Tasks

Write a program that randomly shuffles a string input from `stdin` and outputs the result to a file.

Example Inputs and Outcomes

Upon executing `strshuffle`, I enter the string "shabooyah" and the program prints "byooaashh"... or "yabashooh"... or any anagram of this text to a file that is not hard-coded.

Goals

Through this project, you will become familiar with:

1. basic I/O in C++ (`fstream, iostream`)
2. PRNG in C++ (`cstdlib`)
3. building a C++ program using Makefile

Background

1 File Streams

Bytes from a file are accessed by opening a 'file stream' and reading from or writing to this stream of data. There are three special file streams reserved for command-line output (`stdout`), command-line input (`stdin`), and reporting errors, warnings, and diagnostic information that doesn't go to `stdout` (`stderr`). The first of these two streams are buffered in that they continue to read in/out characters until they encounter a newline character '\n'. The most rudimentary ways to access these file streams are the standard C++ functions `std::cout`, `std::cin`, and `std:cerr` (characters out, characters in, and characters (for?) error). A basic example of a program that accepts input from the `stdin` file stream is below. Note that the input entered for the string cuts off after any whitespace! However this input is still stored in `stdin` and the next word will be provided in the next call to `cin`. In order to capture a full line, `cin.get(); getline(cin, s);` is a more practical alternative to reading full lines from command-line input in C++.

```cpp
#include <iostream>
#include <string>

using namespace std;  // use functions from the std namespace without using
                      // the resolution operation (::) to prefix the call
                      // to that function, e.g., std::cout vs. cout

int main (void) {

    string buf;

    cout << "Enter string: "; // output chars to stdout filestream
    cin >> buf;               // input chars from stdin filestream to buf
    cout << "Your string is: " << buf << endl; // output chars from buf

}
```

A file is opened and accessed by having a reader march along each byte in a file and accessing the byte at that position. The byte at this position may be read out or written to before moving to the byte at the next position. A 'buffered reader' stores a certain amount of the file in a buffer (large array of characters) at one time and provides this entire buffer to the user. The string of characters in this buffer is limited either by 1) the maximum size allowed by the buffer (a large-ish power of 2, e.g., 256, 4096) or 2) the reader encountering a 'delimiter'. A delimiter is a special character, typically the newline character ('\n' or '\r\n' for Windows) for standard file reading operations, that indicates when to terminate a buffer. Another example of a delimiter is the character '\0' that terminates all strings in C and for this reason strings in C are called "null-truncated" or "null-terminated". This is why in C you must allocate one more byte than the length of the string itself in order to have enough space for '\0'!

2 Random Numbers

Have you ever tried to come up with a sequence of random numbers yourself? Try picking a sequence of 10 digits at random. If you repeat this experiment many times, you might find that you prefer certain digits and disfavor others.

Computers face a similar issue: they cannot generate a truly random sequence of numbers from nothing. However, we can leverage a computer's ability to perform arithmetic on very large numbers in very little time to generate sequences of

pseudo-random numbers. Often, a 'seed' is provided to initiate the sequence and every subsequent application of the pseudo-random number generator (PRNG) uses the previous result to obtain the next number in the sequence. See Ch7 of Numerical Recipes – The Art of Scientific Computing 3ed (Press, Teukolsky, Vetterling, Flannery) for more details on PRNG. I have a .pdf copy in the main repository; it is definitely worth checking out!

The C++ standard library (`cstdlib`) comes with a suite of PRNG functions though Numerical Recipes (hereafter NR) claims that "rand and srand [...] are often badly flawed". Since we don't care much for the actual randomness of the shuffle for this project, the standard functions will suffice. A code snippet that uses `srand` and `rand` to generate a sequence of PRNs is included below:

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(void) {

    char c;
    int seed, x;

    /**
     * the seed is 2025 in base 16 (8229 in base 10). you can make it
     * whatever you want; i just happen to think that putting the 0x
     * in front of the number makes it more fun :)
     */
    seed = 0x2025;
    srand(seed);   // seed the PRNG

    cout << "Generate Number? [ENTER to contine]: "; // prompt user
    cin >> noskipws >> c; // get input char from stdin

    while ( c == '\n' ) {
        x = rand();       // generate random int from 0 to 2^32 - 1
        cout << x << endl; // output number
        cout << "Generate Number? [ENTER to continue]: "; // prompt user
        cin >> noskipws >> c; // get input char from stdin
    }
}
```

3 Makefile

Makefiles are files that provide a short-hand way of entering commands used to compile a C or C++ program. While the command `g++ main.cpp -o main` which compiles some program `main` from `main.cpp` is quite brief, the command to build larger codebases can get quite hairy. Remembering to include all of the right source files in the right order, linking the necessary libraries, and then updating this every time the project structure changes can be exceedingly difficult if solely done by directly entering a command each time. This is why utilities like `make` and `cmake` exist to document and maintain the commands we use to build our projects. I've included an example `Makefile` from one of my own projects (written in C) here. Another example is included in this project's repository.

```
CFLAGS = -Wall -Werror -Wextra -O3
LDFLAGS = -lm
CONSTS = src/constants.c
FILES = src/random.c src/linalg.c src/bmr.c src/flow.c src/field.c
    src/init.c src/methods.c src/io.c
DEPS = $(FILES) src/constants.h $(patsubst %.c,%.h, $(FILES))

main: $(CONSTS) $(DEPS)
       gcc $(CFLAGS) $(CONSTS) $(FILES) main.c $(LDFLAGS) -o $@

clean:
       rm -f main
```

To use the `make` utility, you must first create a file called `Makefile`. The default behavior of entering `make` on the command line will be to execute the first 'command' (target) in the file. In this case, it is `main` which compiles all of the source files, halting compilation if any errors or warnings arise, compiles with optimization level 3 (certain set of optimizations made by the gcc compiler), links the C math library, and outputs an executable named `main`. This behavior is achieved either by running `make main` or simply `make`. To delete the executable, enter `make clean`.

In Makefile speak, the 'target' is the name of the word you will use with make to execute the command(s) that follow. The 'prerequisites' or 'dependencies' are the files that follow the target name and colon. In this case, 'main' is the target and 'src/constants.c src/random.c [...] main.c' are the prerequisites, obtained by evaluating the variables CONSTS and FILES in that line. Some more Makefile tips and tricks that I have used are below but you can find must of what you need from either the

GNU Make Manual or a Stack Overflow search.

- `make` executes the first target by default if no target is provided
- `make` will only execute a given target if it detects than at least one of its prerequisites have been modified since the last execution of that target. If you notice your program's behavior isn't changing, run `make clean` or update your target to include more specific (and sometimes more complex) arguments.
- variables are assigned value using VARNAME = value
- variables are are accessed using $(VARNAME)$
- variables can be changed at command line, e.g., `make CONSTS=config.c`
- automatic variables (more on them here):
  - $@ the target name (main)
  - $^ all of the prerequisites (src/constants.c src/random.c [...] main.c)
  - $< the first prerequisite (src/constants.c)
- patterns (like in filenames) are written as %PATTERN, e.g., %.o, %.c
- functions can be used with patterns to manipulate inputs and patterns for a more flexible command. They have syntax $(function arg1,arg2,...)$; more info here