Date due: Monday, February 7, 11:59:59 p.m.

1 Introduction

This is a relatively short first project intended primarily for you to get practice with the code development facilities that are different in this course from what you are used to in Java and Eclipse. This description is lengthy relative to the amount of code to be written, because the first project needs to explain the procedures used in all of the remaining ones. The actual description of the functions that you have to write is just around a page of this assignment.

Read the entire project assignment carefully before starting to code.

Students will be activated in the CMSC project submission server **around the end of this week**. Until then you will be able to write the project and test it yourself on the Grace systems, but not turn it in. Watch the **News feed** on ELMS to see when students have been activated in the submit server, which is when you will be able to submit.

To encourage (or require) you to write your own tests of your code, almost all the credit for this project will be for secret tests. There is only one public test, worth 5 points, which exists just so you can verify that your code compiles on the submit server. (There are cases where code can work perfectly on the Grace systems yet not even compile on the submit server.) Since there is only one public test, which only tests one thing, if you want to know whether your code has problems or not you will have to do what every professional who writes code as part of their job has to do to know whether their code is right, which is to write your own tests to check your code's correctness. If you aren't sure how to check your code yourself or to write your own tests you are welcome to ask in the TAs' office hours.

Keep in mind that your tests of your code are part of your coursework for this project, which must be done individually. Exchanging tests or test data, or writing student tests together, is cheating and will be referred to the Office of Student Conduct. (See Appendix D below.)

All projects in this course assume you set up your account correctly in discussion section shortly before this project is being assigned. As explained then, the command `check-account-setup` should tell you whether your account is set up correctly or not. And **before** you will be able to submit this project, you ***must*** upload the required PDFs for it and all of the other projects, as described in the handout linked to from the top of the **Projects** page on ELMS.

Be sure to carefully read the course project policies handout on ELMS. It describes in detail how projects are graded, which submission will be graded if you submit more than once, etc. Another handout describes the programming style that you should be using in projects, and how projects will be graded for style. Although this project is not being graded for style future projects will be, so you should **read that handout carefully as well**. Also **carefully read the constraints in Appendix C**, because you are **only** allowed to use certain C language features in writing the project.

Due to the size of the course it is not feasible for us to be able to provide project information or help via email/ELMS messages, so we will be unable to answer such questions. However you are welcome to ask any questions verbally during the TAs' office hours (or during, before, or after discussion section or lecture if time permits).

2 Development procedure

2.1 Obtaining the project files

The starting code for all projects in this course will be in a compressed tarfile; for this project the tarfile is named `project01.tgz`, and it is located in the public class directory `~/216public/project01`. The UNIX/Linux tape archiver program `tar` is used to save data on backup tapes, but it can also combine and store separate files into a single file called a tarfile, which is what we used it to do with the starting files for the project. A tarfile is analogous to a zipfile, which you are presumably familiar with, except tarfiles are generally UNIX–specific. The convention is that the filename extension `.tar` is used for tarfiles created by `tar`, and the tarfiles that, to save disk space, are also compressed (using a utility called `gzip`) have the extension `.tgz`.

- To obtain the project files log into `grace.umd.edu` and use the command `cd ~/216` at the prompt.

  This command will change your current directory to your course disk space. (When your account was set up, a symbolic link named `~/216` would have been created, assuming your account was properly set up, otherwise nothing written here is going to be able to work.) You **must** write all of your projects in this course in your course disk space, so this command will always be the first one to use when doing any work on a project.

- Then extract the files from the tarfile located in the public class directory using the command

```
tar -zxvf ~/216public/project01/project01.tgz
```

The `z` in `-zxvf` tells `tar` to uncompress the tarfile, `x` tells it to extract the files, `v` says to list the files as they are extracted, and `f` indicates that the next thing in the command is the name of the tarfile to operate upon. This will create a subdirectory in the current directory named `project01` that contains the following files for the project:

`functions.h`: This is a header file containing prototypes of the functions you need to write.

`functions.c`: This is a source file containing skeletons of the functions you need to write, which are described in Section 3 below. This is the only file that you should modify.

`public1.c`: This is the single public test for this project, which contains a `main()` function that calls the functions you will write in `functions.c`.

`.submit`: This is a hidden file that will allow you to submit your code to the CMSC project submission server.

Note that `tar -ztvf ~/216public/project01/project01.tgz` will list the files in the tarfile without actually extracting them, which is what the `t` instead of `x` in the command tells `tar` to do.

Once you have extracted the files from the tarfile once do not do it again later– that would just clobber whatever work you have done on the project by replacing it with the original tarfile contents. (If you think you need to extract the files from the tarfile again for any reason first talk with the TAs about it in their office hours, to find out of this is necessary, as well as for them to help you do it safely if it is actually needed.)

- Now use the command `cd project01` to change to the new subdirectory where the project files are. Use the text editor (e.g., Emacs) to edit `functions.c` and write the functions whose skeletons are in it.

If you happen to accidentally delete or make incorrect changes to your `functions.c` file, note that the UNIX tutorial explains how Emacs creates backup files that you can use to recover the most recent version.

2.2 Compiling your code and running it on the public test

After you have written the functions described in Section 3 here is how you can compile your code with the public test that is testing it (or with tests that you write). However, **before** you compile your code for the first time, **use the `cp` command to make a backup copy of it with a different name**, just for safety in the unlikely case you were to make any mistakes that would clobber the work you've done so far. (Appendix C discusses this further.)

To keep things simple for this project we won't use separate compilation or the make utility (both to be covered in class later; don't worry if you don't even know what these are yet). You can use the following command to compile your code for the one single public test: `gcc public1.c functions.c -o public1.x`

(In discussion section you should have seen a convenient way to run the compiler via Emacs, which you can use instead if you like.)

The option `-o public1.x` causes the compiler to use the filename `public1.x` for the executable version of the program (which will be created if there are no syntax errors in your code), rather than the name `a.out` that it uses by default. (You can use a different executable name if you want.)

Due to how you set up your account an alias for `gcc` is added that automatically includes in the compilation command the options below, which are required for projects in this course. If they aren't used your code may not work when submitted to the project submission server, even if it works fine when you compile and run it on the Grace systems.

`-ansi` causes the compiler to recognize only programs that follow the first standard version for C, originally published by the American National Standards Institute.

`-pedantic-errors` will cause the compiler to strictly follow the ANSI standard, and reject programs that do not follow it with errors rather than warnings.

`-Wall` turns on checking for many questionable programming constructs, so `gcc` will issue warnings for them.

`-fstack-protector-all` checks for one specific type of common error in C that is not checked by the above options, and is not explained further here, as it involves concepts that have not been covered yet.

`-Werror` causes the compiler to treat all warnings as errors.

To run the compiled executable program and check its results, just use its name as a command, so if you used the exact compilation command above, just type `public1.x` as a command. If your code is right and the test passes, "The test passed!" will be printed, while if your function was wrong you will see an error message about a failed assertion.

If you change your code and want to compile it again you can retype the compilation command in the second paragraph in this section, but if you press the up–arrow key it will cycle through previously–typed commands, and you can re–execute the compilation command when it is shown again by just pressing return.

If you want to compile your code with one of your own tests– suppose one that's named `student1.c` instead of `public1.c`– you can use a command like `gcc student1.c functions.c -o student1.x`, after which (assuming there are no syntax errors) you would just run `student1.x` as a command.

3 Project description

All you have to do is to write (in the file `functions.c`) the three fairly simple functions described below, which only require using basic C features: variables, expressions, conditional statements, and loops. They don't even need arrays. The functions just return values based on their parameters– they do not read any input, or produce any output. You can calculate their results however you want, except that, as Appendix C says, you **cannot use any C library functions in implementing your functions**. (Read that sentence again, and be sure not to forget it.)

3.1 `int classify_triangle(int side1, int side2, int side3)`

Triangles can be categorized based upon whether they have any sides that are of equal length (as well as any angles that are equal):

- A triangle that has three equal–length sides (therefore also having three equal angles) is called equilateral.
- A triangle that has two equal–length sides (therefore also having two equal angles) is called isosceles.
- A triangle that has no equal–length sides (therefore no equal angles either) is called scalene.

This function's three parameters represent the lengths of a triangle's three sides. (Since the parameter types are `int`, we are only concerned with triangles whose sides have lengths that are whole numbers.) The function should return:

- 3 if the triangle represented by the three side lengths is equilateral.
- 2 if the triangle represented by the three side lengths is isosceles.
- 0 if the triangle represented by the three side lengths is scalene.
- -1 if the triangle represented by the three side lengths is invalid, according to the following error cases:
  - The length of each side of a triangle must be strictly greater than zero. A negative–length dimension is impossible, and if one or more of the sides of a triangle had length zero you would have a line or a point, but not a triangle. So if any of the side lengths are zero or negative the triangle is invalid.
  - The lengths of the three sides of a triangle must satisfy the triangle inequality, which says that the sum of the lengths of any two sides of a triangle must be (strictly) greater than the length of the third side. For example, a triangle cannot have sides with lengths 1 inch, 2 inches, and 36 inches. So if this property does not hold for all of the side lengths the triangle is invalid.

Note that the return value for valid triangles represents the number of sides that have equal length, so the function can return 0, 2, or 3, or -1, so 1 is not a possible return value.

3.2 `long reverse_digits(long n)`

This function should return the number whose digits are the reverse of those of its parameter `n`. For example, if `n` is 321 it should return 123. `n` may have any integer value but the function should return -1 in these two error cases:

- We do not have a interpretation for what the reverse digit for of a negative number would be. For example, would the reverse of -123 be 321 or -321 or even $321-$ (which is not even a valid number)? Since we do not have a good answer we call this an error case and the function should just return -1 if `n` is negative.

- The reverse digit form of zero is just zero. However, we consider it to be an error case if `n` has more than one digit but its rightmost digit is zero, for example the number 3210. There is no way to form a number that has a leading zero, so 0123 cannot be returned. As a result, if `n` is **not zero itself** but the rightmost digit of `n` is zero then -1 should just be returned.

Your function is just supposed to return a value, and not perform any input or output. However, you may want to print its result in the process of testing or debugging. If so you will have to use the format specifier `%ld` to print a value of type `long`, which is the return type of the function.

As you are writing your own tests of your function, note that the largest positive `long` on the Grace machines is 9,223,372,036,854,775,807 so you cannot pass any argument to the function that is larger than this. Also keep in mind from Chapter 3 in the text (as will be covered in discussion section shortly after this project is being assigned) that leading zeros in an integer constant cause it to be in octal. If you forget this and pass in a number like 0123 the results will likely not be what you expect. (Just avoid using any arguments with leading zeros when testing your function, except for the number zero itself.)

3.3 `unsigned long catalan(short int n)`

This function should return the value of the `n`'th Catalan number, where Catalan numbers can be computed in different ways, but here is a straightforward one:

$$catalan(n) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ \dfrac{2 \times (2n-1) \times catalan(n-1)}{n+1} & \text{for any } n \text{ greater than } 0 \end{cases}$$

For example, $catalan(1)$ (the second Catalan number) is also 1, $catalan(2)$ is 2, $catalan(3)$ is 5, and $catalan(4)$ is 14. Catalan numbers come up in many combinatorics problems.

Note that Catalan numbers can be computed recursively, based on the definition above, but the definition above can also be used to compute them iteratively as well.

The Catalan number sequence starts at 0. It is not defined for negative numbers. So if `n` is negative the function should just return -1 to indicate that this is an error case.

Because Catalan numbers grow large quickly, there is a limit to how large you can calculate them, without using special techniques. And if you try to call your function on arguments that are sufficiently large you will get an incorrect result. For example, $catalan(34)$ is 285,894,211,472,315,003 (and if you don't believe me you can calculate it by hand), but when I call my function to compute $catalan(35)$ the result returned is 71,108,695,437,787,977, which is wrong. We will discuss in lecture shortly why this occurs. But you do not need to worry about (or test yourself) what results your function produces for any arguments that would cause such behavior, because we will just not call your function on any values of `n` larger than 34. Note that the format specifier `%lu` prints a value of type `unsigned long`.

Remember, you **cannot use any C library functions at all in writing your functions for this project**.

A Submitting your code

Remember that you will not be able to submit until around the end of the week. Watch the **News feed** to know when it is possible to submit.

Before you submit your code you must first make sure that it passes the one and only public test, by compiling and running it yourself, as explained in Section 2.2 above. After that, submit using the single command `submit` in your `project01` directory. This will prompt you for your UMD directory ID (**not** your nine–digit UID) and password, send your code to the CMSC project submission server, and inform you if the submission succeeded– which only means that your code was successfully received by the submit server, **not** whether it worked right there. You **must** then log into the submit server at https://submit.cs.umd.edu/spring2022 (also linked to from ELMS, under **Pages** → **Administrative and resources**), and check whether your program worked right on the public test there also. Some notes:

- If you are trying to submit but the submission fails with a message about your ID or password being wrong, **and** it is after the **News feed** says that students have been activated in the submit server, then try logging into the submit server by clicking on the link above:

  - If you can't submit and **cannot** log into the submit server at that link via your web browser, and you only recently added the course or only recently uploaded the required project PDFs, then just wait a day and try again. I will check every day or so and activate students who have successfully uploaded the PDFs by then. (Note that the TAs cannot activate you in the submit server.)
  - If you can't submit and **cannot** log into the submit server via your web browser, and you did **not** add the course recently, then you have probably not been activated in the submit server due to not uploading the required project PDFs, or doing it wrong. Carefully read (or reread) the description of the procedure that you have to follow on the ELMS **Projects** page, and either follow it if you didn't do it, or fix any mistakes if you did things wrong, and just try again the next day.
  - If you can't submit and **can** log into the submit server at that link via your web browser, then something could be wrong in your account setup (the configuration that was done during discussion section). Run `check-account-setup` and come to the TAs' office hours for help if you can't fix any problems that it identifies on your own.

    (If you are trying to submit and get a message that ends with the line "You did not provide enough arguments." then your account setup is definitely wrong.)

- If you can submit and your code compiles on Grace but does not on the submit server, you may have changed the header file `functions.h` which you were not supposed to do, or something in your account setup may be wrong.
- Do not wait until the last minute to submit your program. The submission server enforces deadlines strictly. Even one second beyond the time at the top of the first page of this assignment means that your project is late. You are urged to finish projects **early**. Finishing at least one day before its deadline will allow time to reread the project requirements, and ensure you have not missed anything that could cause you to lose credit, or to make any necessary changes if you have.

  Illness does not justify project extensions unless it is **extremely severe**, and lasts most or all of the time that a project is assigned (see the course syllabus).
- If this is your first CMSC programming course here and you aren't familiar with the submit server, you can ask in the TAs' office hours for help understanding it, and what information it's showing.
- Do **not** submit projects using the submit server's mechanism for uploading a zipfile or individual files. You must use the `submit` command to submit, or for reasons described in the project policies handout mentioned above you may end up losing significant credit on the project.
- Do **not** make unnecessary submissions of the project, or you may **lose credit**. (See the project policies handout on ELMS for details.) To avoid losing credit, do not submit this project more than five times. Suggestion: reread this entire assignment **before** submitting, to ensure you didn't miss anything important.
- Do **not** use the submit server to keep backups of your project code. Having a large number of unnecessary submissions just slows the submit server down for everyone. Instead use UNIX commands (see Appendix C below) to frequently make copies of your project files, with different filenames or in different directories. Similarly, do **not** use the submit server to check if your program compiles, or to check its results on the public test. You should be compiling and testing your code yourself, as described above, before submitting.

B Grading criteria

Your grade for this project will be based on:

public tests	5 points
secret tests	95 points

Since secret tests are 95% of your score for the project, be sure to test your functions carefully on special/edge cases.

Secret tests and their results will not available until the project grading has been finished. The single public test is obviously not comprehensive, and you will need to do testing on your own to ensure the correctness of your functions.

C Other notes

- You will **lose credit** if you use any features of C that are not allowed in writing the project:
  - You may use the features of C that have been covered **so far in class this semester before arrays were covered**,

    (Even if C features are covered in the textbook you cannot use them unless they were covered in class, except for what is allowed by the next two items.)
  - You may use any C operators covered in Sections 5.1.4 through 5.1.8 of the Reek text– these are mostly the same as in Java so they will not be discussed in detail in lecture except where they differ from Java.
  - You may use any C features in Chapter 4 of the textbook, most of which will also be skipped in class because they are also similar to Java.

    The one exception is that you **cannot** use the `goto` statement mentioned in Chapter 4.
  - **You may not use any C library functions at all in writing your code.** The project can be written using only the features of C covered so far, without needing any C library functions.
- Your code for this project must all be in the single file named `functions.c`, otherwise it will not compile when submitted, so do not rename it or split it up. Do not add any new header files to the project. Also do not move `functions.c` elsewhere– it must remain in the `project01` directory for your code to work on the submit server.
- If you get the error "Segmentation fault" when running your program this means it had a fatal error. For the moment, the best way to debug errors is to just add debug `printf()` statements to your code, to print the values of the variables that are being used. Note that for technical reasons to be discussed later in class, the last thing printed by every debug `printf()` should be a newline, to ensure its output appears correctly.
- Besides the public test, you should write your own tests for your code! If you have a problem or bug and have to come to the TAs' office hours, you **must** come with tests that illustrate the problem (meaning you must write tests of your own, besides the public test). In particular you will need to show the smallest test you were able to write that illustrates the problem, so whatever the cause is can be narrowed down as much as possible before the TAs even start helping you. (The submit server does not run your own tests for C projects, as it may have in Java projects in the prior courses.)

  But don't modify the public test, otherwise you won't be testing your code on the cases that the submit server is running. Just create your own tests.
- Note that the project policies handout says that all your projects must work on **at least half of the public tests** (by the end of the semester) in order for you to be eligible to pass the course. (See the project policies handout for full details.) Since there is only one public test for this project, this means that your code just has to pass it, which should be easy. But since the public test is worth only 5 points you won't get a good score on the project unless you test your functions yourself to make sure they work right in other cases.
- As the UNIX tutorial discusses, a wireless network can be slow when running a graphical program like Emacs. If you are able to get a wired Ethernet cable and do work with your computer connected to an Ethernet jack (either a wall jack or one on the back of your wireless router), things will be much faster. On a slow connection, "`emacs -nw`" will be much faster (see the tutorial).
- If you happen to be familiar with them, don't use nano or pico for editing programs. These are very small and limited text editors intended for editing short emails. There are several full–featured text editors available on Grace. Emacs is generally similar to nano and pico, except it's far more powerful.
- It's possible for a program to work fine on one machine (like the Grace machines), but not work at all on another machine (like the submit server), and this may occur more frequently in C, due to the nature of the language. Consequently you **must** log into the submit server and see whether your code worked there after you submit it.
- If your code is passing tests on Grace but not on the submit server, the first thing to do is to **check whether you are using any uninitialized variables**, which is the most common error that is made in this class.

- Keep backup copies of your project and any other important files (like your own tests) in a different directory in your course disk space than where your project is. It's a good idea to save backup copies every time you log in or log out, if not more frequently. Recall from the first discussion sections and the UNIX tutorial that the `-r` option to `cp` will make a copy of a whole directory and all its contents, so a command like `cp -r project01 project01.bak` would make a new directory containing a copy of everything in the `project01` directory (this command assumes you are located above the `project01` directory, e.g., `cd ~/216` first).

  Furthermore, it's a good practice to save your work every few minutes in Emacs, so that an unexpected power outage or lost network connection would only cause a small amount of work to be lost.

D Academic integrity

Please **carefully read** the academic honesty section of the syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, publicly providing others access to your project code online, or unauthorized use of computer accounts, **will be submitted** to the Office of Student Conduct, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand the academic integrity requirements in the syllabus. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. More information is in the course syllabus– please review it now.

The academic integrity requirements also apply to any student tests for projects, which must be **your own original work**. Sharing student tests or test data, or working together to write them, is prohibited.