Exercise 7.1 – Make a catalogue of English words

Online STL reference: `https://en.cppreference.com/w/cpp/container`

- The goal of this exercise is to use C++ STL classes to make a catalogue of all the English words that occur in a given text file, including the number of times that each word occurs.
  a) Write a small `main()` program that reads in file `ex5.3/example.txt` word by word. Open the file using an `ifstream` class and read each word into a STL class string using `operator>>`. As a first step in the development of your program, print each word as your read it.
  b) The next step is to use an STL map to keep track of all the words. Create an object name `myMap` of the type `map<string,int>` in the beginning of your program that will hold the catalogue of words. Then replace the line of code in your reading loop that prints out the word with a line of code that stores it: `myMap[word] += 1`. Explain what the preceding line of code does.
  c) Finally, at the end of your program – once you have processed all words in the file – print out the contents of `myMap` using the iterator mechanism: First create an iterator that points to the first element of `myMap` using the `begin()` method. Then create a `while()` loop that compares the iterator to the value of `myMap.end()`, which returns an iterator pointing to the last element in the `map()`. Inside the loop, print out each map element. Remember that a map stores a pair of values and the iterator points to a `pair` object, which has data member `first` and `second` that hold the key and value of each map entry. Finally, use the `++` operator to move the iterator to the next element inside the while loop.

Exercise 7.2 – A palindrome tester

Online STL reference: `https://en.cppreference.com/w/cpp/container`

- The goal of this exercise is to write a program that checks if a vector is a palindrome
  - A palindrome is a sequence of values that reads the same forward as backward (e.g. 1,2,3,2,1)
- Approach – `vector<int>` palindromes

  a) Create a small `main()` program that allocates a `vector<int>`

  b) Now write a piece of code that checks if that vector is a palindrome. To do so use two iterators: one that starts at the front and is incremented and one that starts at the end and is decremented. Compare elements pointed to by both iterators at each step, if there is a mismatch the vector is not a palindrome

  - Functions `begin()` and `end()` return iterators starting at the begin and end respectively.
  - Note that the iterator returned by `end()` is positioned *beyond* the end of the collection – you need to decrement it once to put it on the last element.
- Approach – `vector<T>` palindromes

  c) Write a template function `bool isPalindrome(vector<T> v)` that can perform the check for a vector of any type

  - *You can ignore any g++ compiler warnings about 'implicit type' that you may get*
  - *But you will need to use typename here. For a (technical) discussion, see e.g. http://pages.cs.wisc.edu/~driscoll/typename.html ; for a brief summary, look under "Before a qualified dependent type"*

Exercise 7.3 – STL performance comparison

- The goal of this exercise is to compare of the performance of mid-collection insertion of elements in a STL `vector` vs insertion in a STL `list`.
  - Class vector is not very efficient in mid-collection insertion and removal of elements, because the data is organized such that all elements beyond the insertion point need to be copied to one slot higher or lower respectively. The doubly-linked list structure of list is much more suitable to handle mid-collection insertions and should result in faster execution times.

- Approach

  a) Write a small program that fills a `list<int>` with 10000 sequential numbers.

  b) Iterate over the loop and remove every 3rd element of it by calling the `erase(list<T>::iterator)` function.
     - Note that when you remove an element the iterator still points to the (now gone) element, which will cause trouble if you will use it later on to navigate further through the list. To solve this problem, the erase() function returns you a new value for the iterator that will allow you to continue without trouble. So be sure to assign the return value of erase() to your iterator.

  c) Run your program and time its CPU consumption with `/bin/time <your.exe>`

  d) Replace `list` with `vector` in the above program and run again. Does the execution time change? Now change the length of the collection from 10.000 to 100.000 and try again.

Exercise 7.4 – Class Stack revisited

Online STL reference: `https://en.cppreference.com/w/cpp/container`

- Take `Stack.hh` from exercise 6.3 and reimplement it using the STL template class `deque`.
- Approach
  a) Replace data member `Array<T>` by `std::deque<T>` and remove data member `int count`. Replace all uses of count by `s.size()`.

  b) Change the constructor to no longer explicitly initialize data member `s` with a size, as class `deque` takes care of this internally. Eliminate member function `full()` as a `deque` is never full.

  c) Reimplement functions `push()` and `pop()` in terms of `deque` functions `push_back()` and `pop_back()`. You can eliminate the growth feature from `push()` as `deque` handles this internally. Note that `deque::pop_back()` does not return a value, it merely removes the element. You should retrieve it using function `T deque<T>::back()` before you pop it off.

  d) Reimplement function `inspect()` using an iterator over `s`.
  - You will need `typename` again