Tree Group Coding Style Guide

The following style guide should be used for **all code** written in the Tree Group. While this guide supersedes any other style guide (including those in use at BYU or in industry), the following are useful links to other style guides that contain good “best practices” for cases not covered by the current guide.

- Google’s C++ style guide: https://google.github.io/styleguide/cppguide.html
- Stroustrup and Sutter’s C++ Core Guidelines: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md
- BYU CS 142 style guide(s): https://byu.instructure.com/courses/1342/pages/cs-142-style-guide
- Google’s Python style guide: http://google.github.io/styleguide/pyguide.html
- Python’s Python style guide: https://www.python.org/dev/peps/pep-0008/

1. Headers

At the top of your “main” file, create a comment block that contains:

- Your name (and/or other contributors) and the date the code was started. If you are significantly changing a code, provide a date for the significant revisions.
- A short description of what the code does
- Any command line arguments it might take and what they do
- A list of current limitations/ideas for future improvement of the code

Create a similar header at the top of all classes. Provide:

- Your name and the date the class was written. Include additional dates when making significant reviions.
- A short description of what the class is for.
- List the base class if it is a derived class
- Any comments with current limitations and/or future improvements.

2. Indentation

Nesting must be represented with **two space** indentation. Nesting is any code found within an if statement, loop, function, class, etc. Vim can be set up to auto-insert 2 spaces when hitting tab. Note that multi-nested code should be indented multiple times.

3. Bracing

Bracing should be consistent in languages that require it. The default should be as follows unless there is good reason otherwise.

```
for (size_t I = 0; I < myVector.size(); i++)
{
  /* (2 space indent) My code here … */
}
```

4. Naming Conventions

The most important thing in naming conventions is **consistency**. Generally follow the guidelines here, but deviations may be acceptable in large codes where other rules are needed.

Local variables and functions (including class member functions) should be lower case with underscores: e.g. *variable_case, my_function*. Mathematically motivated variable names are acceptable, but should be closely detailed in code-specific documentation (e.g. papers describing the methods). Greek letters should be spelled out (e.g. use *epsilon* or *eps* instead of *e* and *pi* instead of *p*).

Classes should be in upper camel case, e.g. *UpperCamel*Case. When appropriate, derived classes should be differentiated by an underscore, e.g. TimeInt_Base for a base class and TimeInt_ModelB for a derived class. Class member variables should be in upper camel case preceeded by an underscore, e.g. _UpperCamelCaseVar.

Anything defined using pre-processor flags (including constants and include guards) should be in all caps with underscores: *ALL_CAPS.*

Global variables and global functions should be in upper camel case, e.g. *UpperCamelCase*.

Summary:

- local variables and functions: *lower_case_with_underscores*
- classes: *UpperCamelCase*, derived classes with underscore: *Base_Derived*
- class member variables: *_UpperCameWithUnderscore*
- pre-processor variables: *ALL_CAPS*
- global variables and functions: *UpperCamelCase*

5. Global Variables

No non-constant global variables and limited use of global functions. Global functions should be limited to highly re-usable code.

For larger codes (i.e. more than one or two files), global variables and functions should be included in a separate file, e.g. global.h and global.cpp. If these files exist, no global variables or functions should be defined outside of them.

6. Use Appropriate White Space

Most compilers will ignore white space, so use it freely! It doesn't make your code better to fit four lines of code onto one line. Here are some ideas for adding whitespace:

- Add spaces in between terms of mathematical equations or use spaces with parentheses.
- Add a couple of line breaks in between functions or parts of the code that fit together.
- Adding whole-line comments to organize parts of code, groups of functions or in class definitions.

Finally, add code folding markers `{{{` and `}}}` around long bits of code. *This should include all functions*. These can be easily used with vim by adding the command: `set foldmethod=marker` in your .vimrc. Folds can be opened and closed with zo (open) and zc (close). More information on code folding in vim can be found here: https://vim.fandom.com/wiki/Folding

Example:

```
void my_function()
{ // {{{
…
} // }}}
```

7. Function Documentation

Functions should be documented with a small description of what they do and what arguments are needed. For example:

```
void findSomethingInSomethingElse(myClass somethingElse, myOtherClass something)
{ // {{{
// Finds Something in Something Else
// somethingElse - Object to be searched in
// something - Object to be searched for
…
} // }}}
```

8. Self-Documenting Code

Your program should be “Self-Documenting” and "easy to read" such that a person reading your code can understand how it works. You must achieve this by:

• Use descriptive names for all of your constants, variables, functions and classes. Names are often be made up of two words except in commonly defined variables, e.g. rho for density or r for radius. Even for common variables, you may consider using two words, e.g. rho_water for the density of water. Using two words greatly decreases the likelihood of variable collision (two variables with the same name).

• Use functions/classes to break up long complex sequences of code into self-contained, single-purpose parts. As a rule of thumb, functions should generally not be more than a single screen in length.

• If you find yourself needing more than an occasional comment, you have probably made your code unnecessarily complex. Fix the code, rather than add a comment.

9. Include Guards

All header files should have either `#define` include guards or the `#pragma once` directive to prevent multiple inclusion. Regardless of the method you pick, *the entire program should be consistent in using one or the other.* If you use traditional include guards, the format of the symbol name that you define should be all caps with a trailing underscore ‘h’: <*CLASSNAME*>*_H.* For example (for a class named FooBar):

```
#ifndef FOOBAR_H

#define FOOBAR_H

...

#endif  // FOOBAR_H
```