Name: Zachary Lowery

Date: 10/17/2017

Instructor: Cao (CS371)

GroupWon Coding Conventions

Introduction

This simple document is meant to be an outline for how this team expects every individual team member to code, and how we expect the code to be documented. If you have any questions about why the conventions are the way they are, or to protest, consult the Librarian (Zachary Lowery) and the Lead Programmer (Jordan Leibman). Coding conventions are not to be changed without the Lead Programmer's express approval. Write permissions to this document are exclusively granted to the Librarian. If the Lead Programmer would like to make a change to these conventions, simply ask the Librarian to edit this document. If the Librarian finds that this document differs from the one found on GitHub, he will overwrite the document there with a copy on his local machine and harbor ill will and other notable biases against said individual while he documents their progress. You have been warned.

"Editor" Level Conventions

The team members may use whatever editor best suits them so long as:

1. The files they generate are compatible with the other editors the team members are utilizing.
2. They support substituting tab characters for a given number of space characters (currently 3).
3. They don't inject mysterious characters into the document during the saving process that causes interference with our operations.

Directory Level Conventions

It is imperative that the workspace is kept clean. Ideally no directory should have more than ten children at any given time, if they do make efforts to reorganize in such a way that the directory structure is both concise and self-explanatory, i.e. clean up the mess without making things hard to understand for your teammates. The Librarian will regularly check the status of the repository to make sure nothing terrible is happening. So be aware where you place files one moment may change abruptly in the future. Here is a hint, if you clean up after yourself, I will not have to do it

for you and you won't have me getting my grubby hands all over your part of the project. I still reserve the right to move stuff around for the sake of organization, but I am unlikely to do it unless I feel it is necessary. Here are some general rules:

1. The root directory of the repository will have a "Documentation" folder, a "bin" folder, and a "dev" folder.
    a. The "Documentation" folder will include any and all documentation that isn't in the source files themselves. This includes manuals, team policies, coding conventions, mandatory reading, helpful resources, and more. Refer to it often. If you do not read mandatory reading documents and I notice going across your code I will notify the Lead Programmer and document accordingly.

    b. The "bin" folder will include all "release" code that has been approved and thoroughly tested. It is from this folder that we will build executables, both for general use and for showcasing during "demo" days. This folder will include a subdirectory called "Presentation" which we will use for all materials that directly relate to the presentation of the material during the aforementioned demo days.

    c. The "dev" folder will include all code that is a "work in progress". This includes prototypes. Point 2 of the parent list (see below) will expound on organization rules *for any directory*. It is especially important to follow them here. For code (or any other resource) to be moved out of the "dev" folder into the "bin" folder, the resource must be granted approval by the Lead Programmer (for quality control), the Librarian (for documentation quality control) and the person who is most knowledgeable/qualified to evaluate that particular subject. If that person happens to be the Librarian he must not only gain the approval of the Lead Programmer but also the next most knowledgeable person. The Lead Programmer is not obligated to follow these restrictions, but must at least notify the Librarian that he has promoted a "dev" resource into a "bin" resource, so the Librarian can evaluate and document accordingly. If anybody does not follow these rules they will be written up for trying to push development code into release code.

2. These are general rules to apply for any and all directories, I understand if this slips a little during hectic times during the development cycle. But I will (and must) move things around to encourage a productive, healthy work environment. If I believe your habits are too chaotic I'll approach you in private and address the issue, in the worst case if this is a habitual problem I will consult the Lead Programmer and document accordingly.
    a. Any directory is not to have more than ten children, this excludes directories. So a parent directory may not have more than ten files in it at any given time. If your workspace is becoming cluttered, consider omitting unnecessary file types from GitHub (such as unnecessary log files). If that approach doesn't work, reorganizing the directory must occur. I will leave this up to your discretion, I

only ask that you organize it in such a way that other team members know what you are thinking.

b. At the root of every directory you will add a mandatory README file, which you will edit to include a brief summary of the directory, its files and their contents, and their sub-directories and their contents. I will regularly visit these directories and edit them accordingly. I don't expect masterpieces of literature and classical antiquity, but I do expect that they are verbose enough so that your team mates know what is going on. If I think the content in your README files is lacking, or I hear team mates complaining about your documentation, I will stare at you into you get very uncomfortable, that or I will just take you aside and ask you to improve your documentation. If I find that you never include README files in the first place (or they are empty for more than 72 hours) I will consult the Lead Programmer and document accordingly.

c. If you build an executable using the children of a directory, you will place it in that directory. If you build an executable using multiple files from multiple sub-directories, you will place it besides the highest level directory you utilized (as its sibling). You will never, EVER place an executable below the source files you used in the directory tree. If I find this I will call the Lead Programmer, notify him of your tragic untimely death, purchase some fava beans and dine on your liver as some questionable homage to Silence of the Lambs. Writing you up will not be necessary. But I'll do it anyway and consult the Lead Programmer.

Code Level Conventions

Ah, the real stuff. What we've been waiting to get too all this time. This section will proceed following this template:

I. General Practices
II. File Level Practices
III. Global Level Practices
IV. Function Level Practices
V. Local (Block) Level Practices
VI. Notes on Documentation
VII. Miscellaneous

General Practices

In this section we will cover what the team expects other team member's code to look like, and how that pans out in terms of variable names, indentation styles, bracket use, etc.

- You are to use C style comments (`/*`, `*/`) for verbose explanations, documentation headers, descriptions, etc.
- You are to use C++ style inline comments (`//`) for quick quips of information regarding a module, variable name, or any other small matter in your code. Do NOT use long strings of inline comments as a substitute for C style comments. Especially to replace document headers.
- All variable names are to use camel-case, where the first letter of the first word is lower case, while the first letter of subsequent concatenated words are upper case, i.e. (`camelCaseForTheWin`) or (`#lifeHack`).
    - The exception to this is `const` variables, which are to be all uppercase. (i.e. `const int ROOM_SIZE = 256`).
        - Take every opportunity you can to eliminate *magic numbers*, numbers by themselves that aren't tied to a variable. Do this using the `const` keyword in conjunction with the appropriate type (usually `int`).
    - If a variable is a pointer to something that is NOT AN OBJECT INSTANCE, it should be prefixed with '`p_`', i.e. (`const int *p_deathToll`).
- Take time to think on what you will call variables, make sure you pick a name that makes sense and tells you and other team members down the line what is going on with your code. It is better to spend five minutes fretting over a name now than 50 minutes wondering, "What the hell was I thinking?", your team members will be wondering something similar, "Why won't [put your name here] just go die?".

File Level Practices

In this section we will expand on what conventions should be used by the team when dealing with file level operations, such as the use of build files (make, file parsers, etc.)

- Variables aren't the only important things that need good names, files do too. Make sure your file has a title which won't cause your whole team to track you down, give you a chat and hold your memorial service/funeral at the same time.
- Before you decide to add/commit a file you used an unapproved file parser on, make sure you get the approval of the Lead Programmer first. If the file parser is approved and it still has potentially significant negative consequences, consult the Lead Programmer.
- Keep unnecessary clutter out of the repository. If your team mates don't need it and you don't need it, just keep a copy on your local machine.

- All source files should have a documentation header, enclosed using C style comments. The documentation header should be composed of seven parts:
    - Name, this honor goes to the one who first created and pushed the file (provided it had useful content in it.)
    - Date, the date the file was first pushed to the repository.
    - Description, what the file is supposed to do, how it tries to do it, and what still needs to be done.
    - TODO, a more verbose version of what is found in the "what needs to be done" in the description section.
    - Implications, how this file fits into the broader scheme of things, and how you plan to go around implementing it.
    - Conclusions, stating a brief summary of the above three sections.
    - LIBRARIAN, anything you want to tell me or have questions about in regards to the file itself, as long as it pertains to the file in question any question is fair game. If you write hate mail here I suggest you find some clever way to tie it to the file or I will delete it.
- I expect everyone to read enough about Git so that they know how to recover lost files in case a certain copy of a repository goes south. Please refer to the free book on Git here: https://git-scm.com/book/en/v2

Global Level Practices

I hate global variables, I hope you do too. But just in case you don't I'm warning you that it's required that you consult the Lead Programmer before you use a global variable in any file, period. No, I'm serious. In addition you must include a description header on why you believe its inclusion is necessary and what we hope to gain by using it; and why it is worth it. If you fail to convince the Lead Programmer it will have to jump off the nearest '}'.

Secondly, avoid polluting the "namespace" as it is called in C++, we don't want to restrict ourselves by implicitly declaring the presence of tens of thousands of functions about which we know nothing (including their names, which is why it is so dangerous).

Function Level Practices

Functions are the bread and butter of the source files we encounter, so we better get this part right.

- Functions must include a description header that states a number of things.
    - Name: Who made this function?
    - Date: When was this function created?
    - Description: What does the function do?
    - Pre-Condition: What does the function need the programmer to ensure/do before calling the function?
    - Post-Condition: What happens after the function is called?

    - Running Time: What order O(n) are we dealing with here?
        - The person who did the most work on the function should calculate running time, anyone who works on it subsequently should make sure they either improve the running time or maintain it without changing the Pre-Condition (ideally).
    - LIBRARIAN: Tell me what's up.
- Functions should have a definition that looks something like this [accessLevel][returnType][functionName][functionParameterList].
    - i.e. private int sirHurtsAlot(int ooh, char* scream, bool dead).
    - Try to think long and hard about the order the parameters in the parameter list should be sequenced in, it will be easier to use as a result.
        - When you use a variable that is a pointer, type it in the list as such: [variableType]* p_[variableName] (param1, char* p_screamString).