17/10/23

COS 214 PROJECT RESTAURANT SIMULATOR STANDARDS

OFENTSE RAMOTHIBE

BOLOKA KGOPODITHATE

NTANDO MAZIBUKO

ENNIS MAPHASHA

REALEBOGA MOATSHE

THASHLENE MOODLEY

TABLE OF CONTENTS

INTRODUCTION

The goal of these guidelines is to create uniform coding habits among our team members so that reading, checking, and maintaining code written by different persons within the group becomes easier. The intent of these standards is to define a natural style and consistency, while leaving room for creativity and improvement in aspects not discussed in this document.

By doing this, we can easily understand and intuitively build on the code done by a team member. This way, code-walk throughs become less painful for each person involved in the development of the program.

PROJECT DEPENDENT STANDARDS

The standards and guidelines described in this document were selected on the basis of common coding practices of our team members and from the C++ standard documents collected on the Internet as this is the programming language we will be using to build our program. The coding practices that we will focus on standardizing are:

1. Naming conventions
2. File structure
3. Specifications for Error Handling
4. Revision and Version Control
5. Guidelines for the use of Git and any other programming tools we might use

FILE STRUCTURE

Each class will have its own .cpp file that describes the interface for the class as well as its implementation. This means, the declaration and implementation of each class, along with its variables and methods will all be located in one file.

The reason for choosing this approach is to simplify the process of locating files in our system, function prototypes, global data types, constants and data, and more importantly, minimizing the number of files in our project for easier management.

The order in which these aspects are going to appear in the file is as follows:

- include paths
- class declaration/definition
  - defining variables
  - defining methods
- implementation of methods

All files shall exist in one directory.

As for file naming, we will follow the camel-case naming convention.

NAMING CONVENTIONS

Select Clear and Meaningful Names

The most important consideration in naming a variable and methods is that the name fully and accurately describes the entity or action that it represents. Complete and meaningful names make the code more readable and minimizes the need for comments.

Naming of methods (verb)

Names of methods shall consist of a verb. This will make the action and purpose of the method clear.

Naming Constants, variables (noun)

Names of constants, variables, and functions shall be nouns. This makes it logically easy to link the variable name to what it represents.

COMMENTS

Comment Block Standard

Code is more readable when comments are presented in paragraph form prior to a block of code, rather than inline comments for a line or two of code. Comments should be preceded by and followed by a single blank line.

- Indent block comments to the same level as the block being described.

- Highlight the comment block in a manner that clearly distinguishes it from the code

We will use the following approach:

```
/*
============================
comment…
comment…
comment…
============================
*/
```

External and internal comments

It is recommended to write a comment block for each method prior to the implementation of that method. This comment block will be located immediately outside our method, just one line above the function definition.

To mitigate the use of inline comments, we will use internal comments. These will be used in cases where we need to explain the logic of our code that spans at least 15 lines, describing the action of the code that follows. This will make the code easier to read and will form a consistent code description that will be easier to verify and maintain as the code is modified.

This will also make a first pass reading of the code much easier than reading the code alone, no matter how self-descriptive the code is.

We will reserve inline comments for counterintuitive logic that we might need to integrate in our implementation. The description will be tightly linked to that line of code that needs further description.

INDENTATION

A consistent use of indentation makes code more readable and errors easier to detect. 3 spaces is recommended per indent, but this will largely depend on the settings of the IDE each member uses. Statements that affect a block of code (i.e., more than one line of code) must be separated from the block in a way that clearly indicates the code it affects.

STRUCTURED PROGRAMMING

One statement per line

There shall be no more than one statement per line. When a single operation or expression is broken over several lines, break it between high-level components of the structure, not in the middle of a sub-component.

Example:

```
if(condition)
{
        //code
}
else
{
        //code
}
```

Brackets, Begin...End, and Delimiting Control Blocks

We will use the following notation:

```
if(true)
{
        //code
}
```

Mixing bracket attachment (one part of the conditional, one part of the functional block) is discouraged.

Example:

```
if(true) {
        //code
}
```

GIT STANDARDS

Workflow

To avoid merge conflicts and to allow for more concise code reviews, develop incrementally, do not make large, elaborate changes to code to allow for easier reverts if conflicts arise.

Commit size

To allow for faster reviews and testing, when attempting to batch fix code, make atomic commits, so unintended interactions can be identified and reverted faster and easier.

Branches

All work done will be on branches so changes made do not have any effect on the main code before the code has been reviewed and tested, where it will then be merged to the main branch.

Commit messages

Provide the context of the commit as it relates to the changes made to make understanding the changes easier.

Descriptive commit messages allow for faster review of merge code in the case of conflicts or commit requests describe changes and the purpose of the coming prepended by a commit type eg.

- feat – a new feature is introduced with the changes.
- fix – a bug fix has occurred.
- chore – changes that do not relate to a fix or feature and don't modify src or test files (for example updating dependencies)
- refactor – refactored code that neither fixes a bug nor adds a feature.
- docs – updates to documentation such as the README or other markdown files
- style – changes that do not affect the meaning of the code, likely related to code formatting such as white-space, missing semi-colons, and so on.
- test – including new or correcting previous tests.
- perf – performance improvements
- ci – continuous integration related.
- build – changes that affect the build system or external dependencies.
- revert – reverts a previous commit.

Branching strategy

Each member will be developing on their own branch, this will allow conflicts, and unexpected behavior to be identified and reverted easier before it affects the main branch.

USE OF GLOBAL VARIABLES

Global data will only be used in cases when necessary. However, in general, they are to be avoided. This helps with managing errors and controlling access to variables to ensure the variable is not tempered with when it should not.

ERROR & EXCEPTION HANDLING

Any errors that could potentially occur must be handled. Functions that can fail should always return a success or error. Error recovery should be handled in the class that is responsible for the domain in which the error occurs. Also, the manner in which the error will be handled is such that the program can continue running until completion/ final state.