CS 2110 Homework 8
Intro to C - 21tenmo

Farzam Tafreshian, Camille Bossut, Sean Crowley, Vivian De Sa Thiebaut, Joshua Vizlai

Fall 2019

Contents

1 Overview

Now that you are experienced with assembly, it's time to dive into the magical world of C programming! In this assignment, you will be creating a simple command-line program called 21tenmo (yes our 2110 version of Venmo!!!) for managing a database of "Accounts" represented in a comma separated values (.csv) format. On a high-level, you will be implementing functionality for adding a new account to the database from the command line along with several other functionalities for allowing an account holder to perform various operations. There are a lot of details, edge cases, and error checking to cover so we highly recommend going through the **entire pdf** and the **source code comments** before you start.

2 Learning Outcomes

1. Understanding C Syntax & Keywords
2. C Program Control Flow
3. Basic Command Line I/O
4. Compiling, Running, and Debugging C Programs

3 Description

3.1 Provided Files (excluding any tester files)

1. 21tenmo.c (TO BE IMPLEMENTED)
2. main.c (TO BE IMPLEMENTED)
3. 21tenmo.h (TO BE MODIFIED)
4. errcodes.h (DO NOT MODIFY)
5. Makefile (DO NOT MODIFY)
6. info.csv (The Database to be used/modified)
7. utility.o (See subsection on utility functions)

3.2 Utilty Functions

The following functions are part of the utility object file which you **should** use in your program wherever necessary. The prototypes of these functions can be found in `21tenmo.h`, however, keep in mind that in order to make everything more simplified, the implementation of these functions are not available for your view.

1. `void readCSV(void)`
   This function validates and reads the sorted info.csv database into the global array (arr) while incrementing the variable DBsize to reflect the number of entries in the database. The reading should happen before any modification is performed in the program. If an invalid row is found in the database, a useful error message is printed out for you and the program is halted.
2. `void writeCSV(void)`
   This function writes the contents of the global array (arr) into the info.csv (database) by replacing the entire file. writeCSV only writes rows with an active accountNumber (i.e. accountNumber != 0) to the csv database. Keep in mind that all floating points are truncated to double-precision format.

3. `void getLine(char *buffer, int size)`
   This is a helper function for reading user input from the standard input (stdin, console). You are **REQUIRED** to use this method instead of any other available standard C functions (e.g. scanf, getline, etc.). This function takes in a length-specified buffer (e.g. char buffer[5]) and reads a **NULL terminated** string **into the passed in buffer**. This function also takes in a size variable which represents the number of characters to read from standard input (stdin). This helper function is super useful as it will discard any inputs passed the buffer size for avoiding buffer overflow issues as much as possible.

4. `void myExit(int status)`
   This is a wrapper function for the standard library exit() function. You are required to use this function whenever the program halts normally (i.e. status = EXIT_SUCCESS) or whenever you need to **terminate** the program from continuing due to bad input, etc. See here for more details on these macros.

5. `void printArray(int numElements)`
   This is a helper function provided solely for debugging purposes. This function will print out the values of the numElements of items from the global array at run-time. numElements is the number of elements to print to standard output (stdout) [1, MAX_CSV_SIZE].

3.3 Implementation

BEFORE START IMPLEMENTING ANY FUNCTIONS, MAKE SURE YOU READ THE 'IMPORTANT NOTES' SECTION.

First look at the header file, 21tenmo.h **which needs to be modified first**. This file contains the definitions for structs and macros used in this program. In this particular header, you will need to implement the Account struct (you can read about typedefs here) with the following **EXACT** field names and **EXACT** data types.

- accountNumber, an unsigned integer (already provided)
- personName, a character array of size MAX_NAME_SIZE + 1 (space for the NULL terminator)
- balance, a float
- requesterNumber, an unsigned integer
- requestAmount, a float

In the main.c file, you will need to implement the main method that checks for command line arguments preceded by flags '-a' or '-l' (see `printUsage()` and section 5.1 for more details). If the arguments passed in are not preceded by '-a' (for adding an account to DB) or '-l' (for logging in a current user), you should print the correct usage and call myExit using EXIT_FAILURE. If the correct flags are provided:

- -a should be followed by a string of the person's name (surrounded by quotations ""), i.e. "Kendrick Lamar". This person should then be added to the database **with the next available accountNumber** (with other fields being set to default value of zero).

- -l should be followed by an integer value representing a person's account number. If the account number specified is invalid, an error message should be shown and you should call myExit using EXIT_FAILURE.

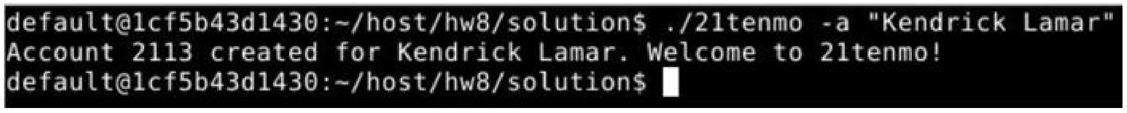
If any of the above specification are not met following the correct flags, you should call myExit using EXIT_FAILURE. Else, display the user's options by calling the **provided** function, userMenu, after validating the accountNumber. Continue running the program until they save and exit.

Notice that there's already a global variable 'myErrNo' for setting error codes upon successful or failed termination of the program depending on what went wrong. Now open up errcodes.h and read the comments associated with each error code. Throughout the program, **you are responsible** for setting the correct error codes (or NO_ERROR) before termination of the program, especially if a database operation fails. This is a common way of being informed of the results of a program execution in C (for more info, look into C's errno and errno.h, however, keep in mind that we have our own pre-defined error codes in errcodes.h).

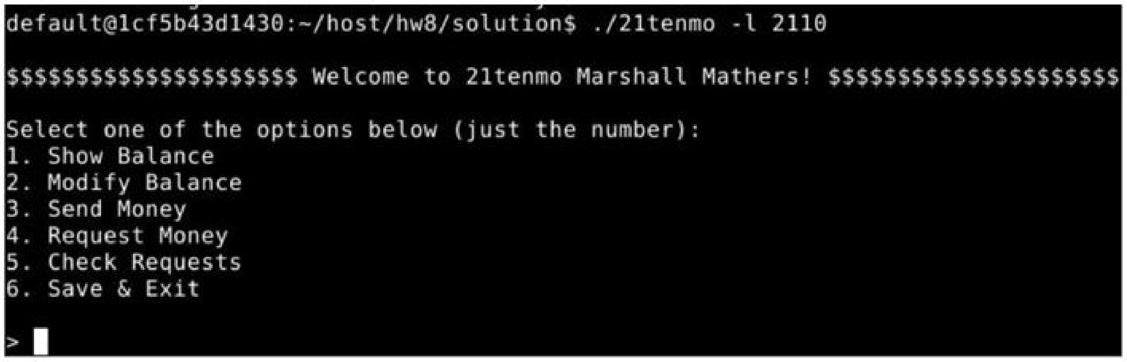
Functions to implement or modify:
Note: For any function that deals with input parsing/validation check the section on 'Helpful Man Pages/String functions'.

- **executeCommand()**: You must implement code to show the user's balance, and update the csv (see the subsection on utility functions) in the case where the user wants to exit.
- **modifyBalance()**: This function should take user input (both negative and positive) to modify the balance of the user's account. Make sure that if the user is withdrawing money, they have enough in their account to withdraw the amount requested (a user's balance should never be negative). You should also print the amount modified by at the end of the transaction.
- **sendMoney()**: This function should allow a user to send funds to another user by specifying their account number and an amount (see formatAndCheckCurrency()). This transaction should fail if: the account number specified is invalid or inactive, the account number specified is the same as the users', the amount specified is negative, or finally if you don't have enough funds to send the requested amount. Update the required fields of the corresponding accounts on success.
- **formatAndCheckCurrency()**: This is the function that should validate any input amount in other functions. This function check's that an amount specified is valid, meaning it contains the following characters: commas, at most one period (for decimals), at most one minus sign (for negative values), and other than that it should be all numbers. The function should return 0 on success and nonzero value otherwise. Note that you **do not need** to perform comma validation, upon error, the passed in currencyString should not be modified. Some valid cases are: 1000, 10000.00, 1,0,0,0,0.00. Some invalid cases are - -1000.00, 1/00.00.
- **requestMoney()**: This function should allow a user to request money from another user by specifying their account number, and an amount. This function should fail if: The account number specified is invalid or inactive, if the account number specified is the same as the users', the account specified already has a request pending, or finally if the amount requested is negative (make sure the corresponding errorcode is set in case of failure). Don't forget to Update the request fields for the corresponding account upon success.
- **checkRequests()**: This function should allow a user to check their requests and approve or deny the request received by entering 'Y' or 'N'. The request should be displayed with the account number it was sent from, and the amount requested. If the request is denied OR the request is approved and amount requested is higher than the users' balance then the request should be **cancelled**. If input is other than 'Y' or 'N' an error message should be displayed and the request should not be modified. If there are no pending requests for the logged in account, this function should print a message and return.

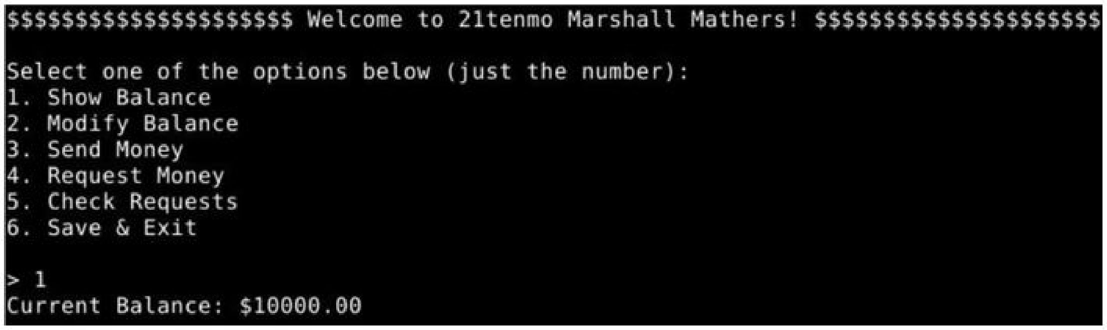
3.4 Example Execution



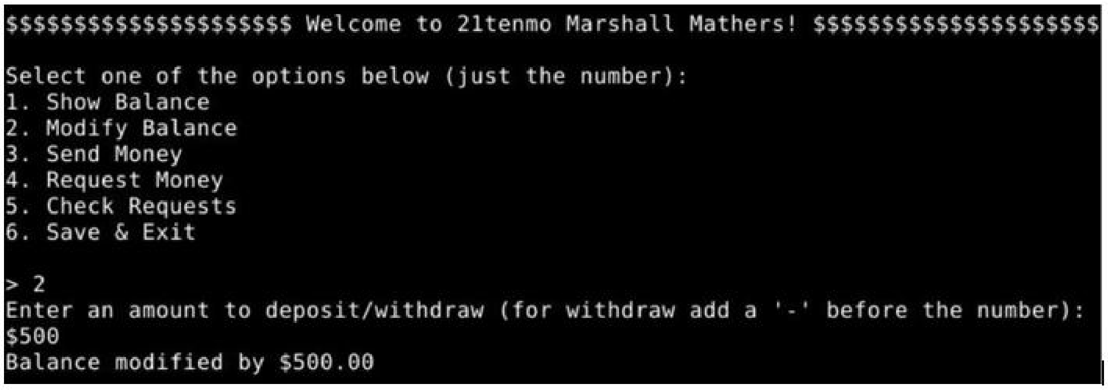
```
default@1cf5b43d1430:~/host/hw8/solution$ ./21tenmo -a "Kendrick Lamar"
Account 2113 created for Kendrick Lamar. Welcome to 21tenmo!
default@1cf5b43d1430:~/host/hw8/solution$
```



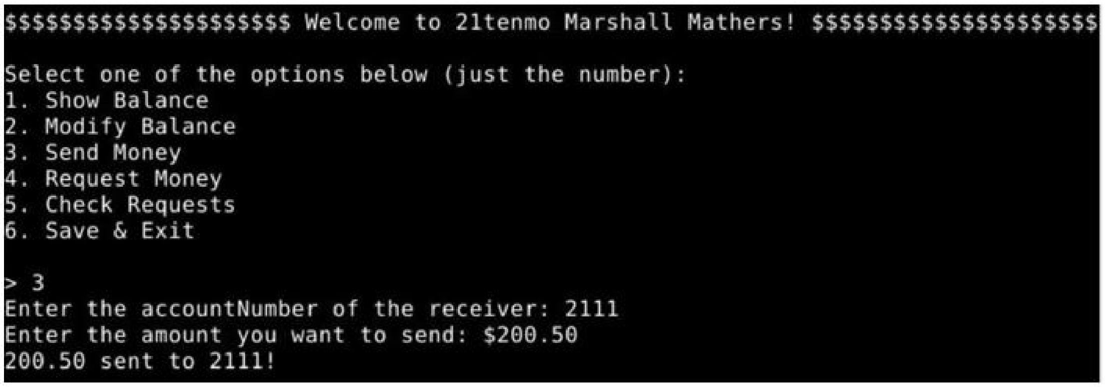
```
default@1cf5b43d1430:~/host/hw8/solution$ ./21tenmo -l 2110

$$$$$$$$$$$$$$$$$$$$$ Welcome to 21tenmo Marshall Mathers! $$$$$$$$$$$$$$$$$$$$$

Select one of the options below (just the number):
1. Show Balance
2. Modify Balance
3. Send Money
4. Request Money
5. Check Requests
6. Save & Exit

>
```



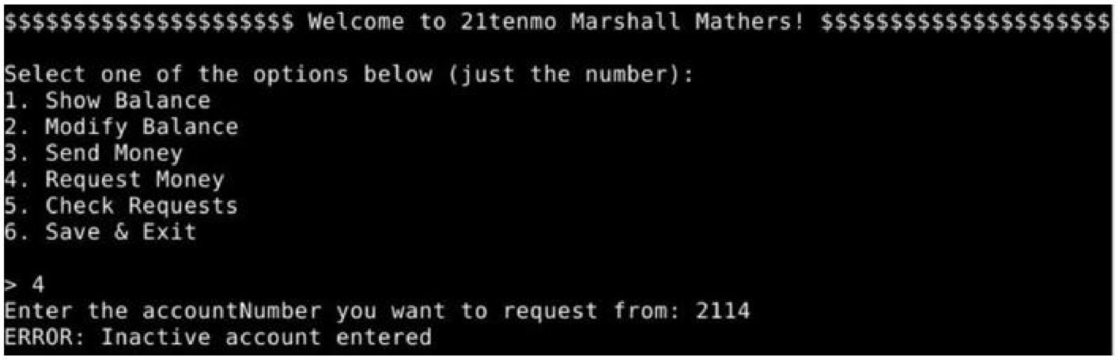
```
$$$$$$$$$$$$$$$$$$$$$ Welcome to 21tenmo Marshall Mathers! $$$$$$$$$$$$$$$$$$$$$

Select one of the options below (just the number):
1. Show Balance
2. Modify Balance
3. Send Money
4. Request Money
5. Check Requests
6. Save & Exit

> 1
Current Balance: $10000.00
```



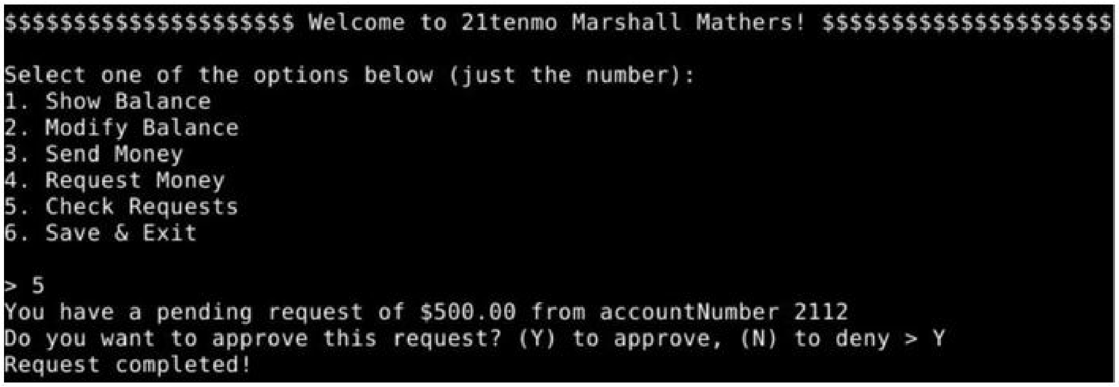
```
$$$$$$$$$$$$$$$$$$$$$ Welcome to 21tenmo Marshall Mathers! $$$$$$$$$$$$$$$$$$$$$

Select one of the options below (just the number):
1. Show Balance
2. Modify Balance
3. Send Money
4. Request Money
5. Check Requests
6. Save & Exit

> 2
Enter an amount to deposit/withdraw (for withdraw add a '-' before the number):
$500
Balance modified by $500.00
```



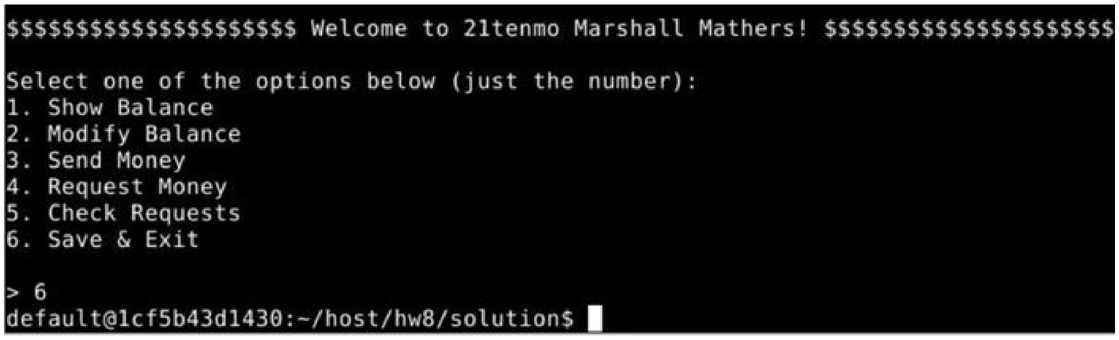
```
$$$$$$$$$$$$$$$$$$$$$ Welcome to 21tenmo Marshall Mathers! $$$$$$$$$$$$$$$$$$$$$

Select one of the options below (just the number):
1. Show Balance
2. Modify Balance
3. Send Money
4. Request Money
5. Check Requests
6. Save & Exit

> 3
Enter the accountNumber of the receiver: 2111
Enter the amount you want to send: $200.50
200.50 sent to 2111!
```



```
$$$$$$$$$$$$$$$$$$$$$ Welcome to 21tenmo Marshall Mathers! $$$$$$$$$$$$$$$$$$$$$

Select one of the options below (just the number):
1. Show Balance
2. Modify Balance
3. Send Money
4. Request Money
5. Check Requests
6. Save & Exit

> 4
Enter the accountNumber you want to request from: 2114
ERROR: Inactive account entered
```



```
$$$$$$$$$$$$$$$$$$$$$ Welcome to 21tenmo Marshall Mathers! $$$$$$$$$$$$$$$$$$$$$

Select one of the options below (just the number):
1. Show Balance
2. Modify Balance
3. Send Money
4. Request Money
5. Check Requests
6. Save & Exit

> 5
You have a pending request of $500.00 from accountNumber 2112
Do you want to approve this request? (Y) to approve, (N) to deny > Y
Request completed!
```



```
$$$$$$$$$$$$$$$$$$$$$ Welcome to 21tenmo Marshall Mathers! $$$$$$$$$$$$$$$$$$$$$

Select one of the options below (just the number):
1. Show Balance
2. Modify Balance
3. Send Money
4. Request Money
5. Check Requests
6. Save & Exit

> 6
default@1cf5b43d1430:~/host/hw8/solution$
```

4 Important Notes

1. accountNumber(s) should start at FIRST_ACCOUNT_NUMBER and increase **in order** every time an account is added to the database all the way to the maximum size of the DB (Hint: create a macro for the upper boundary for account validation checks). Keep in mind that **at any time** the database size should not exceed the MAX_CSV_SIZE macro.

2. A valid accountNumber (accountNumber in range) that is not in the DB (e.g. accountNumber = 2120 but there isn't an account associated with that number) is considered an INACTIVE account (meaning that once read into the global array, accountNumber of that index is zero).

3. The personName's length should be within the MIN_NAME_SIZE and MAX_NAME_SIZE (see 21tenmo.h for more information regarding the **NULL** terminator.

4. The database should only be modified (written back) when a user is done with all operations and exits the program, see example for operation '6' (Save and Exit).

5. As mentioned in section 3.3, make sure that myErrNo is set to the correct errcode (e.g. myErrNo = ERR_INVALID_AMOUNT **whenever** the parsed/passed in amount is invalid).
6. For all floating point operations make everything double-precision format (i.e. $25.12).
7. DO NOT call getLine or any other input-reader method more than necessary. For example, if you are getting an accountNumber and amount, you should only call getLine twice.
8. DO NOT modify the given include(s) list. All necessary libraries are already included.
9. DO NOT try to read and write the database with functions/code **other than** the provided readCSV() and writeCSV() utility functions. Use the myExit function provided **instead** of stdlib's exit function.
10. DO NOT modify the **given macros and function prototypes and declarations** (we are going to test the given functions but you are allowed to create your own helper functions if you want).
11. There are many checks and validations in place for database (info.csv) entries that will block your program from running if an invalid row is found, therefore, be very careful with manually modifying the .csv file.
12. Wherever applicable, DO NOT use hard-coded sizes, try using the built-in `sizeof` operator instead (this is a very important concept in any C program).
13. Although you are not gonna be tested on buffer overflows, whenever a buffer is necessary (e.g. before calling getLine), make sure you do not allocate more than what's needed (or less than :D); this goes for input reading and parsing as well. However, DO NOT dynamically allocate any memory (malloc, calloc, realloc, etc.).
14. We highly recommend that you create your own macros for usage in your program. Feel free to add them in 21tenmo.h (or the C file if it is only used in one file) as you see fit.

5 Useful Tips

5.1 Command Line Arguments

When you write a C program, you can work with arguments you receive on the command line through two parameters you receive in the `main` function, `argc` and `argv`.

argc: The number of command line arguments you receive.

argv: An array of your command line arguments in string form.

Note: the zeroth argument to your program is always going to be the name of the program itself. This means that `argc` and `argv` will indicate that you have one more parameter than you might expect.

5.2 Helpful Man Pages/String functions

The following are some helpful functions you should know to complete this homework. To get more detailed descriptions of any of these functions, or others in the C standard library, type into your terminal emulator:

```
$ man <function_name>
```

This will print the corresponding man page for that function.

- `int strcmp(const char *s1, const char *s2)`
  This function allows you to compare two strings. Returns 0 if they are the same, and a nonzero value otherwise.

- `size_t strlen(const char *str)`
  This function returns the length of the string passed in. (note: size_t is an unsigned integer data type).
- `char *strcpy(char *dest, const char *src)`
  This function copies the 'string' pointed to by src to the dest(ination) buffer.
- `unsigned long int strtoul(const char *nptr, char **endptr, int base)`
  This function allows you to pass in a string and base (you'll usually want base 10) and will return the integer value of the string you passed in, according to the specified base. It will return 0 in case of invalid parameters. For now, you may replace the 'endptr' argument with NULL (we recommend understanding the type of this argument and how it is used). **Use this function instead of other functions such as `atoi`.**
- `float strtof(const char *nptr, char **endptr)`
  This function is similar to strtoul but will convert the string to float instead.

5.3 Debugging with GDB

If you run into a problem when working on your homework, you can use the debugging tool, GDB, to debug your code! Former TA Adam Suskin made a series of tutorial videos which you can find here here.

When running GDB, if you get to a point where user input is needed, you can supply it just like you normally would. When an error happens, you can get a java-esque stack trace using the `backtrace(bt)` command.

6 Checking Your Solution

When you have programmed more than 100 lines of code and you are about to compile it
@techprogramming
TECH PROGRAMMING
Something's wrong, I can feel it

We have provided a Makefile for this assignment that will build your project. (We highly recommend looking at the Makefile source and familiarizing yourself with Make and GCC for future)

Here are the available commands for your usage with this Makefile:

To compile and run the program with different command-line arguments first run: `make 21tenmo`
then upon a successful compilation, run the program as specified by its usage
(e.g. `./21tenmo [flag] [argument]`).

OR:

1. To clean your working directory (use this instead of manually deleting the .o files): `make clean`
2. To run the tests without gdb: `make run-tests`
3. To run a specific test case: `make check-tests TEST=test_case_name`
4. To run the tests with gdb (except main.c due to the way autograder is setup): `make run-gdb`

Your files must compile with our Makefile, which means it must compile with the following gcc flags (taken from the Syllabus):
`-std=c99 -Wall -pedantic -Wextra -Werror -O2 -Wstrict-prototypes -Wold-style-definition -g`

All non-compiling homeworks will receive a zero. If you want to avoid this, do not run gcc manually, use the Makefile as described above.

Note: The grade (or percentage) that you see locally (from the check library) might be slightly different from the grade on Gradescope (which is the overall grade).

Note: Due to the nature of this program (I/O), the checker may not reflect your actual grade on this assignment. We reserve the right to update the checker as we see fit when grading your solution.

7 Deliverables

Submit ONLY the following files (all at the same time) to Gradescope under the Homework 8 assignment:

1. 21tenmo.c
2. 21tenmo.h
3. main.c

Please check your submission after you have uploaded it to Gradescope to ensure you have submitted the correct files.

Homework due date: **October 24, 2019**.

8 Rules and Regulations

8.1 General Rules

1. Starting with the assembly homeworks, any code you write must be meaningfully commented. You should comment your code in terms of the algorithm you are implementing; we all know what each

line of code does.

2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.
3. Please read the assignment in its entirety before asking questions.
4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.
5. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

8.2 Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files unless otherwise noted.
2. When preparing your submission you may either submit the files individually to Canvas/Gradescope or you may submit an archive (zip or tar.gz only please) of the files. You can create an archive by right clicking on files and selecting the appropriate compress option on your system. Both ways (uploading raw files or an archive) are exactly equivalent, so choose whichever is most convenient for you.
3. Do not submit compiled files, that is .class files for Java code and .o files for C code. Only submit the files we ask for in the assignment.
4. Do not submit links to files. The autograder does not understand it, and we will not manually grade assignments submitted this way as it is easy to change the files after the submission period ends.

8.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.
2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Canvas/Gradescope. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Canvas/Gradescope.
3. There is a 6-hour grace period added to all assignments. You may submit your assignment without penalty up until 11:55PM, or with 25% penalty up until 5:55AM. So what you should take from this is not to start assignments on the last day and plan to submit right at 11:54AM. You alone are responsible for submitting your homework before the grace period begins or ends; neither Canvas/Gradescope, nor your flaky internet are to blame if you are unable to submit because you banked on your computer working up until 11:54PM. The penalty for submitting during the grace period (25%) or after (no credit) is non-negotiable.

8.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use a private repository on github.gatech.edu

8.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, share algorithms and pseudo-code, as well as help each other debug code. What you shouldn't be doing, however, is pair programming where you collaborate with each other on a single instance of the code. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, because it is frequently the case that the recipient will simply modify the code and submit it as their own.

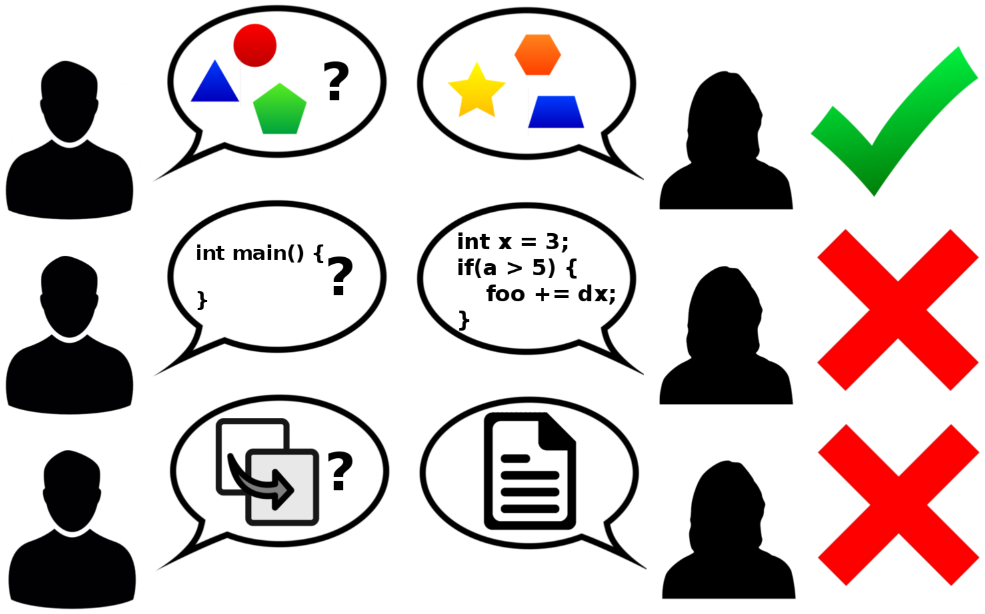


Figure 1: Collaboration rules, explained colorfully