Personal Coding Style Standard

Ugnius Teišerskis

1st year Software Engineering student in Vilnius University.
1st group, 2nd subgroup

Version 1.0

2024.11.12

Abstract

This document defines the personal code standard followed in the 4th homework assignment in the Procedural Programing course.

Contents

1 Introduction

This document outlines my personal coding conventions. These conventions are used to structure, format, and document code in a way that enhances readability and consistency. The aim is to make the code more maintainable and easier to understand for myself and others who might work on it in the future. The conventions include rules for code alignment, variable naming, commenting, and overall code structure. Each point is immediately followed by an example which showcases the how each rule is applied.

2 Code Alignment and Formatting

2.1 Indentation

- All code blocks are indented by a tab.

```
if(isTxtFile(fileName)){
    dataFile = fopen(fileName, "r");
    if(dataFile != NULL){
        fileOpened = 1;
    }
}
```

2.2 Braces

- Opening braces must be placed on the same line as the control statement.
- Closing braces must be aligned with the control statement, not the block of code.

```
if(bytesRead == 0){
    return 0;
}
```

3 Naming Conventions

3.1 Variable Names

- Variable names must be written in camelCase, starting with a lowercase letter.
- Use descriptive names that clearly indicate the purpose of the variable.

```
unsigned numParticipants = 0;
```

3.2 Function Names

- Function names should also use camelCase and start with a verb to represent actions.
- Function names should be clear and concise.

```
void deleteTopParticipant(List *list);
```

3.3 Constants

- Constants should be written in uppercase letters with underscores separating words.

```
#define BUFFER_SIZE 1024
#define MAX_NAME_LENGTH 30
#define MAX_PARTICIPANTS 1000
```

4 Commenting Code

4.1 General Rule

- Comments explain what is done in some places, where the code is complex and may require more attention to understand.

```
// ensuring the buffer ends at '\n' or EOF
if(bytesRead == BUFFER_SIZE - 1){
    int backtrackIndex = bytesRead - 1;

    while(backtrackIndex >= 0 && buffer[backtrackIndex] != '\n'){
        --backtrackIndex;
    }

    if(backtrackIndex >= 0){
        fseek(file, backtrackIndex - bytesRead + 1, SEEK_CUR);
        buffer[backtrackIndex] = '\0';
    }
}
else{
    buffer[bytesRead] = '\0';
}
```

5 Code Structure

5.1 File Organization

- Each module (such as input handling, linked lists, etc.) should have its own header and source files. Personal header files are included using the following format: "name.h", while the C standard library ones are included following the usual format.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "linked_list.h"
#include "messages.h"
#include "input_validation.h"
```

5.2 Function Size

- Large functions should be split into smaller, more manageable sub-functions.

```
void extractNamesAndScores(char *buffer, List *list, unsigned *numParticipants){
    char *line = strtok(buffer, "\n");

    while(line != NULL){
        char name[MAX_NAME_LENGTH];
        double score;

        if(sscanf(line, "%s %lf", name, &score) == 2){
            insertElement(list, createParticipant(name, score), *numParticipants
            ++(*numParticipants);
        }

        if(*numParticipants >= MAX_PARTICIPANTS){
            return;
        }

        line = strtok(NULL, "\n");
    }
}
```

5.3 Control Flow

- Use early returns to make the code easier to understand.

```
if(bytesRead == 0){
    return 0;
}
```