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	03/05/2021 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 Monday 3rd May. Demonstration in your practical slot between 4th May and 14th May. **Your work cannot be marked without a demonstration.** 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	Feedback will be provided at the demonstration session. Final marks will be returned via Moodle within three weeks. Additional one-to-one feedback will be given to any student that requests it.

14. Assessment criteria	See attached specification.

SET07109 Programming Fundamentals Coursework 2: Implementing a compiler symbol table

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 the first part you will build a command line application, called `symbol_table`, that functions as a compiler symbol table. This will read in a `.c` file containing a C language program, and will record information about each identifier (variable name or function name) in that program. The specification below goes into detail about the information that your application should record about each identifier and what should be output. In the second part of the coursework you will improve the performance of your application by creating a binary search tree class to store the identifiers and their associated information. This builds on the binary tree case study in Chapter 6 of the Workbook.

Your application and binary search tree class must be written in C++, and may only use prewritten functions from the C and C++ standard libraries. You may find a list of the functions in the C and C++ standard libraries here: `http://www.cplusplus.com/reference/`. No prewritten functions from other sources may be used, but you can (and should) split your application up into your own functions and classes. You should consult the Module Leader if you have any doubt as to what is permitted.

2 Part A: The symbol table command line application

Your application should be invoked as follows on the command line:

`symbol_table` *file.c*

where *file.c* can be replaced by the name of a file containing source code for a C program. Your application should read in the `.c` file and output the following information about it to the console:

1. The number of variables declared in the file (remember that a variable called `x` would be declared as follows: *type* `x`, where *type* would be the name of a primitive type (char, short, int, long, long long, float, double), the name of a struct, or a pointer type. Variables include those declared as function parameters, and those that are arrays.

2. The number of functions declared in the file (you should exclude those declared in header files – only consider those declared in the `.c` file itself). Note that a function declaration can be distinguished from a variable declaration because the name will be followed by a (in a function declaration.

3. The number of if statements in the file.

4. The number of for loops in the file.

5. The number of while loops in the file.

In addition, your application should produce an output file called `identifiers.txt` that contains a symbol table. This table should contain one row for each identifier (variable name or function name) declared in the `.c` file. The columns of the table should be the following:

1. Identifier name.

2. Line number in the file on which the identifier is declared.

3. Whether the identifier is a variable, array, or function.

4. The type of the identifier (variable or array type, or return type of a function).

5. The number of times that the identifier is referenced in the file (i.e. the number of times in which its name occurs), excluding the time in which the identifier is declared (created).

You may separate each column with a comma, and each row should start on a new line. You should pay attention to the function scope of variables, i.e. two different variables may have the same name if they are declared in different functions. **You do not need to consider different levels of scope within a function for this coursework**, i.e. you can assume that within a single function identifier names will be unique. Variables with the same name but different function scopes should have separate rows in the symbol table. **You are provided with four test files on Moodle to test your application, and which you will be asked to run your program on during your demonstration**.

You should decide how to store your symbol table. One suggestion is that you make a struct to store all of the above data about each identifier (i.e. one row of the table). You may then store instances of this struct in a vector. Alternatively, you may use another approach if you prefer. You are encouraged to discuss your approach with the teaching team before implementing it.

You will need to tokenise the `.c` file, i.e. split it into words of the C language (keywords, identifiers, operators etc. Chapter 5 of the Workbook covers how to read in lines of a text file in C++. You should discuss your approach to tokenisation with the teaching team in your practical session. *For simplicity you may assume that every identifier name will be preceded and followed by a space*, e.g. `int x = 5;`. This assumption will make tokenisation easier.

3 Part B: Enhancing performance with a binary search tree

This second part of the coursework builds on the binary tree case study in Section 6.17 of the Workbook, and should be viewed as an extension to your application, i.e. you should complete Part A first. In the Workbook case study, the nodes of the tree stored integers. Here, each node should store one row from the symbol table (a node may, for example, store the identifier struct that you created in Part A). The tree should store the nodes so that they are sorted by the names of the identifiers, i.e. every identifier to the left of the root node should have a name that comes alphabetically before the root, and every identifier to the right should have a name that comes alphabetically after it (it is this property of storing the identifiers in sorted order that makes searching for an identifier much faster than in an unsorted array or vector).

To implement this, you should produce a binary search tree class that stores a pointer to the root node of the tree as a private variable. Your class should contain a public `update` method that takes in an identifier as a parameter and updates the symbol table with that identifier. This will involve creating a new node in the correct place in the tree if the identifier is not already in the symbol table, otherwise it should increment the count of the number of times an existing identifier is referenced. It should also contain a public method to print the symbol table from the tree. Finally, because your `update` method will need to make new nodes on the heap, your binary search tree class should contain a destructor to free all memory by calling delete on all of the nodes when the object is destroyed (there is pseudocode for this in the Workbook). Alternatively, you may use smart pointers. You may add further private helper methods to your binary search tree class as required. Your binary search tree class should be built as a library and linked to your `symbol_table` application from Part A (see Chapter 4 of the Workbook for building library files). You should make appropriate use of separate header and `.cpp` files for this.

4 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 of every file 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.

Properties of objects should be made private and accessed through public methods.

You should make use of C++ language features where appropriate, e.g. strings, vectors.

You must include a *makefile* with targets to build Part A and Part B. It must also contain a target to build your binary search tree class as a library file, which your target to build Part B should use. Your *makefile* must also include a clean target to delete executable, library and object files.

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

5 Submission

Your code must be submitted to the Moodle Assignment Dropbox by 15:00 on Monday 3rd May. If you submit late without prior authorisation from your Personal Development Tutor your grade will be capped at 40%.

You must demonstrate your work during your practical session on the week beginning 3rd May or the week beginning 10th May. You must be available at the practical times during these two weeks. If you do not demonstrate your work then you will receive a mark of 0. The demonstration session is the point where the teaching team will give you verbal feedback on your work. Final grades will be returned via Moodle within three weeks of the hand-in date.

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

- The code files required to build your application for both Part A and Part B.
- A `makefile` to allow building of your application. It should contain separate targets for Part A and Part B. The `makefile` should build your binary search tree class as a library, and your target for Part B should link your application to the library.
- A *readme* file indicating how the `makefile` is used to build your application, the toolchain used to build it (i.e. Microsoft Compiler, Clang, etc.), and a brief description of what your application does and how to build it.

These files must be bundled together into a single archive (a zip file) using your matriculation number as the 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.

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. **You must not copy and paste code from other sources (you will not get the marks for this). If you consult**

other sources you must adapt and understand the code to make it your own, and be able to explain it during your demonstration. 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**.

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**.

6 Marking Scheme

The coursework marks will be divided as follows:

Description	**Marks**
Number of variables declared output	2
Number of functions declared output	2
Number of if statements, for loops and while loops output	2
Symbol table correct for names, line numbers and types	4
Symbol table correct for number of times identifiers referenced	4
Part B: Binary search tree created as a library class	4
Part B: Print symbol table method correctly implemented	2
Part B: Update method correctly implemented	4
Part B: Memory freed correctly in binary search tree class	2
Part B: `symbol_table` application works correctly with binary search tree	4
Code quality	4
Makefile to specification and works correctly	1
Submission correctly collated including readme	1
Total	36