CPSC 3720 SPRING 2018

Code:Blocks Plugin: Implementation File Generator

The Remaining Jerry's
Project Group E

Jace Riehl
Nathan Tipper
Vincent Cote

February 2nd, 2018

Table of Contents

Introduction

Welcome to the Implementation File Generator plugin for CodeBlocks developed by The Remaining Jerry’s. This plugin will help the development process through code generation that is required by any developer in any project.

Proposal

The idea behind the Implementation File Generator plugin is that we would like to simplify the class creation process. When a developer is creating a class, we start by making the interface for the class i.e. the header file. Once a header file is made, we have to go through the tedious process of rewriting all the signatures of our prototypes and then implementing the functionality of those functions. What the Implementation File Generator would do for us is the following:

- Create a .cc file with the same name as the header file and save it to your /src folder or a designated folder given by the user.
- Include the header file from which the code was generated.
- Create all signatures within the header file with braces for the implementation and, if there is a return type or no variables names provided, create a return statement with a default value and default argument names.

Now the user no longer has to create an implementation file again. They simply need to create the implementation and configure their arguments and return statements to match their implementation. Look at figure 1.1 and 1.2 for reference of the idea.

Project Management

Team Organization

All none code related documentation is to be in pdf format in order to keep a set standard for all team members. As this is a school project and is used as a learning tool for all team members, we understand that we are all designers, programmers, leaders and documentation gurus. With that being said the group is still structured with each member having a specific role. These roles are a guideline and will help the team know who to approach with specific issues concerning the different aspects of the project. The specific roles are as follows:

- **Design Lead:** Nathan Tipper
- **Repository Admin:** Vincent Cote
- **Kanban Sensei:** Jace Riehl

Risk Management

With any project, we need to assess the risks involved in creating the project. The Remaining Jerry's have identified the following risks and the solutions to those risks if they arise:

- **Feature creep** - With a project that is so openly designed with no limitations other than time, it is easy to design a plug in that cannot be made within the allocated time. Therefore, throughout the development process, we will build modular code which can be built upon if time is permitted. We will reserve the right to extend the Implementation File Generator, allowing for other modules which add functionality for users to explore for the projects.
- **Lack of experience -** Throughout our school experience our assignments have been focused on small programs that process a relatively simple task. Also, these assignments only required our knowledge of C++ to complete. Integrating functionality to an already existing application is a task we have not looked at yet. Therefore, a great deal of research is needed to complete this project. We will need to assess what the design constraints are early so we don't run into any implementation problems later in the project.
- **Busy schedules -** All members of the team are taking classes which require a lot of attention. Therefore, coordination will be key to success throughout the development process. Slack, and weekly meetings will need to well structured in order to keep workflow positive and moving in the right direction.

Development Process

Coding Conventions

Throughout the lifetime of this project, our team has decided to follow the following coding conventions:

- **Starting blocks on the same line of any scope.**
    - Ie :

```
Class Class1 {
   …
};
```

- **Using camelCase for both local variables (attributes) and functions (members). For member variables that are private, we will use put a underscore before the variable to indicate it is private data.**

- Ie:

  ```
  int thisIsAVariable = 1;
  void thisIsAFunction() {
          ...
  }

  int _thisIsAPrivateVariable;
  ```

- **Class names are to be capitalized and then use camelCase**

  - Ie:

  ```
  Class ThisIsAClassThatDoesNotFollowExistingConventions { ... } ;
  ```

- **Variables, functions, and class names should be clear and concise and be named with respect to their function.**

  - Example:

  ```
  int numberOfJerrys = 3;
  ```

- **Comments should be short and give a broad overview, leaving the variable, function, and class names to speak to the functionality of the code.**

- **All nested scopes should be indented a level in for a readability.**

- **Each header file should start with a ifndef, def statement to ensure header files are not duplicated throughout the code. The defined statement should be read as the name of the class in all capital letter followed by "_H."**

  - Example:

  ```
  #ifndef SOMECLASS_H
  #define SOMECLASS_H

  class SomeClass { ... };

  #endif // SOMECLASS_H
  ```

- **Use accessor methods for users to use instead of public attributes.**

- **Documentation of all members in header files is recorded and is required to have all of the following:**
    - ❖ What the function does.
    - ❖ Description of parameters and what the return value is.
    - ❖ How the function modifies the object.
    - ❖ The pre-condition and post-conditions of the function.
    - ❖ Any restrictions the function may have.

Requirements

Essential

- There is a button which will look at a header and generate the .cpp and all signatures are generated with scope operators and the header file included.
- The functions auto-created to the .cpp files have the correct return type with default values
- An abstract method is not generated in the .cpp file
- If changes have been made, make sure you it checks with the user to override the current .cpp file.

Desirable

- New functions will be added to the .cpp file automatically after a each function is added to the .h file (without hitting a "run" button)
- If the button is pressed after the .cpp file is created it will generate the new functions and delete the old not listed functions
- Add a wizard to set up where to locate the .h and and where to put .cpp files for different file structure in a project

Optional

- Add refactor tool to rename functions across the entire project
- As private variables are typed, ask user if they would like to add a getter and a setter for the variable
- Ask if the user wants to add initialization list for their private

Procedures for Configuration Management

Code Review Process

Every member of the team is involved in the Code Review process. Each member will have a branch for the work that has been assigned to them. When a member has completed the work, they will create a merge request to merge their work back into the master branch. All other members will be notified and any other member is responsible for the review of the code and the merging process. The philosophy behind this is that the member who created the merge request is not able to merge it back into the master. It must be reviewed by another member and that member can choose to either merge back if they think the work is complete and all guidelines are followed, or they may choose to leave a comment that it should be fixed in some way or ask for another member to review it as well.

Communication Tools

The primary methods of communication for the duration of the project will be face to face meeting as well as using slack for text based communication regarding general information, random information, design documentations, structure and repository documentation; all of which are subdivided into their own channels. If face to face meetings are not feasible due to scheduling conflicts the team is to arrange a video conference call on Discord at the earliest opportunity.

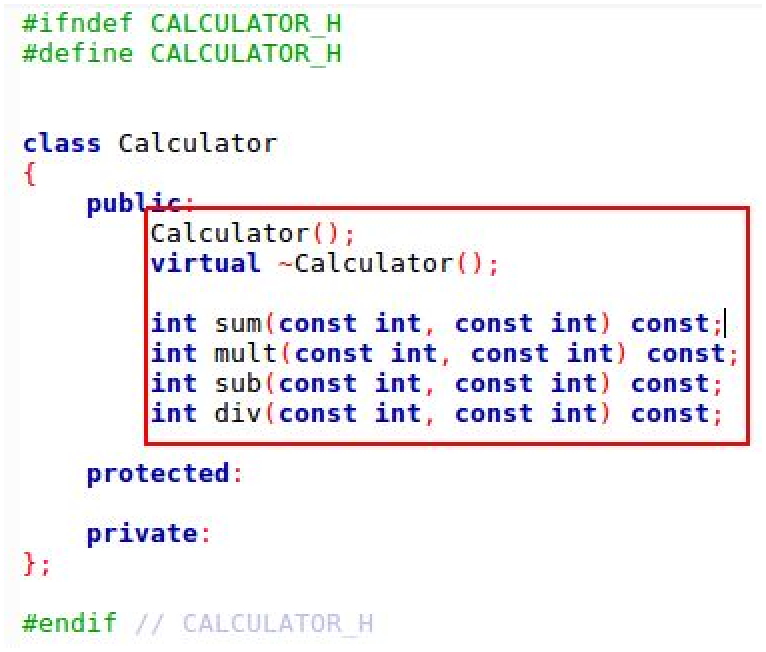
Change Management

Any group member will be able to issue bug reports that they encounter from the program through GitLab. When a bug is found, the reviewer who identified the bug(s) will assign them to the developer who wrote the function/class. Only the reviewer who identified the specific bug is allowed to close it. Once the bug has been resolved, the developer is to report back to the reviewer who will test it again until it is agreed that it is fixed or that it is more of a feature than a bug; at which point the issue will be closed through GitLab.

Appendices

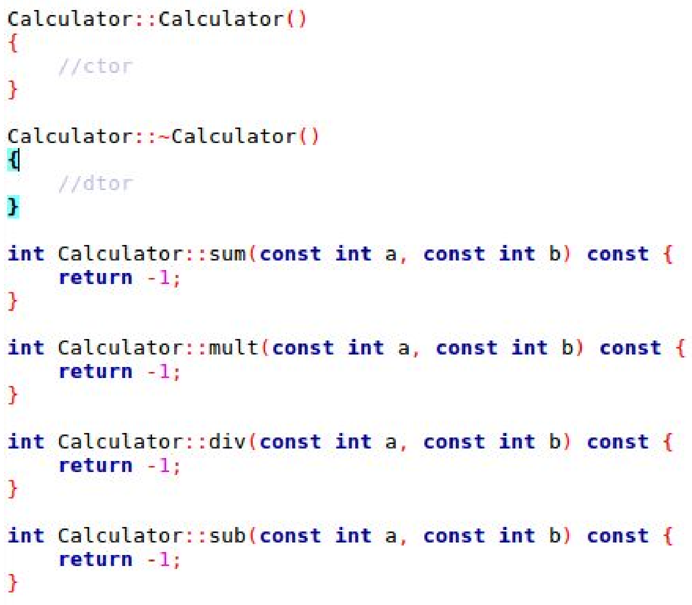
Appendix A: Figures and Tables

Figure 1.1



```
#ifndef CALCULATOR_H
#define CALCULATOR_H


class Calculator
{
    public:
        Calculator();
        virtual ~Calculator();

        int sum(const int, const int) const;
        int mult(const int, const int) const;
        int sub(const int, const int) const;
        int div(const int, const int) const;

    protected:

    private:
};

#endif // CALCULATOR_H
```

Figure 1.2



```
Calculator::Calculator()
{
    //ctor
}

Calculator::~Calculator()
{
    //dtor
}

int Calculator::sum(const int a, const int b) const {
    return -1;
}

int Calculator::mult(const int a, const int b) const {
    return -1;
}

int Calculator::div(const int a, const int b) const {
    return -1;
}

int Calculator::sub(const int a, const int b) const {
    return -1;
}
```