More STL Templates

• Avoid reinventing the wheel!
• Keep code portable.
• Get just the functionality you need, and...
• ...preserve the efficiency you want.
• Helps to know about data structures.

std::vector<class T>

• Supports random access
• Provides fast appending and truncation...
• ...but slow for internal inserting and erasing.
• Provides dynamic resizing.
• Owns its own memory.
std::vector construction

explicit vector (const A& al = A());

explicit vector (size_type n, const T& v = T(),
    const A& al = A());

vector (const vector& x);

vector (const_iterator first, const_iterator last,
    const A& al = A());

Note that using the second constructor requires either a default constructor
for T or an explicit value for the second argument.

std::vector access

• Access is limited to the defined range of the vector. Other access is
  either an error (with at) or undefined (with operator[]).
• Can be used on either side of assignment

const_reference operator[] (size_type pos) const;

reference operator[] (size_type pos);

const_reference at (size_type pos) const;

reference at (size_type pos);
more std::vector access

• Access to elements at the ends of the vector is made very easy
• Again, can be used on either side of assignment

reference front();
const_reference front() const;

reference back();
const_reference back() const;

std::vector as a stack

• Adding and removing elements at the end is also made very easy
• Note that push_back makes a copy of the argument to put into the vector

void push_back(const T& x);
void pop_back();
void clear();  // remove everything
Getting information

size_type size() const;
bool empty() const;
size_type capacity() const;

You can use resize to change the size of a vector. If it gets longer, the new elements are initialized with the second argument.
void resize(size_type n, T x = T());

reserve does the same thing for capacity, but without any initialization. Current contents are preserved.
void reserve(size_type n);

std::pair<class T, class U>

• Supports *duplicates*
• A pair has two public members, first and second.

template<class T, class U>
struct pair {
    typedef T first_type;
    typedef U second_type
    T first;
    U second;
    pair();
    pair(const T& x, const U& y);
    template<class V, class W>
    pair(const pair<V, W>& pr);
};
std::pair<class T, class U>

• Default construction
• Construction from two items
• Construction from another pair (even with other types)
• The function make_pair(item1, item2) makes a pair.

```cpp
pair();
pair(const T& x, const U& y);
template<class V, class W>
  pair(const pair<V, W>& pr);
};
```

Other simple STL containers

• stack
• queue
• deque
• list
Iterators

- Iterators are like pointers
- Minimally, they can be:
  - incremented
  - compared
  - dereferenced
  - used with ->
- Come in const and non-const varieties, just like pointers.
- Two iterators on the same structure define a range.

Declaring iterators

- std::vector declares two typedefs:

```cpp
using std::vector;

vector<int> v1;
vector<int>::const_iterator iter1;
vector<int>::iterator iter2;
```
Using iterators

- begin() and end() provide special iterators for a vector:

```cpp
using std::vector;
vector<int> v1;
... 
int total = 0;
vector<int>::const_iterator iter = v1.begin();
while( iter != v1.end() ) 
{
    total += *iter;
    ++iter;
}
```

Using iterators, cont

- Two iterators form a range, closed on the left but open on the right.

- insert uses an iterator to say where to insert a new item
  (returns iterator pointing to new element):
  ```cpp
  iterator insert(iterator it, const T& x = T());
  ```

- erase uses an iterator to tell it where to erase a new item:
  ```cpp
  iterator erase(iterator it);
  ```

- Or a whole range of items:
  ```cpp
  iterator erase(iterator first, iterator last);
  ```

- Values returned point to the next item after the deleted ones
Mistakes when using iterators

- end() does not point to anything...
- So you can’t dereference it!
- You can’t use -> on it either.
- Iterators can become “stale” and hence unsafe to use.

Kinds of iterators

- Iterators on vectors are random-access
- This means they can be
  - decremented
  - added with integers to give new iterators
- Other types include forward and bidirectional

E.g.:
```cpp
vector<int>::const_iterator iter = v1.begin();
iter += 7; // Now iter points to 7th element!

// this will be expensive, maybe use a list instead?
iter = v1.insert(iter, 3);
iter = v1.insert(iter, 8);
```
The iterator abstraction

• Iterators are a widely used abstraction in the STL
• The idea is to have something that can be used with arrays, strings, lists, and even trees.
• Combined with liberal typedef'ing, it should be possible to switch representations with a minimum of other changes.
• E.g.:

```cpp
class Foo {
    typedef vector<int> StorageType;
    typedef StorageType::const_iterator CIter;
    typedef StorageType::iterator Iter;
    StorageType m_buffer;
    ...Now write a lot of code using these typedefs.
    Later:
    typedef list<int> StorageType;
}
```

Template Metaprogramming

```cpp
// factorial.cpp
#include <iostream>

// factorial.cpp
#include <iostream>

template <int N>
struct Factorial
{
    enum { value = N * Factorial<N-1>::value }
};

template <>
struct Factorial<1>
{
    enum { value = 1 }
};

// example use
int main()
{
    const int fact15 = Factorial<15>::value;
    std::cout << fact15 << endl;
    return 0;
}
```
Advice

• Be a client!
• Use iterators where possible
• Helps to know about data structures.
• Use typedefs privately or publicly for classes that include vectors.
Object Oriented Programming in C++

Assignment 4 -- Due April 9

In this assignment we will use some pieces of the STL to write a text utility class. Throughout this assignment, you should focus on being a client of STL classes that provide functionality you need. Try actively to avoid writing your own structures and classes!

In this assignment, we will write a class TextUtil that provides the following interface:

```cpp
using std::string;
using std::vector;
using std::pair;

typedef vector<pair<int, int>> IntPairVector;
typedef vector<pair<string, int>> StringIntVector;
class TextUtil
{
  public:
    TextUtil(const string& filename);
    IntPairVector wordLengths() const;
    size_t wordCount(const string& word) const;
    StringIntVector topNWords(size_t n) const;
  ...
};
```

Part 1.
Write the constructor and the wordLengths function. The constructor can either just store the name of the file, or it can open it and read it into some internal storage for use by the other functions.

wordLengths() finds and returns the number of words of different lengths in the file of text. Each pair in the vector returned should consist of a length and the number of words of that length.

You should make sure that you strip any punctuation off the beginning and end of the words in the file before you record their length. For the purposes of this assignment, punctuation is anything other than a character from A to Z or from a to z. Note that you do not have to worry about punctuation that is internal to a word, like this-silly#$%^@_example.

Your vector should only contain the lengths that actually occurred, so no pair should have 0 as the second item. As an example, if the file contained "Now is the time for all good men to come to the aid of their country" then the array returned would contain the following pairs:

2 4
3 7
4 3
The following sample program shows how to do word-level file input.

```cpp
#include<iostream>
#include<fstream>
#include<string>
using std::string;
using std::ifstream;
using std::cerr;
using std::endl;
using std::cout;
using std::cin;

int main()
{
    cout << "Enter the name of a text file: ";
    string filename;
    // Note: this requires that the filename have no spaces!
    cin >> filename;

    // class ifstream defines an input stream from a file.
    // The c'tor that takes a filename also opens the file (for reading, by default)
    ifstream file ( filename.c_str() );

    // Test to see if file is now open; die otherwise
    if ( ! file.is_open() )
    {
        cerr << "Failed to open file: " << filename << endl;
        return -1;
    }

    string word;
    string longest;

    // Read through the file one word at a time. word will contain
    // individual words that are separated by whitespace. The
    // whitespace is
    // just discarded. Note that words with punctuation after them will
    // still
    // have it there (for example, this comment would return words like
    "(for" and
    // "example,")
```
while ( file >> word ) {
    if ( word.length() > longest.length() )
        longest = word;
}
    cout << "The longest word in " << filename << " is " << longest << endl;
    return 0;
}

**Part 2.**

Write the second two functions. *size_t wordCount(const string& word) const;* gives the number of occurrences of *word* in the text file. Again, you should make sure that you strip any punctuation off the beginning and end of the words in the file before you count them. You may assume that the argument to this function is already stripped of punctuation.

*StringIntVector topNWords(size_t n) const;* returns a vector of the *n* most frequent words in the file, paired with their counts. You should write this function without writing a sorting algorithm of your own.