# Practical 4: Words


## Task

Your task is to write a program, **words**, that reports information about the number of words read from standard input. For example, entering the following text should cause the program to report 9 words:

```
./words // or words.exe (of course, you can just run from within CLion)
the quick brown fox jumps over the lazy dog
Total: 9
```

## SVN check out

Before you begin, you will need to check out the practical directory from the topic repository. This can be achieved by selecting Get from Version Control on the CLion welcome screen. In the window that appears, select Subversion from the dropdown list. If you have not previously connected CLion to the repository, click the add button (**+**) and paste the following URL into the text field (replacing **FAN** with your FAN):

```
https://topicsvn.flinders.edu.au/svn-repos/COMP2711/FAN
```

If you are prompted to login, use your University FAN and password. Expand the repository listing and select the **words** directory. Click Check Out then select a location to store your project (preferably in a practicals directory) and click Open. From the destination list, choose the second option to create a project directory named **words** and then click OK to complete the check out. A dialogue may appear confirming the subversion working format; if so, 1.8 is fine. Finally, when prompted if you would like to open the project, click Yes.

## Automated marking

This practical is available for automatic marking via the quiz **Practical 4**, which is located in the Assessment module on FLO. To assess your solution, first commit your changes to the topic repository. This can be completed within CLion via the Commit option under the VCS menu (or alternatively with ⌘+K on macOS or Ctrl+K on Windows).

You should adhere to good development habits and enter a commit message that describes what you have changed since your last commit. When ready, click Commit to upload your changes and receive a revision number. Enter this revision number into the answer box for the relevant question (level).

There are no penalties for incorrect solutions, so if you do not pass all test cases, check the report output, modify your solution, commit, and try again. Remember to finish and submit the quiz when you are ready to hand in. You may complete the quiz as many times as you like—your final mark for the practical will be the highest quiz mark achieved. If you do start a new quiz attempt, ensure you reassess any levels you have previously completed.

## Background

The following skeleton code for the program is provided in **words.cpp**, which will be located inside your working copy directory following the check out process described above.

```
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;
        }
    }
```

Flinders University

```
    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;
    }

    return 0;
}
```

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 have already seen. When each word is read, check to see if it is already in the vector; if it is 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 vector class. You will need to write your own class to keep track of the list of words. At this level, you can assume that the test case input will be no more than 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 MyVector {
  public:
    typedef T* iterator; // creates an alias of the T* type called iterator
    MyVector();

    iterator begin();
    iterator end();
    int size();
    iterator insert (iterator position, const T& item);

  private:
    T items[1000];
    int used;
};
```

Flinders University

For this practical, you may wish to implement your class as a *template class*, which is defined entirely in the header file without an accompanying **.cpp** file. In this approach, the complete method is written within the class scope as a single unit. Given the class is likely to be small, you could also opt to place the class at the top of your **words.cpp** file. If you do elect to create a separate class file, ensure you add it to the repository. CLion should prompt you to do this when creating the class file(s), so just click Add.

### 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

```
./words -i // or words.exe -i (the -i can also be specified as an argument within CLion)
the quick brown fox jumps over the lazy dog
```

should result in the output shown below.

```
brown: 1
dog: 1
fox: 1
jumps: 1
lazy: 1
over: 1
quick: 1
the: 2
```

Your program will need to store two pieces of information for each word. Define a struct called **WordInfo** that contains a **string**, named **text**, for the word's text, and an **int**, named **count**, for the number of times it has 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 is 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 is 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 do not know how large the collection of words might become, you will need to make your 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 is still enough space. When the space runs out, allocate a new block that is twice as large, copy all of the old values into the new space, and delete the old block.

You can test large text input by copying and pasting form a test file or alternatively using file redirection if you are on a Unix-based machine (Linux or macOS). The latter can be achieved by running the program from the command line and redirecting the contents of your test file as follows:

```
./words < test.txt
Total: 1234
```

Flinders University