Programming Practicum Report: How the System for Test Cases Work

R. Ethan Halim

1 Preface

In the start of every source file that I write for all of my practicum submissions, there is this line of preprocessor code below which includes the content of the `preamble.h` header located at the root of this practicum repository.

```
#include "../../preamble.h"
```

The rationale of this document is to explain the single ubiquitous testing system which is enclosed within `preamble.h` that is used in every practicum assignment. I created this document as its own separate file, in order to reduce redundancy, instead of writing the exact same explanation for each practicum report.

2 Inclusion of the Entire C++ STL

Before diving into how `preamble.h` implements the system for testing, it is worth mentioning that the header also includes all of the headers entailed by the C++ standard library by including the utility header `bits/stdc++.h` for convenience. This eliminates the need to list all of the headers needed by the code, albeit sacrificing compilation time.

```
#include <bits/stdc++.h>
```

3 Format of the Test Cases

The test cases of each assignment problem is enclosed inside of a `tests.txt` file. Each test case is formatted as below, wherein the input text to be fed in the program is placed in between the special keylines `%INPUT` and `%OUTPUT`, and the output text expected from the program is placed between the special keylines `%OUTPUT` and `%END`; the latter keyline marks the end of the test case. More test cases can be added following.

```
%INPUT
...
%OUTPUT
...
%END
```

4 The Test System

The code handling the test system is contained in the actual `main` function of the program. It is the reason behind why the code in what was expected to be the `main` function is placed inside of a function called `program` instead. Depending on whether the program is compiled under test mode, the `main` function either directly routes to the `program` function, or perform testing of the test cases by the `tests.txt` file.

```
int main() {
    // For consistent formatting through streams
    std::setlocale(LC_ALL, "en_US.UTF-8");

#ifndef TEST
    // Normal execution of the program through manual user
    ↪  input
    return program(std::cin, std::cout);
#else
    ...
#endif
}
```

Test mode is triggered by inputting a `TEST` preprocessor macro during compilation, which `#ifndef TEST` detects the absence of. Additionally, there is a call to `std::setlocale` preceding the main preambulatory code, which sets all locale settings to accord to `en_US`. This is necessary as to provide uniform consistency in formatting among differing standards, such as how the decimal point would differ from country to country.

The `program` function, which encloses assignment-dependent code, has virtual `cin` and `cout` arguments. This modularity in the input and output streams allow for the flexibility in setting whether the input and output streams should be connected to the user terminal (by referencing them respectively to `std::cin` and `std::cout`, as shown above previously) for regular user execution, or whether they are to be connected to dummy streams for the use of testing and programmatic output-extraction.

```
int program(std::istream& cin, std::ostream& cout);
```

The part of the `main` function handling the tests opens, parses, and performs checking on the test cases in the `tests.txt` file. It firstly opens the file and returns an error message upon failure.

```
int main() {
    ...

#ifndef TEST
    ...
#else
    // Test mode through the TEST macro enabled from the
     ↪  compiler by `-DTESTS`
    std::cout << "[*] The program is currently in test
     ↪  mode!\n";

    // The test cases are stored in a `tests.txt` file.
    std::ifstream tests("tests.txt", std::ios::binary);
    if (!tests) {
        std::cout << "[*] Unable to locate `tests.txt`
         ↪  file!\n";
        return 1;
    }

    ...
#endif
}
```

Then the content within `tests.txt` is parsed as per the format detailed in the previous section. The loop iterates from line to line in the file to check for the special keylines to feed the enclosing text into its own respective `input` or `expected_output` string variable.

```
int main() {
    ...

#ifndef TEST
    ...
#else
    ...
    for (std::string line; std::getline(tests, line, '\n');) {
        if (line != "\%INPUT") {
            continue;
        }

        std::string input;
        for (std::string line; std::getline(tests, line,
         ↪  '\n');) {
            if (line == "\%OUTPUT") {
                break;
            }

            input += line + '\n';
        }

        std::string expected_output;
        for (std::string line; std::getline(tests, line,
         ↪  '\n');) {
            if (line == "\%END") {
                break;
            }

            expected_output += line + '\n';
        }


                ...
    }
        ...
#endif
}
```

Following the retrieval of input and expected_output, input is fed into an std::istringstream object, which is a subinstance of the std::istream class. This instance along with the dummy std::ostringstream stream are able to be passed through the program function as opaque std::istream and std::ostream objects respectively, therefore allowing the system to virtually feed and receive the input and the output of the program. It feeds the input of each test case and receives the output from the program function, which is then compared with the expected output of the test case. If the test case fails, the test system will return the diagnostics information regarding the actual output. Additionally, the test system counts how many test cases failed.

```
int main() {
    ...

#ifndef TEST
    ...
#else
    ...
    size_t test_i = 1, test_failures = 0;
    for (std::string line; std::getline(tests, line, '\n');) {
        ...

        std::cout << "\n[*] Running test #" << test_i << " with
        ↪  the input...\n" << input;
        std::istringstream pseudo_cin(input);
        std::ostringstream pseudo_cout;
        program(pseudo_cin, pseudo_cout);

        std::string output = pseudo_cout.str();
        if (output == expected_output) {
            std::cout << "[*] Test ran successfully.\n";
        }
        else {
            test_failures++;
            std::cout << "[*] Test failed! Dumping output...\n"
            ↪  << output;
            std::cout << "[*] Output did not match what was
            ↪  expected below.\n" << expected_output;
        }

        test_i++;
    }
    ...
#endif
}
```

Upon the conclusion of the testing, the program informs how many test cases have failed.

```
int main() {
    ...

#ifndef TEST
    ...
#else
    ...
        if (test_failures > 0) {
        std::cout << "\n[*] " << test_failures << " test(s)
         ↪  failed in total.\n";
        return 1;
    }
    else {
        std::cout << "\n[*] All tests passed.\n";
        return 0;
    }
#endif
}
```

5 Makefile

In order to easily fascilitate testing through the Makefile, there exists a `test` command, which triggers compilation of the program in test mode, and names the binary as `[program]_test`. As mentioned before, the preprocessor condition handling the activation of test mode relies on a preprocessor macro named `TEST` being defined. Thus, upon the command `make test`, the compiler is called with the `-DTEST` flag, which defines said macro.

```
CXX = g++
FLAGS = -Wall
TARGET = ...
SRC = ...

all: $(TARGET)
    ./$(TARGET)

$(TARGET): $(SRC)
    $(CXX) $(FLAGS) $(SRC) -o $(TARGET)
```

```
test: $(TARGET)_test
	./$(TARGET)_test

$(TARGET)_test: $(SRC)
	$(CXX) ${FLAGS} -g ${SRC} -o ${TARGET}_test -DTEST

clean:
	rm -f $(TARGET) ${TARGET}_test
```