CMPS2212 GUI Test1 and Test2 4 March 2020

You are going to continue building on the existing Notepad application. In particular, you will be adding a feature that allows a user to draw graphs based on different options based on the words that appear within the currently open text file.

The first step is to add a new menu option called **Document Properties**. When selected, launch a QTabbedWidget with four tabs labelled as follows: **Word\_Info**, **Character\_Info**, **Miscellaneous**, and **Graph**. Inside of the of the **Word\_Info** tab, add a label that says **Word Count**. Next to it, add a combo box with two options: **All Words**, and **Unique Words**. Next, add another label that will show the result of the selected option. Inside the **Character\_Info** tab, do the same as the **Word\_Info** tab but this time your combo box will have the following two options: **Total Characters (excluding spaces)**, **Total Characters (including spaces)**. The **Miscellaneous** tab will have a label that says **Paragraph Count** and will show the result next to it within another label. Below those two labels you will add a third label that says **Line Count** and next to it show the resulting value within a fourth label. The **Graph** tab will be explained later in the document.

You will need to create a class called **Word**. This class will have three data members, **text** of type QString, **frequency** of type int, and **length** of type int. You will need to implement the following functions:

Word(QString t);//the constructor will build a Word object based on t and

//initialize count to 1

Word(QString t, int length);

void setFrequency(int i);

int getFrequency() const;

void setLength(int i);

int getLength() const;

QString getText() const;

void operator++();

bool operator<(Word & w);

bool operator>(Word & w);

bool operator==(Word & w);

friend ostream& operator<<(ostream & out, const Word & w);

You will also need to overload << to work with QDebug and with the fstream operator. The code for both functions will look very similar to what you do when overloading the ostream operator.

The next step is to test your **Word** class to ensure that it works.

Once the **Word** class works, you will then create a QList object to store the **Word** objects. Do something similar to **QList<Word> wordlist;**

You can use a different data structure if you want. You will then need to traverse the text file that is currently open. As you read a word, create a **Word** object by passing the newly read word to the **Word** constructor. You need to determine the length of the Word and call that Word’s **setLength()** function. Alternatively, you can use the overloaded Word constructor and pass in both the QString for the Word and its length. Next, traverse the QList and compare the **Word** object you just created with each **Word** object currently in the list. Recall that you have overloaded the comparison operators so this should be easy. If you find a **Word** object in your list that matches the one just built, simply bump up the ***count*** for the **Word** object in the list. On the other hand, if you traverse the entire QList and do not find a matching **Word** object, that means that this is the first time you are coming across this word and should instead insert the **Word** object into the list.

At this point, you should have a QList of only unique **Word** objects with their corresponding frequencies. The next step is to sort the QList. This can be done many ways. I will introduce you to function **sort** which takes a QList and sorts it automatically (assuming you have overloaded the **<** and **>** operators. The **sort** function is overloaded but we will use the simplest version show below:

**std::sort(wordList.begin(), wordList.end());**

The other approach is to call function **sort** and pass in a separate static function that you create within the Word class that describes how to compare two Word objects. This approach assumes that the Word class has not overloaded the **<** and **>** operators and so it must rely on an external function that shows the compiler how Word objects are compared. You might find the second version of sort to be particularly useful in performing one or more of your options.

**std::sort(words.begin(), words.end(), compare);**

Below I have provided some code that you can put in your program to test whether things are working to this point:

Word w1("hello");

Word w2 ("mules");

Word w3 ("help");

++w1;

++w1;

++w3;

QList<Word> words;

words<<w1<<w2<<w3;

qDebug()<<(words.at(1));

qDebug()<<w1<<w2<<w3;

if(w1>w2)

qDebug()<<"greater";

if(w1==w2)

qDebug()<<"equal";

if(w1==w1)

qDebug()<<"1equal";

qDebug()<<w1.getText()<<w1.getCount();

// Let's check the content of the list

qDebug() << words;

// sort

std::sort(words.begin(), words.end(), compare);

//std::sort(words.begin(), words.end());

// check result

qDebug() << words;

Expected Output:

"mules" 1

"hello" 3 "mules" 1 "help" 2

Should print - greater

Should print - equal

"hello" 3

("hello"3, "mules"1, "help"2)

("mules"1, "help"2, "hello"3)

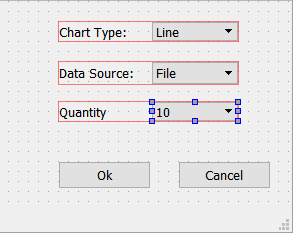
**PART 2**

1. Add a new menu option called **Statistics**
   1. Create a sub-menu called **generate** **statistical file**
   2. The statistical file must be named the same as the current document that is open but must append to its name an underscore followed by the word, “**statistics**”. The file extension should be .**sta**. So, if you currently have open a text file called **cow.txt**, the statistics file generated must be named **cow\_statistics.sta**. If no document is currently open, then notify the user and return. Once the file is generated, notify the user that the operation was successfully completed.
2. The contents of the statistical file will be as follows:

|  |
| --- |
| Statistical Data for “cow.txt”  Generated Date: (insert current date)  Total Word Count: (insert total word count)  Unique Word Count: (insert unique word count)  Word Frequency:  Hello 109  Is 107  The 99  A 99  Are 99  It 85 |

The words listed in the file should be done in descending order (highest frequency to lowest)

1. In the **Graph** tab you created within the Tabbed Widget, you must display the following dialog shown below. The dialog will look similar to what is shown below:

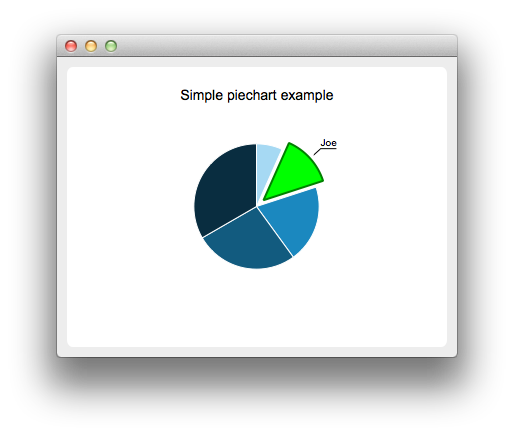


**Chart type** will have three options: **line, bar and pie**.

**Data Source** will have two options: **File** and **Auto Generate**

**Quantity** will have three options: **Show Top 10**, **Show Top 20** and **Show All Words**.

1. If the user chooses **Auto Generate**, you must ensure that a document has been loaded in your Notepad or there is text in the editor from which to generate the chart. Otherwise, generate an error message. In this step, you can refer to the example code for building bar charts that was provided to you. You will need to traverse the QList you built earlier which is now sorted in ascending order. So rather than read 10 Word objects starting from the front of the QList, start reading them from the back of the QList. As you grab a Word object from the QList, use its text data member to initialize the constructor of a QBarSet. For example, if the text portion of the first Word object is “the”, then create a QBarSet and pass “the” to its constructor. Next, using function append, append the current Word object’s count data member to the QBarSet just created. Do this 10 times for all 10 Word objects.
2. Using the code from the bar chart example, you should now be able to tweak things to make the chart display. If the user chooses a pie chart, ensure that it shows the word with the highest frequency as a separate slice, similar to the example below:



1. If the user chooses their **Data Source** to be a file, you must launch the **File Dialog** and have them search for the file. Your dialog should be done so that only files with an extension of **.sta** are listed. If you recall, the user can generate **.sta** files when doing step2. Furthermore, once a file has been selected, you will then, based upon the other options selected, generate the specified chart with the specified word amounts.
2. You will need to overload the **>> operator** in the Word class so that you read an entire Word object per line.
3. If the text file containing the statistics is empty for whatever reason, you must generate an error explaining why a chart could not be generated.

Please let me know if anything is incorrect or unclear. Again, because this is a test and because I have given you 90% of the algorithm, you must complete it on your own.

I have also provided two text files which you can use to test your code. There is also a Website, <https://wordcounttools.com/> that allows you to submit a file or paste text and have it provide statistics concerning the document such as word count etc.