17.5
The STL list Container
And now that you know how doubly linked lists work...

C++ provides a template for a doubly linked list:

- `#include <list>
- `list<double> list1;
- `list<YourOwnClassType> list2;

A part of the Standard Template Library (STL)
THE STL LIST CONTAINER

- Member functions for
  - locating beginning and end of list:
    - **front** – return first element in the list
    - **back** – return last element in the list
    - **begin** – return a bidirectional iterator that points to the beginning of the list
    - **end** – return a bidirectional iterator that points to the end of the list
  - adding elements to the list:
    - **insert** – insert a value at the given iterator position
    - **merge** – merge two lists into one
    - **push_back** – add element at the end of list
    - **push_front** – add element at the front of list
THE STL LIST CONTAINER

- Member functions for
  - removing elements from the list:
    - **erase** - erase element pointed to by an iterator
    - **pop_back** - remove the last element
    - **pop_front** - remove the first element
    - **unique** - remove all duplicates from the list
  - querying list size:
    - **empty** - return true if list is empty, false otherwise
    - **size** - return the number of elements in the list

See Table 17-1 for a list of member functions
THE STL \texttt{LIST} CONTAINER: ITERATING

```cpp
#include <string>
#include <list>
using namespace std;
int main() {
    list<int> ll;
    for(int i=0; i < 6; i++) ll.push_back(i);

    list<int>::const_iterator it; // read-only iterator
    it = ll.begin(); // assign to the start of the list
    while (it != ll.end()) // while hasn't reach the end
    {
        cout << *it << " "; // print the current value
        it++; // and iterate to the next element
    }
    cout << endl;
}
```
#include <string>
#include <list>
using namespace std;

int main() {
    list<int> ll;
    for(int i=0; i < 6; i++) ll.push_back(i);

    list<int>::reverse_iterator it;  // reverse iterator
    it = ll.rbegin();  // assign to the end of the list
    while (it != ll.rend()) // while hasn't reach the end
    {
        cout << *it << " ";  // print the current value
        it++;  // and iterate to the previous element
    }
    cout << endl;
}
THE STL \texttt{LIST} CONTAINER

- **Question:** What is more efficient: adding elements to a \texttt{vector}, or adding elements to a \texttt{list}?
  - List is more efficient: data need not be shifted or copied when an element is added

- **Question:** What is more efficient: reading an element from a \texttt{vector}, or reading an element from a \texttt{list}?
  - If it is the first or the last element, equally efficient
  - If it is an element in the middle of a structure, vector is more efficient
Conclusion:

- **Vector** is more efficient when you **read** more than you **write**: add few elements every now and again, use random access a lot.

- **List** is more efficient when you **write** more than you **read**, and when you read in **sequence**: add elements rapidly on either side of the list, find an element in the middle of the structure every now and again;
  - **Stack, queue data structures**: read the first/last elements only.
What is more efficient?

```cpp
while (myList.size() != 0)
{
    cout << myList.back();
    myList.pop_back();
}
```

empty() is much more efficient than size()

Why?

Because size() has to iterate through the whole list, and empty() just checks if head is NULL
LIST<T> FOR NON-PRIMITIVE TYPES

- What if you want a doubly linked list of wizards, or students, or unicorns?

- What is so special about non-primitive types?
  - Comparing Wizards or Unicorns with a comparison operator such as < or != is not defined
  - ...Unless you overload the corresponding operators
  - STL linked list algorithms rely on comparison operators
THE STL LIST CONTAINER

- To conclude, STL provides an efficient doubly-linked list implementation
- Play around with the functions that are available:
  - [http://www.cplusplus.com/reference/list/list/](http://www.cplusplus.com/reference/list/list/)
- If a ready-to-use linked list is available, why bother coding it yourself?
- As an exercise: because you need to know how each data structure works, so that you would use the most appropriate one in each specific case
- No lectures/pracs/tuts next week
- Happy spring everyone!