CS211 Spring 2020
Programming Assignment I

David Menendez

Due: February 24, 2020, 11:00 PM
Hand in by February 25, 2019, 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 seven small C programs.

Section 1 describes the seven programs, section 2 describes how your project will be graded, and section 3 describes how to structure and submit your project. In particular, section 3.1 gives a simple method for organizing your source code and compiling and testing your code. Please read the entire assignment description before beginning the assignment.

Note that the assignment is due at 11:00 PM, but submissions will be accepted without penalty at late as 3:00 AM the following morning. **Submissions after the grade period 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 February 23.

1 Program descriptions

You will write six 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 `EXIT_SUCCESS` (that is, return 0 from `main`).

1.1 `roman`: Decoding Roman numerals

Write a program `roman` which converts numbers from deciman notation into Roman numerals. `roman` takes a single argument, a string containing a non-negative integer, and prints its Roman numeral equivalent to standard output.

Usage

```
$ roman 3
III
$ roman 54
XIV
$ roman 1977
```

Table 1: Numeric values for Roman numerals

Symbol	Value	Symbol	Value
M	1000	D	500
C	100	L	50
X	10	V	5
I	1		

```
MCMLXXVII
$ roman 0

$ roman 4321
MMMMCCCXXI
```

Roman numerals Roman numerals are written using the symbols I, V, X, L, C, D, and M. The values for these symbols are given in table 1. Often, the value for the Roman numeral can be found by converting each character to its corresponding integer value and adding up the results. For example MMXX becomes $1000 + 1000 + 10 + 10 = 2020$.

The exception occurs when C, X, or I appear before a symbol in the next higher row of table 1. In that case, the value of the first symbol is *subtracted* from the value of the next symbol. Thus, IV is 4, IX is 9, XL is 40, XC is 90, CD is 400, and CM is 900.

Note that we do not use any symbols larger than M, so very large numbers may be written with arbitrarily many M's (e.g., 10,000 is MMMMMMMMMM). In contrast, the output of `roman` will include C and X at most four times (not more than three consecutively), I at most three times, and D, L, and V each at most once. Thus, your program will write IV and not IIII.

Roman numerals do not have a notation for zero, so your program should print nothing if the input is 0.

Notes For `roman`, it is sufficient to use `atoi()` to convert `argv[1]` to an `int`. You will not need to handle any error cases.

The code to handle hundreds, tens, and ones is very similar. Try writing a loop, rather than repeating your algorithm three times.

1.2 `palindrome`: String operations I

Write a program `palindrome` that tests whether a string is a *palindrome*, meaning that the sequence of letters is symmetric and is not changed by reversal. `palindrome` takes a single argument, which is a string containing any combination of upper- and lower-case letters, digits, and punctuation marks. `palindrome` prints "yes" if its string is a palindrome, and "no" otherwise.

Usage

```
$ ./palindrome Bob
yes
$ ./palindrome Robert
no
```

```
$ ./palindrome hohoho
no
$ ./palindrome "Madam, I'm Adam."
yes
$ ./palindrome "A man, a plan, a canal: Panama."
yes
$ ./palindrome "Two men, two plans, two canals: Panamas."
no
```

Notes Note that palindromes are case-insensitive, meaning that "Bob" is a palindrome because "B" and "b" are considered equivalent.

Note that the definition of palindrome only considers letters, and ignores all other non-letter characters. Therefore, "Rise to vote, sir!" is a palindrome, because it is equivalent to "risetovotesir", which is clearly symmetric. (The definition of palindrome we are using is unclear regarding digits, so test inputs will not include digit characters.)

You are free to implement `palindrome` in any manner you find convenient. If you are looking for a challenge, implement `palindrome` such that it allocates no memory, does not modify its argument, and does not examine any character in the argument more than once.

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
```

Notes Note that `rle` prints the original string if its compression method results in a larger string than the input. Thus, in the third example above, it prints "aaabab" (length 6) and not "a3b1a1b1" (length 8).

You MUST NOT assume that input strings have a maximum length. You will need to allocate space to store the compressed string dynamically, based on the length of the input string. Given an input string containing n characters, what is the maximum number of characters it a *printed* output

string? Is it necessary for `rle` to compress the entire input string before determining that it should output the uncompressed string instead?

You MUST NOT assume any maximum number of times a character will be repeated. `rle` should work equally well given a sequence of 10 or 1000 A's. Rather than write your own integer to string function, you can use `sprintf` or `snprintf` with an appropriate format string. Remember that these functions work perfectly well when given a pointer to the middle of an allocated char array, and will begin printing after that pointer. Both functions return the number of bytes witten, which you can use to advance your pointer. (Read the manual and do some experiments to make sure you account for terminator characters properly.)

1.4 `list`: Linked lists

Write a program `list` that maintains and manipulates a sorted linked list according to instructions received from standard input. The linked list is maintained in order, meaning that the items in the list are stored in increasing numeric order after every operation.

Note that `list` will need to allocate space for new nodes as they are created, using `malloc`; any allocated space should be deallocated using `free` as soon as it is no longer needed.

Note also that the list will not contain duplicate values.

`list` supports two operations:

insert n Adds an integer n to the list. If n is already present in the list, it does nothing. The instruction format is an `i` followed by a space and an integer n.

delete n Removes an integer n from the list. If n is not present in the list, it does nothing. The instruction format is a `d` followed by a space and an integer n.

After each command, `list` will print the length of the list followed by the contents of the list, in order from first (least) to last (greatest).

`list` must halt once it reaches the end of standard input.

Input format Each line of the input contains an instruction. Each line begins with a letter (either "i" or "d"), followed by a space, and then an integer. A line beginning with "i" indicates that the integer should be inserted into the list. A line beginning with "d" indicates that the integer should be deleted from the list.

Your program will not be tested with invalid input. You may choose to have `list` terminate in response to invalid input.

Output format After performing each instruction, `list` will print a single line of text containing the length of the list, a colon, and the elements of the list in order, all separated by spaces.

Usage Because `list` reads from standard input, you may test it by entering inputs line by line from the terminal.

```
$ ./list
i 5
1 : 5
d 3
```

```
1 : 5
i 3
2 : 3 5
i 500
3 : 3 5 500
d 5
2 : 3 500
^D
```

To terminate your session, type Control-D at the beginning of the line. (This is indicated here by the sequence `^D`.) This closes the input stream to `list`, as though it had reached the end of a file.

Alternatively, you may use input redirection to send the contents of a file to `list`. For example, assume `list_commands.txt` contains this text:

```
i 10
i 11
i 9
d 11
```

Then we may send this file to `list` as its input like so:

```
$ ./list < list_commands.txt
1 : 10
2 : 10 11
3 : 9 10 11
2 : 9 10
```

1.5 `mexp`: Matrix manipulation

Write a program `mexp` that multiplies a square matrix by itself a specified number of times. `mexp` takes a single argument, which is the path to a file containing a square $(k \times k)$ matrix M and a non-negative exponent n. It computes M^n and prints the result.

Note that the size of the matrix is not known statically. You must use `malloc` to allocate space for the matrix once you obtain its size from the input file.

To compute M^n, it is sufficient to multiply M by itself $n-1$ times. That is, $M^3 = M \times M \times M$. Naturally, a different strategy is needed for M^0.

Input format The first line of the input file contains an integer k. This indicates the size of the matrix M, which has k rows and k columns.

The next k lines in the input file contain k integers. These indicate the content of M. Each line corresponds to a row, beginning with the first (top) row.

The final line contains an integer n. This indicates the number of times M will be multiplied by itself. n is guaranteed to be non-negative, but it may be 0.

For example, an input file `file.txt` containing

```
3
1 2 3
```

```
4 5 6
7 8 9
2
```

indicates that `mexp` must compute

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^2 .$$

Output format The output of `mexp` is the computed matrix M^n. Each row of M^n is printed on a separate line, beginning with the first (top) row. The items within a row are separated by spaces.

Using `file.txt` from above,

```
$ ./mexp file1.txt
30 36 42
66 81 96
102 126 150
```

1.6 `bst`: Binary search trees

Write a program `bst` that manipulates binary search trees. It will receive commands from standard input, and print resposes to those commands to standard output.

A binary search tree is a binary tree that stores integer values in its interior nodes. The value for a particular node is greater than every value stored its left sub-tree and less than every value stored in its right sub-tree. The tree will not contain any value more than once. `bst` will need to allocate space for new nodes as they are created using `malloc`; any allocated space should be deallocated using `free` before `bst` terminates.

This portion of the assignment has two parts.

Part 1 In this part, you will implement `bst` with three commands:

insert n Adds a value to the tree, if not already present. The new node will always be added as the child of an existing node, or as the root. No existing node will change or move as as result of inserting an item. If n was not present, and hence has been inserted, `bst` will print `inserted`. Otherwise, it will print `not inserted`. The instruction format is an `i` followed by a decimal integer n.

search n Searches the tree for a value n. If n is present, `bst` will print `present`. Otherwise, it will print `absent`. The instruction format is an `s` followed by a space and an integer n.

print Prints the current tree structure, using the format in section 1.6.1.

Part 2 In this part, you will implement `bst` with an additional fourth command:

delete n Removes a value from the tree. See section 1.6.2 for further discussion of deleting nodes. If n is not present, print `absent`. Otherwise, print `deleted`. The instruction format is a `d` followed by a space and an integer n.

Input format The input will be a series of lines, each beginning with a command character (`i`, `s`, `p`, or `d`), possibly followed by a decimal integer. When the input ends, the program should terminate.

Your program will not be tested with invalid input. A line that cannot be interpreted may be treated as the end of the input.

Output format The output will be a series of lines, each in response to an input command. Most commands will respond with a word, aside from `p`. The format for printing is described in section 1.6.1.

Usage

```
$ ./bst
i 1
inserted
i 2
inserted
i 1
duplicate
s 3
absent
p
(1(2))
^D
```

As with `list`, the `^D` here indicates typing Control-D at the start of a line in order to signal the end of file.

1.6.1 Printing nodes

An empty tree (that is, `NULL`) is printed as an empty string. A node is printed as a `(`, followed by the left sub-tree, the item for that node, the right subtree, and `)`, without spaces.

For example, the output corresponding to fig. 1 is `((1)2((3(4))5(6)))`.

1.6.2 Deleting nodes

There are several strategies for deleting nodes in a binary tree. If a node has no children, it can simply be removed. That is, the pointer to it can be changed to a `NULL` pointer. Figure 2a shows the result of deleting 4 from the tree in fig. 1.

If a node has one child, it can be replaced by that child. Figure 2b shows the result of deleting 3 from the tree in fig. 1. Note that node 4 is now the child of node 5.

If a node has two children, its value will be changed to the maximum element in its left subtree. The node which previously contained that value will then be deleted. Figure 2c shows the result of deleting 5 from the tree in fig. 1. Note that the node that previously held 5 has been relabeled 4, and that the previous node 4 has been deleted.

Note that the value being deleted may be on the root node.

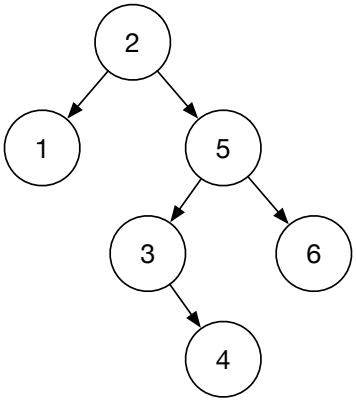
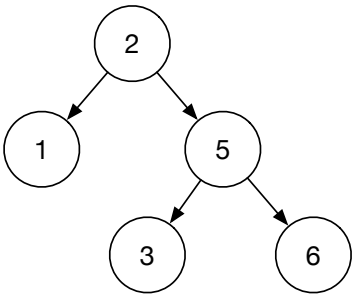
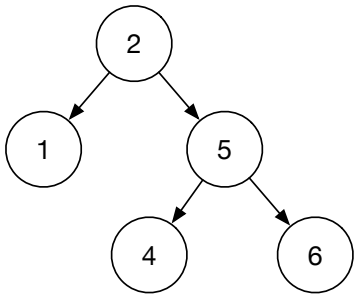


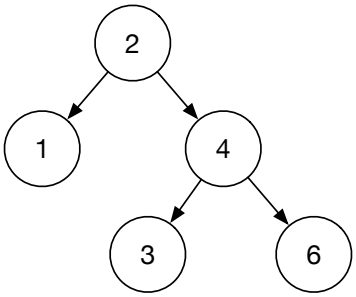
Figure 1: A binary search tree containing six nodes



(a) Deleted 4



(b) Deleted 3



(c) Deleted 5

Figure 2: The result of deleting different values from the tree in fig. 1

2 Grading

Your submission will be awarded up to 100 points, based on how many test cases your programs complete successfully. `roman`, `palindrome`, and `rle` are each worth 10 points; `mexp` and `list` are each worth 20 points; and parts 1 and 2 of `bst` are each worth 15 poins.

The auto-grader provided for students includes half of the test cases that will be used during grading. Thus, it will award up to 50 points.

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.

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 (section 3.2), the requirements for your makefiles (section 3.3), how to create the archive (section 3.4), and how to use the provided auto-grader (section 3.5).

3.1 Getting started

The simplest way to set up your environment is to download the auto-grader and use tar to extract it. (The `$` in these example indicates a prompt. The command you should type comes *after* the prompt and does not include the `$`.)

```
$ tar -xf pa1-grader.tar
```

This will create a directory `pa1/`, containing the auto-grader script and its associated data files. A Makefile in that directory may be used to conveniently initialize, test, and prepare your project for submission.

First, create a subdirectory `pa1/src/`, containing further subdirectories for each of the six programs in the assignment. This may be done all at once using `make tree`, or individually for each program.

```
$ cd pa1
pa1$ mkdir src
```

(Here we are changing the prompt to indicate the working directory. As before, you only type the text after the `$`.)

If you are creating directories manually, you will also need to create subdirectories for each program, containing the source code and makefile for that program. (See section 3.2 for the full layout of the project directory.) For example, to start work on `roman`, one could create a directory `roman` inside `pa1/src/` and copy the sample makefile into it.

```
pa1$ mkdir src/roman
pa1$ cp Makefile_template src/roman/Makefile
```

The template is already set up for `roman`. For other programs, open the Makefile in the editor of your choice and change the definition `TARGET = roman` so that `TARGET` is the name of that program. (Again, this is done automatically when using `make tree`.)

Create or edit your source code, `roman.c`, inside `pa1/src/roman/` using an editor of your choice.

Once you are ready to try compiling your code, you have a few options. One possibility is to use `make` inside your source directory:

```
pa1$ cd src/roman
pa1/src/roman$ make
```

This will compile your code, creating an executable `roman` in `pa1/src/roman`.

You may prefer to keep your source code separate from your executable programs. In that case, you can use the build directories created by the auto-grader, such as `pa1/build/roman/`. These are created when the grader script is invoked. The simplest way to do so is from `pa1/`, using make or by invoking the script directly:

```
pa1$ make
pa1$ ./grader.py
```

This will create `pa1/build/`, attempt to compile your code, and then test your code with the provided suite of test cases. If you are in a hurry, you can use `--init` or `-i` to stop the process after the build directories are created.

```
pa1$ ./grader.py -i
```

Depending on which options you choose, this will create `pa1/build/roman/`, containing a makefile and executable program. If you make additional edits to your source code, you can use this makefile to recompile your program in the build directory.

```
pa1$ cd build/roman
pa1/build/roman$ make
pa1/build/roman$ ./roman 16
```

Note that the auto-grader can recreate the build directory if it is erased, so do not worry about damaging it.

Running fewer tests When developing one program, it is advisable to use the auto-grader each time you feel you have improved your code—even if you do not feel that your program is complete. To save time, you can explicitly tell the auto-grader which programs it should test. For example, to only test `rle`:

```
pa1$ ./grader.py rle
```

Early on in development, it may not be overwhelming to receive information about all failed test cases. Using `--stop` or `-1` (the number 1) tells the auto-grader to stop after the first failed test case. This may also provide additional information, such as the program's input and output. This option can be combined with the restriction of the auto-grader to a single program.

```
pa1$ ./grader.py --stop rle
pa1$ ./grader.py -1
pa1$ make 1
```

The `--verbose` or `-v` option may also be used to request more information from the auto-grader.

Using `pa1/Makefile` A file `pa1/Makefile` has been provided as a convenience. With it, you can run the auto-grader in a single step:

```
pa1$ make
```

Or you can use `make 1` to run the auto-grader with `--stop`.

You may find the makefile most helpful when creating your archive for submission (see section 3.4). The make target `archive` will create the Tar archive `pa1.tar` and then use the auto-grader to verify that `pa1.tar` contains a working submission.

Note: if you use additional source files, you will need to add them to the lists of files given in `pa1/Makefile` in order for the archive to be successfully created.

3.2 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 `factor` would be a file `factor.c`, located in the directory `src/factor`.

This diagram shows the layout of a typical project:

```
src
 +- roman
 |   +- Makefile
 |   +- roman.c
 +- palindrome
 |   +- Makefile
 |   +- palindrome.c
 +- rle
 |   +- Makefile
```

```
|   +- rle.c
+- list
|   +- Makefile
|   +- list.c
+- mexp
|   +- Makefile
|   +- mexp.c
+- bst
    +- Makefile
    +- bst.c
```

3.3 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).

The auto-grader script is distributed with an example makefile, which looks like this (note that an actual makefile must use tabs rather than spaces for indentation):

```
TARGET = roman
CC     = clang
CFLAGS = -g -std=c99 -Wall -Wvla -Werror -fsanitize=address,undefined

$(TARGET): $(TARGET).c
        $(CC) $(CFLAGS) $^ -o $@

clean:
        rm -rf $(TARGET) *.o *.a *.dylib *.dSYM
```

It is simplest to copy this file into the directories for each program, replacing `roman` with the name of that specific program. This will ensure that you programs will be compiled with the recommended options.

It is further recommended that you use `make` to compile your programs, rather than invoking the compiler directly. This will ensure that your personal testing is performed with the same compiler settings as the auto-grader. The makefiles created in the build directory by the auto-grader refer to the makefiles you create in the source directory and therefore pick up any changes made.

You may add additional compiler options as you see fit, but you are advised to leave the compiler warnings, sanitizers, and debugger information (`-g`). The makefile shown here specifies the C99 standard, in order to allow C++-style `//` comments; you may change that to C89, if you prefer.

3.4 Creating the archive

If you have used the simple project layout described in section 3.2, then the makefile provided in `pa1/` can create an archive file for submission by using the targets `pa1.tar` or `archive`. The latter also uses the auto-grader to verify that the archive contains a working submission.

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 should be moved or removed.

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

```
pa1$ tar -vzcf 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:

```
pa1$ tar -tf pa1.tar
```

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

3.5 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:

```
pa1$ 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 `pa1`. If you prefer to create `src` outside the `pa1` 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:

```
pa1$ ./grader.py
```

The auto-grader will compile and execute the programs in the directory `src`, assuming `src` has the structure described in section 3.2.

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 `palindrome`:

```
pa1$ ./grader.py palindrome
```

To stop the auto-grader after the first failed test case, use the `--stop` or `-1` option.

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 for unsuccessful tests, use the `--verbose` or `-v` option:

```
pa1$ ./grader.py -v
```

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

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

```
pa1$ ./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 `--src` or `-s` option. For example,

```
pa1$ ./grader.py -s ../path/to/src
```

Refreshing the build directory In the unlikely event that your build directory has become corrupt or otherwise unusable, you can simply delete it using `rm -r build`. Alternatively, the `--fresh` or `-f` option will delete and recreate the build directory before testing.