School of Computing, Edinburgh Napier University

Assessment Brief Pro Forma

1. Module number	SET07109
2. Module title	Programming Fundamentals
3. Module leader	Simon Powers
4. Tutor with responsibility for this Assessment Student's first point of contact	Simon Powers S.Powers@napier.ac.uk
5. Assessment	Practical Skills Assessment 2
6. Weighting	36% of module assessment
7. Size and/or time limits for assessment	You should spend approximately 36 hours working on this assessment.
8. Deadline of submission	07/04/2019 at 15:00 Your attention is drawn to the penalties for late submissions.
9. Arrangements for submission	Submit to the Moodle Dropbox by 15:00 on Sunday 7th April. You are advised to keep your own copy of the assessment.
10. Assessment Regulations	All assessments are subject to the University Regulations.
11. The requirements for the assessment	See attached specification.
12. Special instructions	See attached specification.
13. Return of work	Initial feedback will be provided at the demonstration session. Additional feedback and marks will be returned via Moodle.
14. Assessment criteria	See attached specification.

SET07109 Programming Fundamentals
Coursework 2

1 Overview

This is the second coursework for SET07109 Programming Fundamentals. This coursework is worth **36%** of the module mark.

This coursework contains two parts. In Part A you will build a library class that implements a binary search tree data structure for storing `string` data for efficient searching. This is an extension of the binary search tree case study presented at the end of Chapter 6 in the Workbook. You should carefully study pages 151-157 of the Workbook for the definition of a binary search tree and necessary background information. In Part B you will use your binary search tree class to implement an efficient program to count the frequency of each word in a text file. The marking scheme is shown at the end of this document.

2 Part A: The binary search tree library

You will be provided with four files – two header files, a supporting C++ file for the implementation of the `BinarySearchTree` class, and a test file to test the `BinarySearchTree` class. **Your task is to complete the implementation of the methods and operator overloads in the `BinarySearchTree` class, as specified in Tables 1 and 2, so that it passes all of the tests in the `test.cpp` file. You must not modify the `test.cpp` file**. However, you are free to add additional methods to the `BinarySearchTree` class, including modifying the header files and adding additional supporting `.cpp files`, so long as you implement the functionality in Tables 1 and 2 and pass the tests in the unmodified `test.cpp` file.

Note that strings in C++ can be compared using equality and inequality operators (e.g. $<$ and $>$) to obtain their order, e.g. "abc" $<$ "def" returns

Method	**Description**
`BinarySearchTree()`	Creates an empty binary search tree
`BinarySearchTree(std::string word)`	Creates a binary search tree with an initial word to store.
`BinarySearchTree(const BinarySearchTree &rhs)`	Creates a binary search tree by copying an existing binary search tree. This must be a deep copy, not a reference.
`BinarySearchTree(vector<std::string> &words)`	Creates a binary search tree from a `vector` of words.
`~BinarySearchTree()`	Destructor for a binary search tree. Releases the memory occupied by all of its nodes.
`void insert(std::string word)`	Adds a word to the binary search tree. If the word already exists in the tree then nothing happens.
`bool exists(std::string word)`	Returns true if the word is in the tree, false otherwise.
`std::string inorder()`	Generates a string containing the words in the tree in alphabetic order.
`std::string preorder()`	Generates a string containing the words in the tree in *pre-order* fashion.
`std::string postorder()`	Generates a string containing the words in the tree in *post-order* fashion.

Table 1: List of Methods for the `BinarySearchTree` class

Operator	Example	Description
+	`tree = tree + ‘‘hello’’`	Inserts a new word into the binary tree.
=	`tree1 = tree2`	Copy assignment operator. Deletes the nodes in `tree1`, freeing memory, and then makes deep copies of all of the nodes in `tree2`.

Table 2: List of Operator Overloads for the `BinarySearchTree` class

true. All of the printing functions should return the words in the tree separated by a space (e.g. “for he she”).

You will need to ensure that your compiler is operating to the C++ 2011 standard for the coursework files to compile. On the Microsoft compiler this is automatic. With other compilers you may need to pass a command line argument. For example, in Clang you would use `clang++ -std=c++11 filename.cpp`.

The `BinarySearchTree` class should be built as a library file, allowing it to be reused in other applications.

3 Part B: Using your library to implement a word counter application

Your task is to write an application that reads in a text file and outputs to the console the number of times that each word in that file occurs.

For example, if the input text file contained “The C programming language is very useful. The C programming language is a low level language”, then the output would be:

```
The: 2
C: 2
programming: 2
language: 3
is: 2
very: 1
useful: 1
a: 1
low: 1
level: 1
```

The counts should ignore punctuation, e.g. “language”, “language,”, “language?” and “language.” should all be treated as “language”. The order in which the words and their counts are printed in the output does not matter.

You must use your binary search tree to do this. You should add an `int` variable to the `Node` struct called `count`. Note that you do not need to change any of the methods and operators that you have written for Part A. Your application should then use the following procedure to count words:

1. Make a binary search tree object.
2. Extract a word from the input file.
3. Check if that word exists in the binary search tree. If not, add it to the tree and set the `count` field of the `Node` to 1. If the word already exists in the tree, add one to the `count` field of the `Node` in which it occurs.

You may add one or more public methods to the `BinarySearchTree` class to achieve this. The application must check the entire input file. You may assume that, apart from upper and lower case letters, the only characters the input will contain are commas, full stops, new lines, question marks, spaces, and tabs. A word is defined as a continuous sequence of upper and lower case letters only. The counting should be case sensitive, i.e. “Language” and “language” should be treated as different words.

You are supplied with two input files that we will use to test your application: `single_words_test.txt`, and `sentences_test.txt`. The first contains a single word on each line, the second contains complete sentences (the text in these files is taken from the C Programming Wikibook located here: `https://en.wikibooks.org/wiki/C_Programming/Why_learn_C%3F`).

4 Restrictions and Resources

Your application must be written entirely in C++ using the C and C++ standard library. The use of non-standard libraries is not permitted.

5 Code quality

Code quality includes meaningful variable names, use of a consistent style for variable names, correct indentation, and suitable comments. Variable names must begin with a lower case letter.

The source code should have a comment at the top detailing the name of the author, the date of last modification, and the purpose of the program. Every function should be preceded by a comment describing its purpose, and the meaning of any parameters that it accepts. Any complex sections of code should be commented.

You must include a *makefile* to build the application. Your *makefile* must include a clean target to delete `.exe`, `.lib` and `.obj` files.

Finally, because C++ is a low level language, remember to free any memory that you dynamically allocate, and to close any files that you open.

6 Submission

Your code must be submitted to the Moodle Assignment Dropbox by 15:00 on Sunday 7th of April. If your submit late without prior authorisation from your personal tutor your grade will be capped at 40%.

The submission must be your own work, written by you for this module. Collusion is not permitted. The university regulations define collusion as

> "Conspiring or working together with (an)other(s) on a piece of work that you are expected to produce independently."

Please acknowledge all sources of help and material through comments in the source code giving a link to the material that you have consulted, e.g. include a URL to any Stack Overflow code segments that you have incorporated in your work. If you use *Git* or other version control then please make sure that your coursework is stored in a private repository to prevent access by others. **If it is suspected that your submission is not your own work then you will be referred to the Academic Conduct Officer for investigation**.

All coursework must be demonstrated during the week beginning 7th April. If the coursework is not demonstrated to a member of the teaching team this will result in a grade of 0. The demonstration session is the point where the teaching team will give you verbal feedback on your work. Final grades, and additional feedback, will be returned via Moodle.

The submission to Moodle must take the form of the following:

- The code file(s) required to build your binary search tree library and word counter.
- The supplied test file.

- A code file to demonstrate your word counter.
- A `makefile` to allow building of your application. The `makefile` should build the binary search tree as a separate library, and also allow building of the supplied test file and your word counter against this library. It should contain targets to run the supplied test file, and targets to run your word counter separately on the `single_words_text.txt` and `sentences_test.txt` files.
- A *readme* file indicating how the `makefile` is used to build your library and word counter, and the toolchain used to build it (i.e. Microsoft Compiler, Clang, etc.)

These files must be bundled together into a single archive (a zip file) using your matriculation number as a filename. For example, if your matriculation number is *1234* then your file should be called *1234.zip*. All submissions must be uploaded to Moodle by the time indicated.

BE WARNED! This coursework requires an understanding of the material covered in units 5 to 9 of the module. If you do not complete these you will struggle. DO NOT LEAVE THIS WORK TO THE LAST MINUTE!

If you have any queries please contact the module leader as a matter of urgency. This coursework is designed to be challenging and will require you to spend time developing the solution.

7 Marking Scheme

The coursework marks will be divided as follows:

Description	**Marks**
Single word constructor	1
Vector constructor	2
Copy constructor	2
Insert method	2
Exists method	2
Different printing methods	4
+ operator	1
Assignment operator (=)	3
Part B: Binary search tree object used to store words and their counts	4
Part B: Word counts correct on single words test file	2
Part B: Word counts correct on sentences test file	6
Dynamically allocated memory freed correctly	2
Code quality	2
Makefile works and builds the binary search tree class as a library	2
Submission zip folder complete and correctly collated	1
Total	36