2 Assignment 0 [Assignment ID: `cpp_tools`]

2.1 Preamble (Please Read Carefully)

Before starting work on this assignment, it is **critically important** that you **carefully** read Section 1 (titled "General Information") which starts on page 1-1 of this document.

2.2 Topics Covered

This assignment covers material primarily related to the following: software tools, C++ compiler, linker, CMake, Git.

2.3 Exercises

1. In this exercise, you will be creating a "remote" repository and then simulating the effects of two different users working with this repository by utilizing two (local) cloned versions of the repository. To begin, select an empty directory for use in this exercise. This directory will be henceforth denoted `$TOP_DIR`. In this exercise, you need to perform the tasks listed below (in order). You should assume that the two simulated users involved in this exercise do not know about the changes that each other is making. This implies that a user must attempt a push operation (and have it fail) before they can know that their local copy of the repository is out of date with respect to the remote.

   (a) Create an empty bare Git repository in the directory `$TOP_DIR/remote_repo.git` to be used as a remote in this exercise. This can be accomplished by typing:

   ```
   cd $TOP_DIR
   git init --bare remote_repo.git
   ```

   (b) User 1. Clone the repository `$TOP_DIR/remote_repo.git` to the directory `$TOP_DIR/repo_1`. In the top-level directory of the repository `$TOP_DIR/repo_1`, create a file `words.txt` with the following three lines of text:

   ```
   Delta
   Echo
   Foxtrot
   ```

   Propagate the changes to the remote repository `$TOP_DIR/remote_repo.git`. [Hint: You will need to use the following commands in order: `git clone`, `git add`, `git commit`, and `git push`.]

   (c) User 2. Clone the repository `$TOP_DIR/remote_repo.git` to the directory `$TOP_DIR/repo_2`. In the repository `$TOP_DIR/repo_2`, modify the file `words.txt` by inserting the following three lines prior to the first line in the file:

   ```
   Alpha
   Bravo
   Charlie
   ```

   Propagate the changes to the remote repository `$TOP_DIR/remote_repo.git`. [Hint: You will need to use the following commands in order: `git clone`, `git add`, `git commit`, and `git push`.]

   (d) User 1. In the repository `$TOP_DIR/repo_1`, perform the following. Append the following lines to the file `words.txt`:

   ```
   Golf
   Hotel
   ```

   Propagate the changes to the remote repository `$TOP_DIR/remote_repo.git`. [Hint: You will need to use the following commands (some possibly more than once): `git add`, `git commit`, `git push`, and `git pull`. Alternatively, `git fetch` and `git merge` can be used in place of `git pull`.]

   (e) User 2. In the repository `$TOP_DIR/repo_2`, append the following line to the file `words.txt`:

   ```
   Zulu
   ```

If any merge conflicts arise, keep only the changes from user 1. Propagate the changes to the remote repository `$TOP_DIR/remote_repo.git`. [Hint: You will need to use the following commands (some possibly more than once): `git add`, `git commit`, `git push`, and `git pull`. Alternatively, `git fetch` and `git merge` can be used in place of `git pull`.]

(f) Check that the repository `$TOP_DIR/remote_repo.git` now contains the desired contents. In particular, the repository should contain the file `words.txt` with the following contents:

```
Alpha
Bravo
Charlie
Delta
Echo
Foxtrot
Golf
Hotel
```

[Hint: You will need to use the command `git clone`. Simply clone the remote repository and examine the contents of the cloned version.]

2. The `fibonacci` project. Write a CMakeLists file that can be used to build the `fibonacci` project in the directory cmake/exercises/fibonacci (in the Git repository for the C++ lecture slides, the URL of which can be found in Section 1.10). The `fibonacci` project consists of a single program `fibonacci`, which is comprised of the source-code files `main.cpp`, `fibonacci.cpp`, and `fibonacci.hpp`. The only library needed for this program is the C++ standard library. After building the project, check to ensure that the `fibonacci` program can be run successfully. (Do not assume that all is well simply because the program compiles and links.)

3. The `hg2g` project. Write a CMakeLists file that can be used to build the `hg2g` project in the directory cmake/exercises/hg2g (in the Git repository for the C++ lecture slides, the URL of which can be found in Section 1.10). The `hg2g` project consists of a single program `answer`, which is comprised of the source-code files `src/answer.cpp`, `lib/question.cpp`, and `lib/answer.cpp`. The code has a single header file stored under the directory `include` (which will need to be specified as an include directory when building the code). The only library needed for this program is the C++ standard library. After building the project, check to ensure that the `answer` program can be run successfully. (Do not assume that all is well simply because the program compiles and links.) Note that, when run, the `answer` program will report that an exception has been caught and the program is terminating.

4. The `buggy` project. Write a CMakeLists file that can be used to build the `buggy` project in the directory cmake/exercises/buggy (in the Git repository for the C++ lecture slides, the URL of which can be found in Section 1.10). The `buggy` project consists of a single program `buggy`, which is comprised of the single source-code file `buggy.cpp`. The only library needed for this program is the C++ standard library.

   The CMakeLists file must provide two options:

   (a) `ENABLE_ASAN`. This option has a boolean value, which defaults to `false`, indicating if the Address Sanitizer (ASan) should be enabled.
   (b) `ENABLE_UBSAN`. This option has a boolean value, which defaults to `false`, indicating if the Undefined Behavior Sanitizer (UBSan) should be enabled.

   The recommended approach to this exercise is to modify the `CMAKE_CXX_FLAGS` and `CMAKE_EXE_LINKER_FLAGS` variables as appropriate, depending on the values of `ENABLE_ASAN` and `ENABLE_UBSAN`. This modification should be performed by appending to the current values of these variables, since completely discarding the existing values would lose potentially important settings. (A code-profiling example given in the CMake section of the C++ lecture slides may be helpful for illustrating how to modify the above variables.) For the purposes of this exercise, you may assume the following:

   - ASan requires the use of the "`-fsanitize=address`" flag for both the compiler and linker.
   - UBSan requires the use of the "`-fsanitize=undefined`" flag for both the compiler and linker.

The above assumptions are valid for the GCC and Clang compilers. It is strongly recommended that the options be defined in a separate file called `Sanitizers.cmake` and then included in the `CMakeLists.txt` file with an `include` command. The file `Sanitizers.cmake` will then be easily reusable in later assignments.

After building the project with ASan and UBSan enabled, run the `buggy` program. This program should fail due to a problem detected by the code sanitizers.

5. The `boost_timer` project. Write a CMakeLists file that can be used to build the `boost_timer` project in the directory cmake/exercises/boost_timer (in the Git repository for the C++ lecture slides). This project consists of a single program `timer`. The `timer` program consists of the source-code files `timer.cpp`, `fibonacci.hpp`, and `fibonacci.cpp`. This program needs to use the `timer` component of the Boost library in addition to the C++ standard library. The find module for `Boost` sets numerous variables, of which the following will need to be utilized: `Boost_INCLUDE_DIRS` and `Boost_LIBRARIES`. (The find module for `Boost` also defines an imported target `Boost::timer`. This imported target can be employed as an alternative to using the `Boost_INCLUDE_DIRS` and `Boost_LIBRARIES` variables.) You can disable multithreading support in Boost by setting `Boost_USE_MULTITHREADED` to `false` before locating the `Boost` package with `find_package`. After building the project, check to ensure that the `timer` program can be run successfully. (Do not assume that all is well simply because the program compiles and links.)

6. The `cgal_in_circle` project. Write a CMakeLists file that can be used to build the `cgal_in_circle` project in the directory cmake/exercises/cgal_in_circle (in the Git repository for the C++ lecture slides). This project consists of two programs: `in_circle` and `in_sphere`. The `in_circle` program is comprised of the source-code files `in_circle.cpp`, `utility.cpp`, and `utility.hpp`. The `in_sphere` program is comprised of the source-code files `in_sphere.cpp`, `utility.cpp`, and `utility.hpp`. In addition to the C++ standard library, these programs require the CGAL library. The find module for `CGAL` initializes several variables, of which the following will need to be utilized: `CGAL_INCLUDE_DIRS`, `CGAL_LIBRARY`, and `GMP_LIBRARIES`. On some systems using the GCC compiler, it is necessary to compile the programs in this project with the `-frounding-math` option. If this option is not used when it is needed, the likely result is that the programs will terminate with a failed assertion when run. After building the project, check to ensure that the `in_circle` and `in_sphere` programs can be run successfully. (Do not assume that all is well simply because the programs compile and link.)

7. The `coverage` project. Write a CMakeLists file that can be used to build the `coverage` project in the directory cmake/exercises/coverage (in the Git repository for the C++ lecture slides). This project consists of a single program called `random`. The `random` program is comprised of the single source-code file `random.cpp`. The only library needed for this program is the C++ standard library. The project also includes a Bash script called `run_tests` that runs the `random` program several times, allowing code-coverage information to be collected. The `run_tests` script must be invoked with a single argument that corresponds to the CMake binary directory (i.e., the directory that will contain the executable program file `random` after the code is built).

   The CMakeLists file must provide an option named `ENABLE_COVERAGE`. This option has a boolean value, which defaults to `false`, indicating if code-coverage analysis using Lcov should be enabled. If this option is enabled, a target called `coverage` should be provided that runs the `random` program several times using the script `run_tests` and then generates code-coverage information for those runs using Lcov. The provided CMake module `CodeCoverage.cmake` should be used (without modification) in order to simplify this exercise. In particular, the `setup_target_for_coverage_lcov` and `append_coverage_compiler_flags` commands defined in this module will be needed. (For more details, see the example of using Lcov with CMake in the Lcov section of the C++ lecture slides.) Building the target `coverage` should result in HTML output (from `genhtml`) being placed in the directory `coverage` in the CMake binary directory.

   After building the `coverage` target, use your web browser to view the HTML file `index.html` in the `coverage` directory (under the CMake binary directory). By examining this HTML document, determine which lines of code in `random.cpp` were not executed when the `random` program was run by the `run_tests` script.

8. Prepare the Git repository for the submission of this assignment. This repository is to contain the work completed in Exercises 2, 3, 4, 5, 6, and 7, organized in the manner described below. (The work for Exercise 1 is not to

be submitted.) The preparation of this repository is most easily accomplished by cloning your empty assignment repository from GitHub, adding content to the resulting local repository, and then pushing the changes from the local repository to the repository on GitHub. The top-level directory of the repository should contain the following:

(a) a file `IDENTIFICATION.txt`, which provides student and assignment information (as described in Section 1.3).

(b) a directory `fibonacci`, which contains the CMakeLists and source-code files for the `fibonacci` project developed above (in Exercise 2).

(c) a directory `hg2g`, which contains the CMakeLists and source-code files for the `hg2g` project developed above (in Exercise 3).

(d) a directory `buggy`, which contains the CMakeLists and source-code files for the `buggy` project developed above (in Exercise 4).

(e) a directory `boost_timer`, which contains the CMakeLists and source-code files for the `boost_timer` project developed above (in Exercise 5).

(f) a directory `cgal_in_circle`, which contains the CMakeLists and source-code files for the `cgal_in_circle` project developed above (in Exercise 6).

(g) a directory `coverage`, which contains the various files for the `coverage` project developed above (in Exercise 7) (i.e., CMake build files, source-code files, and test scripts).

After having prepared the repository, run the `assignment_precheck` program on the repository to confirm that the precheck passes.

9. In the local repository created in the previous exercise (i.e., Exercise 8), create a new branch called `experimental` and then checkout this branch. In this new branch, introduce various combinations of errors into the files (e.g., by adding or deleting random characters). Push the resulting buggy version of the files to the `experimental` branch of the remote repository. Then, run the `assignment_precheck` program on the `experimental` branch of the remote repository (using the `-b` option to specify the branch to check). Observe the errors that are obtained. Repeat this process, each time corrupting different files and observing what type of errors result. This exercise is intended to help you better understand what various types of errors mean so that you can more effectively troubleshoot problems in later programming assignments. When doing this exercise, be careful not to accidentally push any of the buggy files to the `master` branch.