Standards Guide for Programming

MCFlow Project

Jeffrey McDaniel
University of California, Riverside
jmcda001@ucr.edu

Brian Crites
University of California, Riverside
bcrit001@ucr.edu

Contents

1 General Coding Standards

This guide defines a standard for the Microfluidics Continuous Flow Design Automation project (MCFlow) at the University of California, Riverside (UCR). All code that is submitted for this project must follow this standard. It has been designed to ease the integration of the many components of this project. Additionally it defines helpful strategies that will aid in the debugging and testing of the code. This standard is to be followed across all languages. There will be certain exceptions, when they arise use your best judgement or contact me for clarification.

1.1 Code Management

Text editors and Makefiles are preferred over IDE's. Without having everyone using the same IDE, it becomes difficult to transfer projects back and forth between the different IDE's. Creating your own Makefile also gives you intimate control over your project. Additionally it allows you to become more familiar with build environments, which will be useful in the long run. In my opinion it is much easier to start using an IDE after you have used text editors and Makefiles, than to go in the opposite direction.
That being said the only requirement here is that a standard directory tree format is used for submitting your project (one that most IDE's will follow to some degree). The format is as follows:

<Project_name>/

Source/
All source files
Headers/
All header files
benchmarks/
All benchmarks/test cases
scripts/
All scripts
Makefile

The project must also be compiled with the compiler versions that are listed.

- g++ 4.1.2
- g++ 4.6.3

This will include using older versions of several compilers, and even older standards of several languages. This is again to keep consistency across all levels of the project.
Templates should be followed when they are available. They will be discussed further in individual language sections.

1.2 Formatting

This section will describe the formatting of the files that are written. When opening files across many different systems, a slight change in formatting can render a file difficult to read and comprehend. This is an attempt to ensure that the file will look the same when opened by every person involved with the project. These formats may not be the default in your particular environment, but there are ways to change these in most if not all IDE's and most programming text editors.

The formatting that will be required is as follows:

- Lines are at most 120 characters.
- Use of spaces only, 4 spaces for an indent, do **NOT** use tabs (this can be specified in most text editors).
- Functions have the return type, function name, and parameters all on the same line. If they do not fit, they should be wrapped at the last comma without going over 120 characters. All subsequent lines for the declaration/definition should be indented.

  ```
  void function_name(string first_parameter, //more parameters...
      string last_parameter)
  ```

- All code blocks must have curly braces "{}". This includes conditionals, loops, and functions.
- Use a space between the key word (if, else, while, etc.) and the opening parentheses "(", and between the closing parentheses ")" and the opening curly brace "{".

  ```
  if (n < 10) { /*some code*/ }
  ```

  or

  ```
  if (n < 10) {
          /*some code*/ }
  ```

- Use shorter functions whenever possible, they help to describe the function rather than long segments of code. They also help to break up logic to ease in identifying locations of errors.

1.3 Testing and Debugging

Testing and debugging your code is an important step in any project. The way that you format your code can help to speed up this process; well structured and formatted code is much easier to use to identify the locations of errors, and to effectively correct them. The use of short function is one such method. With

shorter functions with more specific functionality, you are able to perform unit testing on the individual functions to ensure that they are correct, and if necessary fix them. They are also much shorter, and likely to be less complicated, and so easier to determine what is going wrong, and how to fix it.
The benchmarks that are provided to you are not meant to be a comprehensive list of test cases. They are used to obtain results necessary to compare the results of the different algorithms used. They will hopefully uncover many errors, but it is still your responsibility to thoroughly test any code that you write. This includes developing your own test cases, for both unit testing, and testing the overall system. These test cases should also introduce errors, and insure that your system is capable of handling these errors in an informed manner. Test cases should be designed to be used in the following way:

- All testing is to be done using test cases passed in as command line arguments. There should be minimal to no hard coded values in your program. Hard coded values make the code less generalized, and less useful in real life situations.
- When testing batches of files, create a script for the test cases, and a file for each individual test case.
- As stated above you will need comprehensive test cases. These test cases will not be able to test absolutely every aspect of your code, as that problem is NP-Complete,but they should test abnormal situations as well as normal running instances.
- Segmentation faults, out of bounds and allocation errors are not acceptable in submitted code. This indicates a lack of testing on your part. These errors, or any other, which crash the program with no indication as to the cause, are not acceptable.

Well structured code will make it easier to identify where the errors in your code lie, but you will still need methods to identify the exact location in your code. The following are several tips you can use during the debug process:

- Insert debug messages into your code. These debug messages can be used by you to help identify where there are errors, by outputting additional information about the execution.
  - A debug variable should be declared at the beginning of your code.
  - The debug messages should first check the debug variable. This allows you to set what level of debug messages you output* additionally it allows you to find all inserted debug messages if they need to be removed later.
  - The debug messages should be output to a log file that can be analyzed later.

- Insert temporary code using the TODO comment. The TODO comment automatically highlights in many text editors and IDE's, additionally it provides a keyword to search for when removing temporary debug code.
- The TODO comment can also be used to indicate where you are going to insert code in the future. Please structure it in the following way: TODO(name/email): brief description of code to add.
- Output informational message to the user showing the progress of the program.
  - A debug or verbose flag (passed from command line) should be used to print the copious useful information usually written in when writing and debugging code. These debug statements should be useful with the assumption that others will need to debug your code later.
  - Non debug/verbose outputs should provide information such as what action is being performed, an estimated progress bar, or how long it has taken. This information should be kept to a minimum and useful to non-developers using your code.

There are many additional ways to assist you in debugging your code. These are simply a suggestion for how to proceed. If you have more suggestions you would like to make, on methods that have worked for you, please write an email to me with the suggestion.

1.4 Submitting Work

When you are submitting your work, whether it be for review, or for a final release, it is important for you to submit in a well defined manner. The code that is submitted should be clean. If you are not through with your work back up a copy that is not cleaned up, but make sure that the files submitted have been cleaned up. This includes, but is not limited too, the following:

- Remove large sections of commented out code. This is code that does not work as intended. It does not aid in the understanding or the functionality of the code.
- All debug code should be appropriately displayed/squelched by the setting of the debug/verbose flag, as mentioned in section 1.3
- All of the code must include comments as described in section 2.1

In addition to cleaning up the files, please provide the test cases that you have used, including a short description of what case each file is testing. Provide the usage for each test case, and what the expected results are to be. The code should either be emailed to with the directory in a zip or tarball, or committed to a github branch if your have github write access.

2 Style Guide

The following sections will formalize the style guide to be followed for the individual languages used in this project. The ideas contained have been taken from many online forums, but the main source, when available, is the style guide for Google [**?**]. The guidelines are to help make the code cleaner and more maintainable in the future. It will also serve to keep they styl consistent across the many parties involved with this project. These stylistic constraints may not always produce the trickiest or the fastest code, but it will produce cleaner more maintainable code, which is the overarching goal of this guide.

2.1 Comments

Comments are a useful tool to help to clarify the code that you write. If you follow the guidelines in the subsequent sections your code will remain fairly clear and will not require as many comments. When writing comments though, remember to always use good punctuation, grammar, and spelling. The comments are to help clarify for, not to confuse, the person reading your code. The examples in this section will use C-style commenting, this is for demonstrative purposes, replace the commenting convention with the one appropriate for your language.

- Variable comments should not be necessary as long as the naming convention suffices as a description. If it is still unclear, provide a comment.
- Provide comments for any implementation tricks used that will reduce the read-ability of the function. These tricks should be avoided unless they provide a measurable speed-up in the resulting code.
- If passing literals to a function, comment what their values mean.
- If unable to complete a function, provide a TODO comment with your name, with a description of what the intended functionality of that block of code was. Minimize the number of these comments remaining when submitting code. For example:

  ```
  //TODO(Jeff): Output an error message describing why the method failed.
  ```

2.2 Function Headers

All functions should begin with the following header above the definition (contained in the .cpp file). The only exception is for short self-describing functions such as some accessors (getters), mutators (setters), and **very** short private helper functions. Remember that these headers must still follow the 120 character line limit, and multi-line comments should have all lines past their first indented.

```
/*
string function_name(int a, string b, double c)

Parameter a (int): short description expected a and how it's used
Parameter b (string): short description of expected b and how it's used
Parameter c (double): short description of expected c and how it's used

Return ret (string): short description of what is returned

Exception int: short description of thrown int
Exception double: short description of thrown double
*/
```

Note that errors using our custom claim() system or using cout or cerr should print out useful error message directing the user to either where the problem has occurred, or how to fix the problem (if the problem can be fixed by the user without editing the code). For sections of code dedicated to error messaging, but that do not explicitly throw an exception (see C++ exception tutorial), you should comment that section of code with what has caused this error code to trigger.

2.3 C++

C++ will be the main language used in this project. A vast majority of this style guide was taken from the Google standard on C++. The C++ style guide is the most robust, and when possible the guidelines contained within will be applied to the other languages used within this project. The exception is when the guideline is overruled by a guideline from the section on that particular language, or when it just does not make sense to follow. If you need clarification on any stylistic point please email either Brian Crites (bcrit001@ucr.edu) or Jeffrey McDaniel (jmcda001@ucr.edu).

2.3.1 General

Do not use exit( ), instead use return values to indicate error values. This allows the program to continue with execution if possible, or to perform error recovery if necessary. The return values from your overall program need to be well documented. This will also help with identifying the cause of errors or crashes.
A strict naming convention will be used for code written in C++. The following describes the convenction:

- Filenames are all lower case with underscores "_" and end in ".cc", e.g. "filename_example.cc".
- Variables and functionsare all lower case with underscores "_", e.g. "variable_example;" or "function_example();".

- Type and Class names start with a capital and are CamelCased, e.g. "TypeNameExample;".
- Constants and Enums start with a k and are CamelCased, e.g. "kConstantExample;".
- Do **NOT** use abbreviations unless they are well known outside your field.
- Do **NOT** abbreviate by removing letters, e.g.cnt for count.

2.3.2 Source and Header Files

The code base should be kept manageable by using smaller source code files with more directed functions contained within. Additionally each source file should have an associated header file with it, using the following formatting constraints:

- All header files should have #define guards in the following manner, with the entire line capitalized:

  #ifndef _<Project>_<Path>_<File>_H_

  #define _<Project>_<Path>_<File>_H_

  <Includes>

  <Code>

  #endif //_<Project>_<Path>_<File>_H_

- If a class is only referenced in the file, it should be forward declared instead of included. This will prevent circular dependencies that cause errors in compilation.
- Small functions, with only a few lines of code, should be inlined.

  inline void set_height(int new_height) {height = new_height;}

- Function parameter list is:

  <Function_name> (<Inputs>,<Outputs>) {

- Includes are ordered as follows:

  C Libaries

  C++ Libaries

  Other Library .h files

  Your .h files

- Inherited functions useful to a class should be mentioned in a comment in the appropriate section of the header file

Each individual class should have it's own source and header file following the above styles. There is an important distinction to make between classes and structs. A struct is used as a passive object carrying data only. The following style guide applies to classes, in addition to the above:

- Default constructors must be defined.
- Constructors declared with only one argument must be declared explicit.
- All members are declared private, access will be provided through the functions.
- Public members and functions are declared first, followed by private.
- Within the public section the order is first inline functions, then function declarations.