Prac 4: Words

Task

Your task is to write a program, **words**, that reports information about the number of words read from its standard input. For example, if the file **qbf.txt** contains the text

```
the quick brown fox jumps
over the lazy dog
```

then running the program with is input redirected to come from that file will report 9 words

```
$ words < qbf.txt
Total: 9
```

Automatic Testing

This task is available for automatic testing using the name **words**. You can run the demo using **demo words** and you can test your work using **try words <<filename>>**.

Background

Use the following skeleton for the program:

```
int main (int argc, char** argv) {
  enum { total, unique } mode = total;
  for (int c; (c = getopt(argc, argv, "tu")) != -1;) {
    switch(c) {
    case 't': mode = total; break;
    case 'u': mode = unique; break;
    }
  }
  argc -= optind;
  argv += optind;
  string word;
  int count = 0;
  while (cin >> word) {
    count += 1;
  }
  switch (mode) {
  case total: cout << "Total: " << count << endl; break;
  case unique: cout << "Unique: " << "** missing **" << endl; break;
  }
}
```

The **getopt** function (**#include <unistd.h>**) provides a standard way of handling option values in command-line arguments to programs. It analyses the command-line parameters **argc** and **argv**, looking for arguments that begin with '**-**'. It then examines all such arguments for specified option letters, returning individual letters on successive calls and adjusting the variable **optind** to indicate which arguments it has processed. Consult **getopt** documentation for details.

In this case, the option processing code is used to optionally modify a variable that determines what output the program should produce. By default, **mode** is set to **total** (indicating that it should display the total number of words read). The **getopt** code looks for the **t** and **u** options (which would be specified on the command line as **-t** or **-u**) and overwrites the **mode** variable accordingly. When there are no more options (indicated by **getopt** returning **-1**), **argc** and **argv** are adjusted to remove the option arguments that **getopt** has processed.

Level 1: Unique words

Extend the program so that if run with the **`u`** option (specified by the command-line argument **`-u`**), it displays the number of unique words. To count unique words, use the STL **`vector`** class to keep track of which words you've already seen. When each word is read, check to see if it's already in the vector; if it's not, insert it. The number of unique words will then be the size of the vector.

Level 2: Your Own Vector

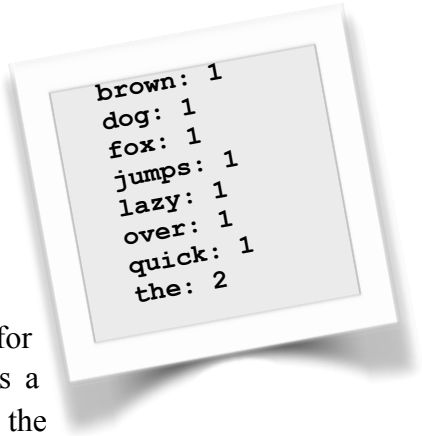
Modify your code so that it does not use the STL classes. You'll need to write your own class to keep track of the list of words. At this level, you can assume that there will be no more that 1000 unique words, so you can use a statically allocated array to store the items.

A minimal STL-compliant class (which will minimise the changes you need to make to your main program) would have an interface like this:

```
template <class T>
class Vector {
public:
  typedef T* iterator;
  Vector () { << code >> }
  iterator begin () { << code >> }
  iterator end () { << code >> }
  int size () { << code >> }
  iterator insert (iterator position, const T& item) { << code >> }
private:
  T items[1000];
  int used;
};
```

Note that, unlike an ordinary class, a template class is defined entirely in a header file, so there won't be any **`Vector.cpp`** file. Also, you'll find it easier to define the template methods within the class scope because it avoids the need to specify the scope for each method individually.

Level 3: Individual Word Counts



Extend the program so that if run with the **`i`** option it displays the counts of individual words in alphabetical order. For example the command

```
words -i < qbf.txt
```

should result in the output shown.

Your program will need to store two pieces of information for each word. Define a struct called **`WordInfo`** that contains a **`string`**, **`text`**, for the word's text, and an **`int`**, **`count`**, for the number of times it's been seen. Then create a vector of **`WordInfo`** instead of a vector of **`string`**. When you add a new word, give it a count of 1. When you see a word that's already in the vector, increment its count.

The simplest way to display the result in alphabetic order is to keep the vector in sorted order as words are added, then simply iterate through the vector at the end. The insertion sort algorithm is a suitable way to find where a new word should be inserted. Iterate through the list until you find a word that's alphabetically after the new word, then insert at that position.

Level 4: Large Data Sets

Make sure that your program works correctly (and efficiently!) even if it is run with large data sets. Since you don't know how large the collection of words might become, you'll need to make the vector grow dynamically. A suitable strategy is to allocate space for a small number of items initially, and then check at each insert whether or not there's still enough space. When the space runs out, allocate a new block that's twice as large, copy all of the old values into the new space, then delete the old block.