

FIRST

Project Guidelines and Software Coding Standards

Written by:

Dan Farrell

Brandon Hespeneide

Scott Menor

Michael Thorpe

Stephen Wells

Last Revised: March 13, 2006

Copyright © 2006

Center for Biological Physics

Arizona State University

| | |
|--|-----------|
| FIRST | 1 |
| 1. Introduction..... | 4 |
| 2. FIRST coding standards..... | 4 |
| 2.1. Why do we need coding standards?..... | 4 |
| 2.2. File naming conventions | 5 |
| 2.2.1. Code files | 5 |
| 2.2.2. Files created by the software..... | 5 |
| 2.3. Comments | 6 |
| 2.3.1. Source code file header..... | 6 |
| 2.3.2. Function comment header | 6 |
| 2.3.3. Comments within functions..... | 7 |
| 2.4. Indentation..... | 7 |
| 2.4.1. Spacing | 7 |
| 2.4.2. Line length..... | 7 |
| 2.4.3. Dividing long lines..... | 7 |
| 2.5. Declarations..... | 7 |
| 2.5.1. class and struct declartions | 8 |
| 2.6. Statements | 8 |
| 2.7. Naming conventions | 9 |
| 3. Version release schedule..... | 10 |
| 4. Nomenclature guidelines | 10 |

1. Introduction

This document is meant to be a guide for people working on projects related to the software FIRST. The majority of this guide is devoted to software coding standards, and as such, is aimed towards FIRST source-code developers. However, this guide also covers accepted nomenclature to be used when describing FIRST and the primary software modules FRODA and TIMME.

The coding standards and other guidelines outlined in this document do not define a complete standard; not everything is covered. However, what is given should be followed. This guide is NOT meant to eliminate the need for communication between people who develop and apply the FIRST software.

2. FIRST coding standards

This section describes the style that should be used when developing source code for FIRST. The bulk of the standard is based on the JAVA code standards proposed by SUN Microsystems. An example source-code file is given in the last section, however, the best example will be to look at the FIRST source-code and header files that currently exist.

In general, it is a good idea to maintain a high level of communication between people who are actively developing the code. The Flexweb forum and group emails are the certainly the best communication methods. This will not only ensure quality code, but limit the amount of overlapping or redundant code that exists. If possible, one person should act as a “project manager”. This person should have a broad understanding of the code, especially the input and output.

2.1. Why do we need coding standards?

Reasons to have coding standards:

1. In the lifetime of this project, no single piece of code will be maintained by the original author.
2. Standards improve the readability of your code, for you and for others.
3. Fewer mistakes occur in a consistent coding environment.
4. *New people can get up to speed quicker.* This is perhaps the most important reason. This project is currently maintained in an academic environment. New students will likely have little or no experience with a C++ project of this magnitude. Standards provide a consistent framework for new programmers to understand the code.

2.2. File naming conventions

2.2.1. Code files

Files that contain code should use the following suffixes:

| File Description | Suffix |
|----------------------|--------|
| C++ source-code file | .cpp |
| C++ header file | .h |

2.2.2. Files created by the software

Files that are created by the software should use the following convention:

`baseName_descriptiveFileName_optionalFileNumber.fileExtension`

| Part of output file above | Description |
|----------------------------------|--|
| <code>baseName</code> | Base name of the input file. |
| <code>descriptiveFileName</code> | Should be a short, one-word description of the what the file represents. |
| <i>optionalFileNumber</i> | If the module you are writing generates many files, append a 6-column field with an integer representing the file order. Use zeros (0) as placeholders for any leading whitespace. (ie, 00001, 00002, ...) |
| <code>fileExtension</code> | If the output file adheres to a standard file format, the appropriate file extension should be used. If the file contains data, but adheres to no known file type, use ".txt" as the file extension. |

Example:

| Input file | Output files |
|------------|---|
| 1a2p.pdb | 1a2p_RCD.pdb 1a2p_RCD.pml 1a2p_froda_00001.pdb 1a2p_froda_00002.pdb 1a2p_froda_00003.pdb 1a2p_froda_00004.pdb 1a2p_dilution.ps 1a2p_dilution.svg 1a2p_results.txt 1a2p_bond.txt 1a2p_data.txt |

Notes:

1. All output files should have a file extension.

2.3. Comments

The C++ “//” comment style should be used in preference to the “/* ... */” style. If you are “commenting out” a block of code for debugging purposes, the “/* ... */” style is fine. Legacy code that had been commented out should be removed from the source file once it has been established that it is no longer needed. It will always be available via the CVS archive.

2.3.1. Source code file header

The top of each source code file should begin with a comment block:

```
// Author name  
// Date file was created
```

2.3.2. Function comment header

Function headers are useful for understanding the purpose of a function before delving into the actual code. Also, third-party software is being used to create source-code documentation, and this documentation software relies on a standard commenting format for proper parsing of the code files. Every function should begin with a comment block in the following format. The long string of //’s contains 80 forward slashes.

```
////////////////////////////////////  
// Description:  
// This function does such-and-such.  
// Parameters:  
// argument var 1 - map of variable attributes  
// argument var 2 - file pointer  
// Return Value List:  
// This function returns a structure containing X, Y and Z values.
```

////////////////////////////////////

2.3.3. Comments within functions

Comments within a function can be used to either demark sequential code sections or clarify a nontrivial code structure/variable. If you find that you are adding many comments within a function, it may be that your code is not clearly written.

2.4. Indentation

2.4.1. Spacing

Two spaces should be used as the unit of indentation. Please adjust the “tab” key to reflect this if you are using a development editor such as Eclipse or Xemacs.

Notes:

1. In Xemacs, the indentation should be set if you are in “c++-mode”. This mode should start automatically if you load a file with a .cpp extension.

2.4.2. Line length

Single lines should not exceed 80 characters.

2.4.3. Dividing long lines

The following rules should be followed when deciding how to break a line:

1. Break after a comma.
2. Break before an operator.
3. Try not to break within parenthesis.
4. Align the next line with where the expression began on the previous line.

2.5. Declarations

The following rules should be followed for declarations:

| Rule | Examples |
|---------------------------|---|
| One declaration per line. | <pre>int newCounter; float newRMSD; ofstream *outputPDBFile;</pre> <p>Wrong:</p> <pre>int a, b, c;</pre> |

Keep reference type with the variable name.

```
siteInfo *prevFramework;  
ofstream *outputFile;
```

Wrong:

```
siteinfo* prevFramework;  
Ofstream * outputFile;
```

2.5.1. class and struct declartions

class and struct declarations should be contained in their own header file. The header file should be named the same as the class or struct name. Elements of these declarations should appear in the following order:

1. Comment block
2. `class` or `struct` declaration
3. Member variables. These should be listed in order of public first, then protected, then private.
4. Constructor/destructor.
5. Member functions. These should be listed in order of public first, then protected, then private.

2.6. Statements

The following rules should be followed for statements:

| Rule | Examples |
|---|--|
| One statement per line. | <pre>siteInfo[a] = 12; a++;</pre> <p>Wrong:</p> <pre>siteInfo[a] = 12; a++;</pre> |
| Beginning braces for conditional statements should start one space after the conditional statement. The closing brace should be on its own line, and align with the opening of the statement. | <pre>if (a == 1) { do_something(); }</pre> <p>Wrong:</p> <pre>if (a == 0) { do_something; } if (a == 0) { do_someting(); }</pre> |

Examples:

An example if-then-else statement:

```
if (...) {
    do something;
}
else if (...) {
    do something else;
}
else {
    do this instead;
}
```

An example do-while loop:

```
do {
    something++;
    and_this_too();
} while (...);
```

2.7. Naming conventions

Naming conventions cover how class names, functions, variables and constants should be written. In general, names should be descriptive. Whole words should be used unless the abbreviation is well known, such PDB for Protein Data Bank. One or two descriptive words should suffice; whole sentence variable names can make code harder to read. In general, verbs should be used for functions, nouns for all other types.

The following naming conventions for the code used in FIRST should be used:

| Type | Convention / examples |
|--------------------|---|
| Class Declarations | These should consist of non-delimited words. All the words should be capitalized. MolFramework SiteInfo MyNewClass |

| | |
|----------------------|--|
| Functions, Variables | <p>These should consist of nondelimited words. The first word should be all lowercase. Any subsequent words should have the first letter capitalized.</p> <pre>siteInfo someVariableToDoSomething identifyHydrogenBonds() runFRODA()</pre> |
| Constants | <p>These should consist of underscore-delimited words. All letters should be capitalized.</p> <pre>MOL_FRAMEWORK_H PI SP2_SP2</pre> |

3. Version release schedule

Release of new versions should coincide with the beginning of the academic year. All precautions should be taken to ensure the release is stable. The major developers together with the FIRST project manager (should there be one) will decide whether the release will be a major release (ie. FIRST5 → FIRST6) or a minor release (ie. FIRST5.2 → FIRST5.4).

4. Nomenclature guidelines

This section is meant to outline the accepted terminology for describing the various algorithms, procedures and applications of FIRST. There are multiple ways of describing the same thing, however, it is **strongly** recommended that these words and phrases be used in publications, posters and written descriptions.

| Preferred word or phrase | Meaning / Synonyms |
|--------------------------|---|
| ghost template | <p>The geometric representation of a group of atoms that are being treated as a rigid unit during FRODA dynamics.</p> <p><i>ghost bodies, ghost body templates.</i></p> |
| FRODA relaxation | <p>Protocol developed by Dan Farrell and Stephen Wells to relax steric overlaps in an structure input to FIRST.</p> <p><i>sanitization</i></p> |