CS 2110 Homework 8
Intro to C

Jim Harris, Henry Harris (no relation), Manley Roberts, Joshua Viszlai, Gibran Essa

Spring 2019

Contents

1 Overview

Whoa, this semester turned out busy, huh? With all of the quizzes, homeworks, timed labs, and midterms (for other courses like CS 1332), it's just so hard to keep up! We need a way to keep track of all of our work... We could always use Google Calendar but who wants to deal with the information privacy concerns created by the monopolies made possible by our oppressive capitalist system? And who wants to deal with the so-called "minimalist" design in modern GUIs? **THERE'S GOT TO BE A BETTER WAY**. Thankfully, we just learned C... we'll show Google true minimalism... we'll make our own To-do list application!

2 Learning Outcomes

1. C Program Control Flow
2. C File I/O
3. Command Line Arguments
4. Using man pages
5. Makefile basics

3 Important Things to Know

3.1 File I/O

For this homework you will need to do I/O with files. This is so that we can save our To-do list to a file and load it back up later! It wouldn't be very useful if we didn't have persistence!

You will find file I/O to work similarly to how it worked in Java; for the most part there should be a function in C that is functionally similar to one you might have used in Java. Let's just cover briefly how to open a file and do some things with it.

Note that all of the functions in this section have a handy-dandy man page. This means that you can type `man insert_command_name_here` into a mac or linux terminal (or into Google; the man pages are all online) and it will tell you all sorts of things about the parameters and usage! For this reason, we will not be giving you full descriptions, but will just be providing some short examples.

The below example covers:

1. FILE *fopen(const char *pathname, const char *mode),
2. char *fgets(char *s, int size, FILE *stream),
3. int fprintf(FILE *stream, const char *format, ...),
4. int atoi(const char *nptr),
5. int fclose(FILE *stream),

```
FILE *in_file = fopen("my_file.txt", "r"); // Open a file for rEADING. We can only rEAD it,
                                           // not write to it.
if (in_file == NULL) {
    return 1; // Always check for errors when you open files! It might have choked!
}
FILE *out_file = fopen("out_file.txt", "w"); // Open a file for wRITING. We can only
                                             // wRITE to it, not read from it. This
                                             // will also create the file if it doesn't exist
                                             // already.
if (out_file == NULL) {
    fclose(in_file); // Close files when we are done with them
    return 1; // Always check for errors when you open files! It might have choked!
}
char buffer[128]; // Make a buffer to read data into
// We read data using fgets; this will read up to the end of the line, the end of the buffer
// length, or the end of the file. When it hits the end of the file it returns 0 which is FALSE
// in C.
while (fgets(buffer, 128, in_file)) {
    fprintf(out_file, buffer); // fprintf is just like printf, except it accepts a file
                               // descriptor to print to
}
fclose(in_file);
fclose(out_file);

FILE *numbers_file = fopen("numbers.txt", "r");

// Checks to make sure the file descriptor is good and read first line.
if (!numbers_file || !fgets(buffer, 128, numbers_file)) {
    return 1;
}
int num_numbers = atoi(buffer); // atoi parses an int from a string
int sum = 0;
for (int i = 0; i < num_numbers && fgets(buffer, 128, numbers_file); i++) {
    sum += atoi(buffer); // Sum up all subsequent numbers in the file
}
fclose(numbers_file);

fprintf(stdout, "The Sum is %d\n", sum); // Print out the sum!
```

Fun fact! `printf` is just `fprintf` with `stdout` hard coded as the stream. You can actually pass `stdout` as the stream to `fprintf` and get the same functionality! You can also do a similar trick with `stdin` and `fgets` to get input from the user's command line **WINK WINK NUDGE NUDGE**.

3.2 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 commnad 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.

The below example will print out all command line arguments to the program. `argv[0]` will be the name of the program.

```
int main(int argc, const char *argv[]) {
    for (int i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
}
```

3.3 Provided Items

1. todo.c,
2. todo.h,
3. todo_data.c
4. todo_helpers.c
5. useful_strings.h,
6. example_program_run.txt,
7. example_save_file_.txt (and others in the test save files directory),
8. Video of our reference implementation
9. GDB Tutorial Videos (RIP Adam Suskin)
10. Autograder stuff; see the section on Checking your Work.
11. Supplemental Reading, Unix Programmer's Manual v7, Make - A Program for Maintaining Computer Programs

We give you the file `todo.c/h` which has some useful helper functions for you. Take a look at it! It will save you some work. One of the ones in there is a wrapper around `fgets` which removes the newline from the end if it read one in. This makes formatting a pain, so we're just giving you that helper to make your life a bit easier. There's also one in there to read in a todo list item from a file that you will find quite helpful.

The header file also contains the declaration of the `struct todo_list_item_t`, the `Todo_list` data structure (array of items and its length), and some macros for the max lengths of some fields. **Use these in your solution**, or feel free to get rid of them if you like. Just keep in mind that some of the provided helper functions will use these variables. Since we want to keep *definitions* outside of header files, the header file will only *declare* these global fields. They will be *defined* in the file `todo_data.c`

Since there is alot of printing out of specific strings for this homework, we are providing you with some of the more annoying ones as macros in the file `useful_strings.h`. Please use this file! Some of the other strings that are not provided as macros are format strings that you will need to write yourself

To help you visualize how the app should work, we've recorded a short video of our reference implementation. Please take a look at it so you can see how it should look! We are also providing the output in the `example_program_run.txt` file if you want to take a closer look at it without rewatching the video. We also include the save file we used.

Additionally, if you would like to read a bit more about `make` and how it works, we've included an original paper written by the authors of `make`. The paper in all is only about 8 pages long, so definitely check it out if you would like to learn more!

3.4 How You Will be Graded

You will be graded on the output of your program. This means that the output for each "screen" will need to be the same as that of our reference implementation for you to receive credit for the "screen". To make sure that you can match yours up exactly, take a look at the provided `example.txt` file or the provided video. The `useful_strings.h` file will also be helpful. You will also see the correct output when you use the autograder.

3.5 Input Validation Requirements

For this assignment, you will not need to perform too much input validation. Specifically, if you need to parse a number, you may assume you will be provided a proper number in the expected range rather than anything else (with a couple exceptions which will be explained). However, you will need to make sure that files exist and that read operations succeed and the like. If you ever have something fail, print out the correct error message if specified. If no error message is specified just terminate the program with an exit status of 1 without further output.

3.6 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 before his tragic and horrific death in a Rocket League accident. They say he's still flaming his team mates to this day. Find them 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 assuming your Makefile works and you included the debugging flag!

4 Solution Description

Our To-do list will be an interactive command line application. This means that we will be processing input from the user and using it to change the state of our app and print things out. You can use `fgets` to get user input as described in a section 3.1.

The project will consist of the following files:

1. todo.c (main file with your main function, you'll make this!)
2. Makefile (you'll make this!)
3. todo.h (provided, but feel free to edit! Your solution must include this file)
4. todo_helpers.c (provided, but feel free to edit! Your solution must include this file)
5. todo_data.c (provided, but feel free to edit! Your solution must include this file)

6. useful_strings.h (provided, but feel free to edit! Your solution must include this file)

The To-do list will store a maximum of 100 items. We do this by having a big, statically allocated array of our to-do list items that is 100 in length (defined in the header file). Now, just because we have 100 slots for to-do list items does not mean we're using all of them! We have a variable called `Todo_list_length` (declared in the provided header file) which is always equal to the number of elements we are actually storing. Our list will be full once this variable equals 100. Once it is full, you will not be able to add more items.

This also means that valid items in our list have indices in the range [0, `Todo_list_length - 1`].

Our To-do list will support the following functionality:

1. Printing All To-do List Items
2. Adding a To-do List Item
3. Marking a To-do List Item as Completed
4. Removing All Completed Items
5. Saving To-do List to File

4.1 Makefile

Write a small Makefile to compile your program. It must have the following targets:

1. todo, the executable that runs the program.

You may include other targets for dependencies.

The `todo` task should be run by default.

You need to use the following compiler flags when compiling any C files:

```
-std=c99 -Wall -pedantic -Wextra -Werror -O2 -Wstrict-prototypes -Wold-style-definition -g
```

The `-g` flag enables debugging symbols. This is **very important** if you want to step through your code using `gdb`!

Remember that we linked that short paper on make in the Provided Items section! It might be helpful if you're feeling confused. There are also a number of online tutorials that you may find helpful.

4.2 Initialization

The program should take 0 or 1 additional command line arguments. This means there are three scenarios to consider when starting your application:

1. Your program receives 0 arguments,
   If the program receives no command line arguments, it should start off with an empty To-do list and go straight to the main menu screen
2. Your program receives 1 argument,
   If the program receives one argument, that argument should be the name of a file that contains the To-do list data. In this case, the items should be read from that file and the `Todo_list` array should be initialized with those items. You will probably find the provided `read_todo_list_item` function very helpful for this. Each item will appear in the file with its fields separated by newlines, and each item

will appear one after another. See the provided header file if you need more specifics. Afterwards, it will show the main menu screen.

If the file provided does not exist, abort execution with an exit status of 1 after printing the error message indicating that the file does not exist (this message includes the file name, so the string in the useful_strings.h file is not complete).

3. Your program receives 2 or more arguments.
   If the program receives more than two arguments, abort execution (with an exit status of 1) and print an error message for usage. See the useful strings header file for the specific error message to print.

If there are no problems, proceed to the main menu screen.

4.3 Main Menu

Your application will have a main menu. From this main menu, you should collect user input in the form of a number. Based on the number, you should do one of the following things:

1. Print my To-do List
2. Add a To-do List Item
3. Mark an item completed
4. Remove all completed items
5. Save as
6. Quit

See the examples for what this screen should look like. Options 1 through 5 should produce some output bring the user to a screen which requests more information to perform the action if needed (explained in subsequent sections).

After the action is performed, the user should be brought back to the main menu where they may continue working.

If option 6 is chosen, the program should terminate with exit status 0 and produce no further output.

The main menu should look like this:

```
================================== TO-DO LIST ==================================

What would you like to do?

1) Print my To-do List
2) Add a To-do List Item
3) Mark an item completed
4) Remove all completed items
5) Save as
6) Quit

> 1
```

Hint: Check out the macros `MAIN_MENU_HEADER`, `QUERY`, `OPTIONS_LIST`, `GRAB_INPUT`, and`INVALID_CHOICE` in the file `useful_strings.h`

4.4 Print All To-do List Items

This action will accept no additional input and will print out all of the To-do list items in the order they appear in the global `Todo_list` array. Some notes:

1. Days and months should be padded to 2 digits with zeros. For example, if the due month of the item is May, the month will be printed as 05.
2. Years should be zero padded to 4 digits as well. So if you have some ancient to-do list item, for example, "Fend off invading forces from Constantinople" might have a due date in the year 711 A.D so should be printed as 0711.
3. If the item is completed, print out "Completed!" instead of the due date. Otherwise print the due date.

Again, see the examples for exactly how it should look.

This screen should look like this:

```
-------------------------------- YOUR TO-DO LIST ------------------------------

Buy milk

Due: 02/26/2019
Description: Drive to the store and get some milk for your cereal
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Graduate!

Due: 05/04/2019
Description: I made it!
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Wash bear

Due: 04/10/2019
Description: Bear is a bit dirty from getting hugged by too many CS majors
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Processor Design Exam

Due: 02/27/2019
Description: Learn verilog so that I don't get owned
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Grade quiz 3

Due: 02/25/2019
Description: Make sure that the quiz gets graded
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Lay seige to Constantinople

Due: 05/29/1453
Description: Need to get the trebuchets set up properly
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Hint: Check out the macros `TODO_LIST_HEADER`, `COMPLETED` and `LINE_BREAK` in the file `useful_strings.h`

4.5 Add a To-do List Item

This action requires some additional input from the user.

Firstly, if there already exist 100 items in the list (which is the maximum), the user should not be queried for output, and they should be returned to the main menu after printing the string “You need to drop some classes...”.

However, if there is space remaining, you should ask the user to provide you with the information needed for a to-do list item. Take a look at the examples, and keep in mind that `read_todo_list_item` function should be useful.

The maximum length of the user input you should expect for titles is 64 bytes (including null bytes and newlines).

For days and months, the maximum length you should expect is 4 bytes (including null bytes and newlines). For years it will be 6 bytes (including null bytes and newlines).

If you use the helper function these limits probably will not be of much concern.

This screen should look like this:

```
------------------------------ ADD TO TO-DO LIST -----------------------------

Type in on separate lines in this order:

- Title
- Description
- If it is completed (1 or 0)
- Due Day
- Due Month
- Due Year

Finish Autograder for Homework 8
The autograder needs to get done... Good thing Maddie is helping me!
0
28
2
2019
```

Hint: Check out the macros `ADD_TO_LIST_HEADER`, `ADD_TO_LIST`, and `LIST_MAX` in the file `useful_strings.h`

4.6 Mark a To-do List Item as Completed

This action requires some additional input from the user.

This screen should prompt the user to enter the index of the item in the list they want to mark completed. The selected item should simply be marked completed (nothing special need be done if it is completed already) and the user should be returned to their menu.

This screen should look like this:

```
------------------------------ MARK ITEM COMPLETED -----------------------------

Enter the index of the item you want to mark completed:

> 1
```

Hint: Check out the macros MARK_ITEM_COMPLETED_HEADER, MARK_ITEM_USER_INPUT and GRAB_INPUT in the file useful_strings.h

4.7 Remove All Completed Items

This action will accept no additional input.

This action should remove all items from the list that are completed. Any items that are not completed should be in the same order relative to each other that they were before this operation occurred (though they may now be at a lower index than when they started).

This may be done in-place or out-of-place. If you do not know what this means, it just means you do not need to worry about being space efficient in the way you do this.

Once this action is performed, the number of removed items should be printed for the user before returning them to the menu.

This screen should look like this:

```
---------------------------- REMOVE COMPLETED ITEMS ----------------------------

Success! 2 items removed!
```

Hint: Check out the macro REMOVE_ITEM_HEADER in the file useful_strings.h

4.8 Save To-do List to File

This action requires some additional input from the user.

It should read in a file name from the user and then write the items in the Todo_list array to that file. It should output the list in the same format it would have been read in. This is so that we can open that file back up in our program later.

This screen should look like this:

```
--------------------------------- SAVE FILE AS ---------------------------------

Please enter the file name you would like to save to up to 127 characters in length:

> fresh_save.txt
```

Hint: Check out the macros SAVE_FILE_HEADER, INPUT_FILE_NAME, and GRAB_INPUT in the file useful_strings.h

5 Where do I start??? There is so much stuff here. What is wrong with you people?

We strongly advise that you do the first 5 things on this list as soon as possible. Doing these first things will get alot of the printing things out of the way, and also get you thinking about the design of this small application. Having the design figured out makes adding features much simpler. Please do these first few steps sooner rather than later so that you can have a better idea of what it will take to finish.

1. Write your Makefile. Make sure that you can compile your code! Try making a simple hello world program or something first.

2. Get the main menu to print out the options correctly! The other tests depend on this!
3. Make it so that option 6 at the main menu terminates the program.
4. Make it so that you can print the todo list out... a good number of the other tests depend on this.
5. Make it so that you can accept a save file as a command line argument and load it.
6. Move onto the other functionality.

Organizing into modules by writing a function for each of the menu options 1-5 is a good idea. And include lower-level helper functions too. Modular code is clean code, and clean code is easy to debug. Moreover, creating modules by using functions lets you set up different layers of abstraction and break this big task down into smaller, more manageable tasks. You need to be able to go from a big picture to a bunch of... uhh... smaller ones, yeah. For example, in your main menu when you decide you want to remove completed items from your list, you could just have a function called remove_completed_items or something like that. Usually I start out by just splitting the problem into modules, writing the corresponding function headers, and then filling in the blanks in each of the functions.

Following this approach helps to:

1. Break the big nebulous task down into a smaller and more manageable series of tasks,
2. Write cleaner and more easily debuggable code,
3. Keep track of what you have left to do more easily (stay organized),
4. Reduce the amount of redesigning you need to do since you've considered the full picture

6 This assignment seems pretty open ended... What am I not allowed to do?

Most of the things you are not allowed to do should appear somewhere else in the document, but for the sake of consolidation, and clarity, you may **not**:

1. Write code in any files other than `todo.c` (which is not provided), and the provided C and H files.
2. Omit any of the C compiler flags that we mentioned in the Makefile section,
3. Dynamically allocate memory (e.g call malloc, calloc, realloc, etc.). If you're breaking this rule you will know.

Doing any of these things will result in **SEVERE** point deductions, so please, please, please don't do them.

7 Checking Your Solution

We have provided you with a grader script that you can use to check your work. It will supply some inputs to your program and make sure that you are printing the correct things to the screen in response. If you fail a test, you will be provided with the inputs, the expected outputs, and what your outputs were.

Important note! The grader, when providing you with diffs, will ONLY print out your program's `stdout` without the `stdin`! This means that the user input will not be included in the diff! Keep this in mind, because it may cause the input prompts to show up on the same line as headers since the user supplies the newlines when they type things in.

To run the autograder locally, follow the steps below depending upon your operating system:

- Mac/Linux Users:
  1. Navigate to the directory your homework is in. **In your terminal, not in your browser**
  2. Run the command `sudo chmod +x grade.sh`
  3. Now run `./grade.sh`
- Windows Users:
  1. On **docker quickstart**, navigate to the directory your homework is in
  2. Run `./grade.sh`

Note: 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.

8 Deliverables

Turn in the following files **only**:

1. Makefile
2. todo.c
3. todo_helpers.c
4. todo.h
5. todo_data.c
6. useful_strings.h

9 Rules and Regulations

9.1 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.

9.2 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.

9.3 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.

9.4 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

9.5 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.

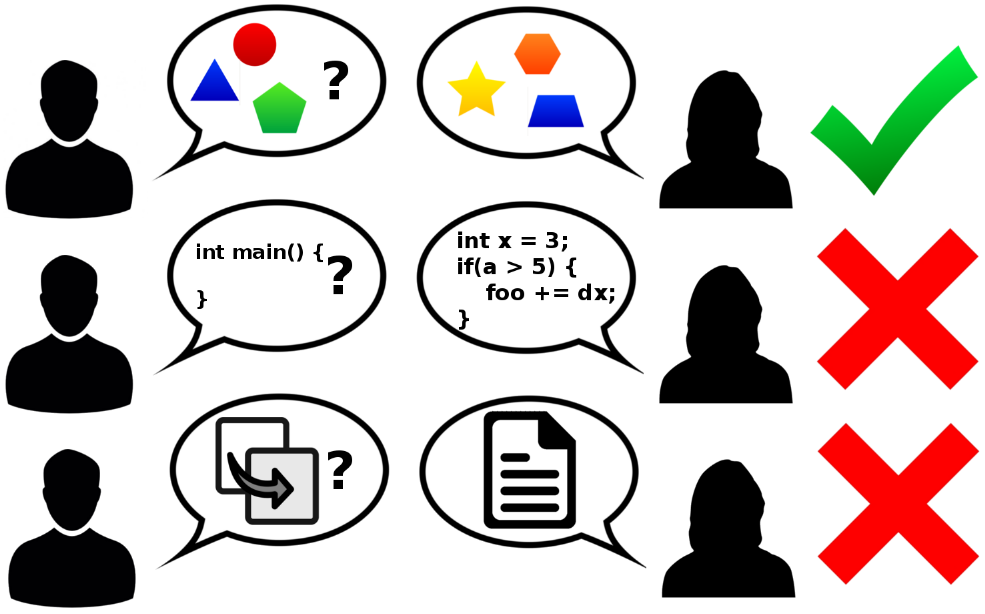

Figure 1: Collaboration rules, explained colorfully