Coding Style Guide
Zeus Numerix Pvt. Ltd.

Contents

1 Introduction

With the objective of improving our coding practices, here as a start, is presented the first installation of the *Coding Style Guide*. Every week we'll append a new page to this document and make it available to everyone. The aim of this document is to grow in to the coding conventions standard text for Zeus Numerix. With that, we begin.

These conventions directly apply to C++ code but can be extended to other languages (Python, Java) as well. The are broadly categorised into the following:

- Formatting Conventions (Section 2)
  Specifies conventions of aspects such as whitespace, indenting etc. The rules under this section are mainly responsible for giving a consistent 'look' to the code and maintain readability. Consistently styled code is easier to read and if everyone follows them, the code becomes a lot easier to understand and maintain.

- Naming Conventions (Section 3)
  These rules specify how variables, functions, classes and other identifiers are to be named. Just by reading the name of an identifier, it should be possible to gain information about it's purpose, type, scope etc.

- Commenting Conventions (Section 4)
  Probably the most important set of rules in this document. Comments allow inclusion of documentation of the code within the source file itself. Such documentation is priceless to maintainers and co-authors. Not to mention, it serves the original author of the code in remembering what's happening in the code when revisiting after a while.

- Programming Conventions
  Conventions for higher level constructs, design, project management and general programming. These are here to ensure consistency in such aspects of programming. Also, things that don't fit in the above section end up here. *This section is not present currently in this document.*

You may find some ideas presented here are brilliant and some are dumb. But it is important for everyone to be consistent for the conventions to be effective. You may still conclude that this is not worth it etc. but there's a road you must follow to accept something different. Allow yourself to travel it for a while.

2 Formatting Conventions

These rules define the basic 'look' of the code.

2.1 Indenting

- Set indent width to 4 spaces. 4 is not too small like 2 and not too big like 8.

- **Use spaces instead of the 'Tab' character**, i.e. an indent should expand into 4 space characters instead of one tab character. All code editors allow this setting and can automatically expand tabs to the set number of spaces.

Example:

```
1 void
2 foo() {
3     // 4 space indent from this point
4     if ( bar ) {
5         doSomething();
6     }
7 }
```

2.2 Line Width

Any line in code should not exceed 80 characters.

- Smaller windows can contain the code.
  This allows for multiple code windows to exist side by side without need for annoying sideways scrolling or ugly wrapping.
- Some text only systems can only print 80 characters wide.
  The code will arbitrarily wrap on such systems.
- Printers can only print so wide.
  At 80 characters wide, any printer can manage without wrapping code.
- We can even view and print diff output correctly on all terminals and printers.

2.3 Wrapping Lines

Since the line width is limited, it is important to wrap it in a readable way.

- Break long lines after operators or commas ',' etc.
- Wrapped parts of long lines are to be double indented (i.e. 8 spaces from previous indent level)
- When block opening lines are wrapped, the opening brace must appear on a new line by itself.

Example:

```
1 if ( someFineBigThing > thisOtherCoolThing &&
2         theThirdSuperThing == theMainThing &&
3         theKillerThing && theClosingCondition )
4 {
5   ...
6 }
```

2.4 Whitespace

The objective here is to have just enough whitespace for comfortable reading while packing as much code as possible per screen.

- **Do not use TAB anywhere in code. Have no trailing whitespace.**
- Leave blank lines:
  - Between functions
  - Before goto labels, class visibility labels.
  - Before start of control blocks (if, for, switch etc.) unless previous line was itself a block opener (contains an opening brace '{') or consists of just a closing brace '}'.
  - Before lines with `return` statements.
  - Between logically separate lines/group of code (eg. after all variable declarations).
  - 2 blank lines at the end of all files.

2.5 Braces

- Use K&R bracing style: left brace at end of first line, cuddle else on both sides. (eg. } else {)
- Always brace controlled statements, even a single-line consequent of an if and/or else. This is redundant typically, but it avoids dangling else bugs.
- When block opening statements need to be wrapped, then the corresponding opening brace will appear in the next line by itself at the opening lines indent level.

Example:

```
1  void
2  foo() {                        // opening brace at end of line
3      if ( bar ) {
4          dosomething()
5      } else {                   // cuddled else
6          doSomethingElse();
7      }
8
9      if ( bar2 ) {
10         singleLine();          // single line in if block,
11                                // still braced
12     }
13 }
14
15 // example of long opening line
16 zn::status InputFilesList::
17 initSuperObjectX( string &title,
18                    unsigned int serialNum,
19                    unsigned int totalCount,
20                    unsigned int effectiveCount )
21 {
22     ...
23 }
```

3 Naming Conventions

These are the general naming guidelines. These guidelines should be adhered to as much as possible.

3.1 General Guides

1. Except names of macros and filenames, **camelCase/mixedCase** names are to be used everywhere.
2. Use as descriptive names as possible. Avoid ambiguous abbreviations or arbitrary characters.

```
int errorCount = 0;                          // Good
float selectedSequenceDuration = 0;          // Good

int k;                                       // Bad - meaningless
bool cprCatchFg;                             // Bad - ambiguous
                                             // abbreviation
```

3. Single character names may be used for counters in small/simple loops. Acceptable names are `a`, `b`, `c` or `m`, `n` or `i`, `j`, `k`. If the code is complex and the counter is also used outside the scope of the loop block, then use a longer and meaningful name.

```
// here 'c' is being used as a counter within the scope of
// the for loop
for ( int c = 0; c < totalFrames; c++ ) {
    // ...
}
```

4. Type and variable names should typically be nouns: e.g., `TextReader`, `errorCount`. Whereas function names must be imperative e.g., `printReport()`, `logError()`

5. Do not use abbreviations unless they are extremely well known even outside the project. For example:

```
int dnsHitCount = 0;                         // Good - DNS is a
                                             // widely known
                                             // acronym
float selectedSequenceDuration = 0;          // Good - no
                                             // abbreviations
int tmpVal;                                  // Bad - what is tmp?
                                             // is it temporary or
                                             // temperature?
int pcReader;                                // Bad - Lots of
                                             // things
                                             // can be abbreviated
                                             // "pc".
```

3.2 File Names

1. Filenames must be all lowercase and can include underscores (_) or dashes (-). Underscores (_) are preferred if there are no existing large-scale use of dashes (-). E.g., `data_file_reader.cpp`, `data_file_reader_test.cpp`, `common.cpp`. This will go a long way in maintaining cross-OS filename compatibility. Using camelCase/mixedCase are strictly prohibited as these can lead to filenames such as `filelogger.cpp` and `fileLogger.cpp` existing legally on some filesystems (e.g., ext3, ext4) but fail on others that are case-insensitive (e.g.m fat32, NTFS). Again, underscores (_) are the preferred way to separate words.

2. C++ source files must end with a `.cpp` and `C` source files with a `.c`. All headers must end with a `.h`.

3. Never use filenames that already exists among standard includes/libraries. E.g., `mpi.h`, `signal.h`

3.3 Type Names

1. Type (classes, structs, typedefs, and enums) names must be camelCase/mixedCase and must start with a capital letter, with no *underscores*.

```
1  // classes and structs
2  class UrlTable { ...
3  class UrlTableTester { ...
4  struct UrlTableProperties { ...
5
6  // typedefs
7  typedef hash_map<UrlTableProperties *, string>
8      PropertiesMap;
9
10 // enums
11 enum UrlTableErrors { ...
```

2. Name the class after what it is. If you can't think of what it is then, that is a clue you have not thought through the design well enough.

3. Avoid the temptation of bringing the name of the base class into the derived class's name. A class should stand on its own. It doesn't matter what it derives from.

4. Could possibly adopt this → [1]

3.4 Variables

1. Use camelCase/mixedCase everywhere. Start with a small letter.

2. Class member variables must be prefixed with an 'm'. This will prevent any conflict with method names and make member variables readily identifiable.

```
1  int mLockCount;
2  std::string mName;
```

3. Global variables must be prefixed with a 'g'.

3.5 Constants

1. Prefix constants with a 'k' Example:

```
1  const int kMaxFrames = 42;
```

3.6 Function Names

1. Regular functions must have names that are camelCase/mixedCase and start with a small letter and have no underscores (_).

```
1 bool loadData();
2 bool killConnection();
```

2. Accessors and mutators (get and set functions) should match the name of the variable they are getting and setting.

```
1 class MyClass {
2 public:
3     ...
4     int height() const { return mHeight; }              // getter
5     void setHeight( int aHeight ) { mHeight = aHeight; }// setter
6
7 private:
8     int mHeight;
9 };
```

3.7 Enumerations

Enum entries must use constant style naming i.e., with a 'k' prefix.

```
1 enum Status {
2     kOK = 0,
3     kErrorOutOfMemory,
4     kErrorMalformedInput,
5 };
```

3.8 Macros

Macro names must be all upper case and separated by underscores (_).

```
1 #define __TOPAZ_PLOT_WINDOW_H__
2 #define MY_CONSTANT 42
```

Important: As much as possible, avoid defining and using Macros. Why? you ask, because they can be terribly annoying during debugging. If you'd like more reasons then:

- Macros don't obey scoping rules.
- Macros can be/get redefined.
- Macros don't have types!! ← baaaad.
- for more see [2]

But there always are exceptions. And some unavoidably useful macro applications are:

- As `#include` guards
- For conditional compilations using `#ifdef`
- To avoid extreme copy-pasting of code

3.9 Prefix Table

Prefix	Usage	Example
g	for global variables	`long gInstances=0;`
m	for class member variables	`int mLockCount;`
k	for constants	`const float kMyConst = 3.1415;`

4 Commenting Conventions

Comments in code help document it. Well documented code is easy to maintain and extend. It must be understood that a piece of code is worked upon by many programmers over time. If it were never documented, then each fellow will have to read the code to learn what it's doing. This can be a tedious process that slows down development.

Some of the conventions listed here are also meant to be used in conjunction with Doxygen for auto-generation of HTML/latex code documentation.

4.1 General

1. Use // comments everywhere except where explicitly specified (see next point).
2. Comment blocks (/* */) are to be used only for temporarily disabling large swaths of code. **These are not to be used for any documentation purposes.**
3. Practice writing documentation along with the code instead of leaving documentation for the end.

4.2 Comment Keywords

Special embedded keywords/phrases in comments are used to point out issues, potential problems or convey intent to mitigate any ambiguity.

- `// TODO: ...`
  Put messages about things to do in code.
- `// FIXME: ...`
  Marker for known issues in code (bugs etc.)
- `// WARNING: ...`
  Beware of something.
- `// Fall Through`
  To be used as last line in a `case` block that does not have a `break`; or `continue`; or `return`; at the end. Indicates to the reader that allowing control to fall through to the following case block is intentional. Note that, a case block allowing fall-through without this comment shall be considered as a bug.

Example:

```
switch ( foo ) {
    case CASE_A:
        handleCaseA();
        break;
    case CASE_ERR:
        logError();
        // Fall through
    default:
        defaultCase();
        break;
} // switch
```

4.3 File Comments

Every file source and header, must start with a file comment with the following details in order:

- Copyright notice.
- License type
  - If a Zeus closed source code, this must say: All rights reserved.
  - In case of code delivery etc. the full license text should be placed in the source root and this line must say: License: see LICENSE.TXT)
- Author line
  - Name of initial author and email address in angle brackets '< >'
  - If other contributors make significant edits, then they should add their name to this list in a new line.

In a following comment block the content description should be present. This is needed only for source files mostly. In case of a header+source pair for a class, the content description must be in one of the files only. The file without the content description should indicate that it is present in the other of the pair. The content description should contain information about the purpose of the class/code, usage etc. For a particularly complicated class or algorithm the comment should include an explanation about it.

```
// Copyright 2010 Zeus Numerix Pvt. Ltd.
// All rights reserved
// author: Super Man <super.man@zeusnumerix.com>

// This is the content description comment. It explains the
// contents of the file.

...
```

4.4 Function Comments

- This is a block of comments at the start of each function, just above its opening line.
- Used by doxygen to generate HTML/latex documentation of the code. Hence, following certain conventions (explained below) in writing these comments is very important.
- Conventions
  - Use /// to start each comment line in the block.
  - The first sentence (till the first dot '.') should be a _short_ description of what the function does. This is used by Doxygen as the function brief. This may be followed by more explanatory text.
  - Each argument that the function takes must be explained using a `@param` tag.
  - The param tag has the format: `@param[inout|in|out]` <argument> <description>
  - The <description> is to be put on the next line, indented by 4 spaces.
  - Return value if any should be explained using `@return` tag.

```
1  /// Determines what type of sequence numbers are in use. The
2  /// numbers may or may not be 0 padded. It also populates
3  /// internal variables accordingly.
4  ///
5  /// NOTE: Call this only after a successful call to
6  /// splitFileName.
7  ///
8  /// @param[in] data
9  ///     Reference to a std::string object containing the file
10 ///     name of the seed file. This must be just the filename
11 ///     without the path.
12 ///
13 /// @return true is returned if successful.
14 bool InputFilesList::
15 determineSequencing( const std::string &data ) {
16     ...
17 }
```

References

[1] `http://www.possibility.com/Cpp/CppCodingStandard.html\#tnames`

[2] `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`