Project 2

Due: Mar 1, before midnight.

Important Reminder: As per the course Academic Honesty Policy, cheating of any kind will minimally result in your letter grade for the entire course being reduced by one level.

This document first provides the aims of this project. It then describes the requirements as explicitly as possible. It lists the files with which you have been provided. Finally, it hints at how these requirements can be met.

Aims

The aims of this project are as follows:

- To expose you to memory allocation.
- To give you some familiarity with programming in a low-level language.

Requirements

Implement a program which can add together some number of pairs of vectors having arbitrary size. The program must be implemented in either C, C++ or Rust using the `gcc`, `g++` or `cargo`/`rustc` toolchains available on `remote.cs`. Specifically, you must submit a zip file which will unpack into a `prj2-sol` directory which will minimally contain two shell scripts:

`prj2-sol/make.sh`
: Running this script from **any** directory should build your program.

`prj2-sol/run.sh`
: When run from **any** directory, this program should run the program built by `make.sh`. It should require two positive integer arguments:

  `N_OPS`
  : The number of additions to be performed.

  `N_ENTRIES`
  : The number of entries in each vector.

  The program should read `N_OPS` pairs of vectors from standard input. Each vector is specified as `N_ENTRIES` numbers separated by whitespace. After each pair of vectors is read, the program should write the pairwise sum of the two vectors on standard output as follows:

  - Each entry in the sum vector is followed by a single space `' '` character.
  - The output for each sum vector is followed by two newline `'\n'` characters.

  The program should terminate after performing `N_OPS` of these additions.

  [It follows that the program will read a total of `N_OPS * 2 * N_ENTRIES` from standard input and will write out a total `2 * N_OPS` lines to standard output with each non-empty line containing `N_ENTRIES` numbers, each followed by a single space `' '`].

Your program is subject to the following implementation restrictions:

- Your program **must** allocate memory for the two addend vectors and the sum vector on the heap. (The use of C's variable-length arrays VLAs is not acceptable).
- All memory dynamically allocated by your code must be explicitly released before program termination.
- The sum must first be computed into the sum vector before being written to standard output.
- Your program should not assume any maximum limits on `N_ENTRIES` beyond those dictated by available memory.

There is no requirement that your program be 100% robust in reacting to errors, but "reasonable" precautions should be taken. (This relaxation for error handling is primarily motivated by the difficulty in handling formatted input in Rust).

Your submitted zip file should not contain any binaries or garbage files unrelated to the functionality of your project.

You may assume that your project will be run in a standard Linux enviroment having the above tool chains. If in doubt about whether a particular tool will be available, please check with me.

Provided Files

The prj2-sol directory contains:

make.sh and run.sh
Starter files for the shell scripts you are required to submit.

[These are similar to those you were given for your last project, except that they have been hardened against being symlinked.]

README
A README file which must be submitted along with your project. It contains an initial header which you must complete (replace the dummy entries with your name, B-number and email address at which you would like to receive project-related email). After the header you may include any content which you would like read during the grading of your project.

The extras directory contains auxiliary files associated with the project, including files which can help with testing your project.

tests
A directory containing tests. There are two kinds of test files:

`*.test`
A test input for which your program is expected to succeed.

`*.out`
The expected pretty-printed output for a successful test of the corresponding `*.test` file.

do-tests.sh
A shell script which can be used for running the above tests. It can be invoked from any directory and takes up to two arguments:

1. The path to your `run.sh` shell script.
2. An optional argument giving the path to a single test to be run.

If invoked with only a single argument, the script will run all tests. If invoked with a second argument, then it will run only the test specified by that argument.

[Note that this script invokes your `run.sh` shell script with arguments derived from the name of the `*.test` file being run].

LOG

A log file illustrating the operation of the project.

Hints

This section is not prescriptive in that you may choose to ignore it as long as you meet all the project requirements.

You may proceed as follows:

1. Review the material covered in class on Runtime and the matrix multiplication example in particular.
2. Read the project requirements thoroughly.
3. Choose the implementation language for your project. Some considerations:
    - C is a simpler language than either C++ or Rust.
    - Both Rust and C++ are opiniated about how heap memory should be managed; C is not.
    - Rust is the newer language and has had less time to accumulate legacy cruft.
    - The demand for Rust skills may be lower than that for the other two languages, but it is growing fast. In general, all three languages are popular.
4. Carefully study the matrix multiplication code for your chosen implementation language and think about what you need to do to adapt that code to handle this simpler project.
5. Copy over the provided `prj2-sol` directory to your work directory.
6. Copy over files from the matrix multiplication example for your chosen implementation language. Rename files if necessary.
7. Start making changes to the files.
    - First change the project build files to reference a vector addition project (rather than a matrix multiplication project). Change your `make.sh` to run the build process.

      [If editing a `Makefile`, please make sure that all lines which contain shell commands start with a tab character].
    - If the code is based on multiple files (true for the C and C++ matrix multiplication examples), then start with the files corresponding to lower-level code. As you finish with a file, check it for compilations errors using something like `make` *file*`.o`.
8. Ensure that your entire project compiles without **any** warnings.
9. Update your `run.sh` to run your executable. Note that you can use the shell variable `$@` to refer to the arguments your script was invoked with.
10. Test your project via the command line as in the provided LOG.

11. Test your parser using the provided scanner tests:

    ```
    $ ~/cs571/projects/prj2/extras/do-tests.sh run.sh
    ```

    Debug any failing tests. Note that you can run a single test by adding an additional argument to the above command providing the path to the failing test.

12. Iterate until you meet all requirements.

13. Submit as in your previous project (the submission link should be available on gradescope next week). Use the ~/cs571/bin/do-zip.sh script along with a `.zipignore` to ensure that you do not submit binaries or garbage files.

If using git, it is always a good idea to keep committing your project periodically to ensure that you do not accidentally lose work.