpoints.

Finally, writing and compiling the code in a templated class is not like writing and compiling code for an ordinary class. First, there's the somewhat strange templating syntax for functions defined outside of the class prototype; see the definition of `push_front`. Second, the code isn't compiled separately. Instead, it is included in a source file and compiled where it is "instantiated" by creating a templated object. In fact, templated functions that are not needed are not compiled! Finally, often the header and source files are not separated, but instead are usually all placed in one header file.

## Checkpoint 1

Implement the `pop_front` and `pop_back` member functions. Start by re-examining the `push_front` member function we wrote for a Lecture 21 exercise. Then, outline carefully what you need to do. Here are several suggestions:

- Use a local pointer variable (not a new node!) to point to the node that needs to be deleted.
- Check for the special case of deleting the only node in the linked list.
- Make sure you decrement the size counter.
- Do not explicitly delete the node until the very end of the function, after it has been removed from the linked list.

The code in the main program has a number of checks to make sure you've done this correctly.

**Complete this checkpoint** by adding more code to test all 4 push & pop member functions. Use `push_front` and `push_back` to guide you, both in terms of the logic and in terms of the format of the function definitions (in `cs2list.h`). The code in the main program is designed to help you start testing these functions.

### Checkpoint 2

Write the `insert` function. Consider the prototype that you need to write when you insert this function at the bottom of `cs2list.h`:

```
template <class T>
typename cs2list<T>::iterator cs2list<T>::insert(iterator itr, const T& v)
```

At first this looks odd. The return type is written `cs2list<T>::iterator`, but inside the argument list we just use `iterator`. These refer to EXACTLY the same type. Why then is there a difference? The answer is "scope": the return type is specified outside the scope of the `cs2list<T>` class and therefore the class scope operator `cs2list<T>::` needs to be used. The inside of the argument list is within the scope of the `cs2list<T>` class and so the scope operator is no longer needed.

Furthermore, we have a new keyword `typename`. It eliminates a warning issued by advanced compilers. It is not clear to the compiler that `cs2list<T>::iterator` is a real type because `T` hasn't yet be specified. If you didn't get that, don't worry about it. You will not be tested on it!

Now, let's look at the arguments themselves and use this to explain the exact purpose of the function. The first argument is an iterator that refers to a node in the list. The pointer to this node is stored in the iterator:

```
itr.ptr_
```

Because `cs2list<T>` is a friend of the iterator class, the `insert` function has direct access to the value (using the syntax `itr.ptr_`). The second argument to the function is the value to be added to the list. The job of the function is to

1. Create a new node containing the value,
2. Insert the node **before** the node pointed to by `itr.ptr_`, and
3. Return an iterator that contains the address of the new node as its `ptr_` value.

Be sure you manipulate the pointers correctly (**DRAW A PICTURE**). Be sure you consider the special case when `itr.ptr_ == head_`.

**To complete this checkpoint** demonstrate your debugged code to a TA. There is code in the main program that has been commented out. It will help you test this function.