CS211 Fall 2018
Programming Assignment I

David Menendez

Due: Tuesday, September 25, 3:00 AM

This assignment is designed to give you some initial experience with programming in C, as well as compiling, running, and debugging. Your task is to write three small C programs.

Section 1 describes the three programs, section 3 describes how to structure and submit your project, and section 2 describes how your project will be graded. Please read the entire assignment description before beginning the assignment.

Note that the assignment is due at 3:00 AM. **Late submissions will not be accepted or graded.** You are strongly encouraged not to work until the last minute. Plan to submit your assignment no later than Monday, September 24.

1 Program descriptions

You will write three programs for this project. Except where explicitly noted, your programs may assume that their inputs are properly formatted. However, your programs should be robust. Your program should not assume that it has received the proper number of arguments, for example, but should check and report an error where appropriate.

Programs should always terminate with exit code 0 (that is, return 0 from `main`).

1.1 gcd: Reading arguments

Write a program `gcd` that computes the greatest common divisor (GCD) of two integers. `gcd` takes two arguments, both integers, and prints the largest integer that evenly divides both arguments.

Usage

```
$ ./gcd 12 20
4
$ ./gcd 1011 51
3
```

Notes You may assume that the arguments are positive integers that can be represented as (signed) `int` values.

Simple algorithms for computing GCD may be found on-line, but you must write your own code. Do not copy any C implementations you find.

1.2 rot13: String operations I

Write a program `rot13` that encodes a string using the ROT-13 encoding. ROT-13, or "rotate 13" is a simple cipher that maps each letter to a letter thirteen places later or earlier in the alphabet. Thus, 'A' (letter 1) maps to 'N' (letter 14), and vice versa.

`rot13` takes one argument, a string, and prints the result of encoding that string in ROT-13.

Usage

```
$ ./rot13 Uryyb
Hello
$ ./rot13 'Awesome Power!'
Njrfbzr Cbjre!
```

1.3 rle: String operations II

Write a program `rle` that uses a simple method to compress strings. `rle` takes a single argument and looks for repeated characters. Each repeated sequence of a letter or punctuation mark is reduced to a single character plus an integer indicating the number of times it occurs. Thus, "aaa" becomes "a3" and "ab" becomes "a1b1".

If the compressed string is longer than the original string, `rle` must print the original string instead.

If the input string contains digits, `rle` must print "error" and nothing else.

Usage

```
$ ./rle aaaaaa
a6
$ ./rle aaabcccc..a
a3b1c4.2a1
$ ./rle aaabab
aaabab
$ ./rle a1b2
error
```

2 Grading

Each program will be awarded up to 30 points during testing, for a total of 90 possible points for the assignment.

1. The auto-grader will award up to 30 points for each program, according to how many test cases the program completes successfully.
2. If the makefile for the program does not include `-Wall -Werror -fsanitize=address`, the score is multiplied by 0.8.

The auto-grader provided for students includes several test cases, but additional test cases will be used during grading. Make sure that your programs meet the specifications given, even if no test case explicitly checks it. It is advisable to perform additional tests of your own devising. Section 3.5 describes how to use the auto-grader to perform additional tests that you create. Note that these additional tests will not contribute to your final score.

2.1 Academic integrity

You must submit your own work. You should not copy or even see code for this project written by anyone else, nor should you look at code written for other classes. We will be using state of the art plagiarism detectors. Projects which the detectors deem similar will be reported to the Office of Student Conduct.

Do not post your code on-line or anywhere publically readable. If another student copies your code and submits it, both of you will be reported.

3 Submission

Your solution to the assignment will be submitted through Sakai. You will submit a Tar archive file containing the source code and makefiles for your project. Your archive should not include any compiled code or object files.

The remainder of this section describes the directory structure, the requirements for your makefiles, how to create the archive, how to use the provided auto-grader, and how to create your own test files to supplement the auto-grader.

3.1 Directory structure

Your project should be stored in a directory named `src`, which will contain three sub-directories. Each subdirectory will have the name of a particular program, and contain (1) a makefile, and (2) any source files needed to compile your program. Typically, you will provide a single C file named for the program. That is, the source code for the program `gcd` would be a file `gcd.c`, located in the directory `src/gcd`.

This diagram shows the layout of a typical project:

```
src
 +- gcd
 |   +- Makefile
 |   +- gcd.c
 +- rle
 |   +- Makefile
 |   +- rle.c
 +- rot13
     +- Makefile
     +- rot13.c
```

If you are providing your own tests, as described in section 3.5, they will be in a directory named `test` inside the program directory. For example,

```
src
+- gcd
|   +- Makefile
|   +- gcd.c
|   +- tests
|       +- tests.txt
...
```

3.2 Makefiles

We will use `make` to manage compilation. Each program directory will contain a file named `Makefile` that describes at least two targets. The first target must compile the program. An additional target, `clean`, must delete any files created when compiling the program (typically just the compiled program).

A typical makefile for the program `gcd` would be:

```
gcd: gcd.c
        gcc -Wall -Werror -fsanitize=address -o gcd gcd.c

clean:
        rm -f gcd
```

Note that the command for compiling `gcd` uses GCC warnings and the address sanitizer. You will points if your makefile does not include these.

Note that the makefile format requires that lines be indented using a single tab, not spaces. Be aware that copying and pasting text from this document will "helpfully" convert the indentation to spaces. You will need to replace them with tabs (literal tab characters), or simply type the makefile yourself. You are advised to use make when compiling your program, as this will ensure (1) that your makefile works, and (2) that you are testing your program with the same compiler options that the auto-grader will use.

3.3 Creating the archive

We will use `tar` to create the archive file. To create the archive, first ensure that your `src` directory contains only the source code and makefiles needed to compile your project. Any compiled programs, object files, or other additional files must be moved or removed.

Next, move to the directory containing `src` and execute this command:

```
tar czvf pa1.tar src
```

`tar` will create a file `pa1.tar` that contains all files in the directory `src`. This file can now be submitted through Sakai.

To verify that the archive contains the necessary files, you can print a list of the files contained in the archive with this command:

```
tar tf pa1.tar
```

You should also use the auto-grader to confirm that your archive is correctly structured.

It is okay to include any user tests you may have created (see section 3.5), but these will not contribute to your grade.

3.4 Using the auto-grader

We have provided a tool for checking the correctness of your project. The auto-grader will compile your programs and execute them several times with different arguments, comparing the results against the expected results.

Setup The auto-grader is distributed as an archive file `pa1_grader.tar`. To unpack the archive, move the archive to a directory and use this command:

```
tar xf pa1_grader.tar
```

This will create a directory `pa1` containing the auto-grader itself, `grader.py`, a library `autograde.py`, and a directory of test cases `data`.

Do not modify any of the files provided by the auto-grader. Doing so may prevent the auto-grader from correctly assessing your program.

You may create your `src` directory inside `pa5`. If you prefer to create `src` outside the `pa5` directory, you will need to provide a path to `grader.py` when invoking the auto-grader (see below).

Usage While in the same directory as `grader.py` and `src`, use this command:

```
python grader.py
```

The auto-grader will compile and execute the programs in the directory `src`, assuming `src` has the structure described in section 3.1.

By default, the auto-grader will attempt to grade all programs. You may also provide one or more specific programs to grade. For example, to grade only `gcd`:

```
python grader.py gcd
```

To obtain usage information, use the `-h` option.

Program output By default, the auto-grader will not print the output from your programs, except for lines that are incorrect. To see all program output, use the `-v` option:

```
python grader.py -v
```

To see program output for all tests, use `-vv`. To see no program output, use `-q`.

Checking your archive We recommended that you use the auto-grader to check an archive before submitting. To do this, use the `-a` option with the archive file name. For example,

```
python grader.py -a pa1.tar
```

This will unpack the archive into a temporary directory, grade the programs, and then delete the temporary directory.

Specifying source directory If your `src` directory is not located in the same directory as `grader.py`, you may specify it using the `-s` option. For example,

```
python grader.py -s ../path/to/src
```

3.5 Providing your own tests

The tests provided with the auto-grader may not check every possible input scenario. If you wish to be certain that your program is correct, you should design and test additional test cases. This may be done by entering the cases manually and running the program yourself, or by using the user test capability of the auto-grader.

To create additional tests for the auto-grader, create a directory named `tests` inside the directory containing your souce code and make file, and in that directory create a file named `tests.txt` (see section 3.1).

The tests file is structured with alternating lines giving the argument(s) to the program and the expected output. For `gcd`, the argument line must contain two integers, separated by spaces. For example,

```
17 32
1
14 21
7
```

For `rot13` and `rle`, the complete argument line will be passed to the program as-is, without shell interpretation. For example, a `tests.txt` file for `rle` might be:

```
$AAAAA BBB!
$1A5 1B3!1
```

User tests are not required. The capacity for user tests is provided to assist you in development, but it will not directly affect your grade.