CODING STANDARDS AND GUIDELINES FOR GOOD SOFTWARE ENGINEERING PRACTICE for Audarya

TABLE OF CONTENTS

1. INTRODUCTION

Be a software engineer – not a programmer!
Programmer is anybody who knows the syntax of a language and can "throw together" some code that works. However, software engineer is someone who not only writes code that works correctly, but also

writes *high quality* code. High quality code has the following characteristics:

- **Maintainability** (easy to find and fix bugs, easy to add new features)
- **Readability** (easy to read)
- **Understandability** (easy to understand – this is not the same as readability, e.g. redundant and repetitive code may be very readable, but not necessarily easily understandable)
- **Reusability** (easy to reuse parts of code, e.g. functions, classes, data structures, etc.)
- **Robustness** (code that can handle unexpected inputs and conditions)
- **Reliability** (code that is unlikely to produce wrong results; it either works correctly, or reports errors)

A software engineer's responsibility is to produce software that is a business asset and is going to last for many years. If an engineer produces low quality code that others find it hard to understand and hard to maintain, then it might as well be thrown away and rewritten from scratch. Unfortunately this happens all too often (in the worst case, a company will first lose a lot of money trying to maintain the old code, before it realizes that it would be cheaper to throw it away and completely rewrite it). *Making code readable and maintainable is as important, or more important than making it work correctly!* If it doesn't work, it can be fixed. If it can't be maintained, it's scrap.

This document attempts to establish some good rules and guidelines that will aid engineers in producing quality software. Standards help to ensure that the software developers communicate more effectively with each other (and with future developers), and that everyone knows and follows the same good rules that improve code quality. It is better for engineers to be creative with algorithms and data structures, rather than being “creative” with the coding style and language tricks.

2. Naming conventions

Overview:

- Names of variables in mixed case, beginning with lowercase.
- Function names in mixed case, beginning with uppercase.
- Global variables start with “g”.

2.1 Regular variables

Use mixed case, starting with lowercase. Avoid names that are too short and cryptic, but also avoid extremely long names. Do not use any prefix (e.g. Hungarian notation), except for the global variables that are discussed next.

2.2 Global variables

Global variables should start with 'g'. The reason is that it is always important to know the scope of a variable, especially for global variables that have dangerously large scope.
Example
gParser, gSimulationClock.

2.3 Function names

Function names are in mixed case, and start with uppercase. They should normally be a verb, or a verb followed by noun. Avoid starting function names with nouns, as it may be confusing to the reader. Usually every method and function performs an action, so the name should reflect this.
Example
CheckForErrors() instead of ErrorCheck(), DumpData() instead of DataDump(). GetLength() instead of Lenth()

2.4 Constants
For global constants, or constants declared inside classes, use all uppercase with underscores to separate words.
This facilitates catching type errors at compile-time.
Example

```
const int FOO_CONSTANT = 78.123;
```

For local variables that are declared as “const” to improve code Understandability (see 5.12 “Declare variables that don’t change as const”), use the regular naming conventions for variable names.
Example

```
const int dimensions = 3;
```

2.5 Keep names consistent

Avoid using different names (or different abbreviations) for the same concept. For example, instead of using all names “transform”, “xform”, and “xfrm” in different parts of the program to refer to the same entity, try to choose one name and use it consistently.

2.6 Use pronounceable, non-cryptic names

One rule of thumb is that a name that cannot be pronounced is a bad name. A long name is usually better than a short, cryptic name (however, be aware that extremely long names are not good either! See 2.7 “Avoid extremely long names”). In general, names should be meaningful and descriptive to enhance readability and comprehension.

Example
Instead of:

```
double trvlTm;
double spdRng[2];
Point stPnt;
```

use:

```
double travelTime; // total travel time
double speedRange[2]; // range of speeds during this trip
Point startPoint; //starting point for the trip
```

Put comments that fully clarify the use of each variable, if it’s not obvious from the name.

2.7 Avoid extremely long names

Names for identifiers should be chosen so that they enhance readability and writeability. In general, longer names are more readable that shorter, abbreviated ones; however, there is a limit to how long a name can be before it actually starts having *detrimental effect* on readability. Names like the following:

```
fgr_n_caption_pairs_to_n_equation_sequence_numbers
FGR_ STATIC_SEQUENCE_EQUATION_MARKER_TYPE_ID
```

are hard to write, and they also make the code harder to read! How easy it is to read (or write) code that looks like the following?

```
plotter_write_figures_block_info_out(FGR_
DESCENDING_SEQUENCE_MARKER_TYPE_ID);
picture_n_caption_pairs_to_n_linear_sequence_markers(block_info_n
_caption_pairs);
plotter_block_info_n_linear_sequence_markers_select(block_info_n_li
near_sequence_markers);
```

A much more readable version of the previous would probably look like this:

```
plotter.Write(figures, DESCENDING);
picture.GetLinearMarkers(captionPairs);
plotter.Select(markers)
```

3. COMMENTS

3.1 General guidelines

Put always a header comment at every function definition. Use brief comments throughout the code to increase code understandability.
In general, comments should be:

- Brief
- Clear
- Don’t contain information that can be easily found in the code
- Explain WHY instead of HOW (except when commenting in the large)

3.2 Comments should be brief

When writing comments, try to be brief and succinct.

Example

Instead of:
```
// the following code will calculate all the missing values of the
// pressure by applying the spline interpolation algorithm on the
// existing pressure values
```
Write:
```
// calculate missing pressures using spline interpolation
```

3.3 Don't restate code with comments

Don't give information in your comments that is obvious from the code.

Example

The following are bad comments (they give information that is obvious from the code):
```
// loop until reaching current time
while (time<currentTime) {
// if vector is greater than screenMinSize, calculate factor
if (vector >10) {
factor = factor + ......
}
...
// increase time
time = time + 1;
}
```
Here is a better version (comments offer new information, and are not obvious):

```
// calculate the new factor
while (time<current_time) {
// only vectors visible on screen affect the factor
if (vector > screenMinSize) {
factor = factor + .......
}
....
time = time + 1;
```

}

3.4 Comments should explain WHY instead of HOW

High quality code is by definition readable and clear enough anyway, so that the average programmer can easily understand *how* it works just by reading it. However, the reasons *why* a particular algorithm was chosen, or *why* a particular data structure is used, or *why* a certain action must be taken, usually cannot be derived just be reading the code. This is the information that should be documented in comments throughout the code.
The only case that comments can be at the "HOW" side is when commenting in the large; that is, when a short comment precedes a large block of code to summarize what the code does (but without getting into the details).
Example

```
// use spline interpolation to calculate intermediate forces
{
....
}
```

3.5 Placement of comments

Comments should precede the block of code with which they are associated and should be indented to the same level. This makes it easier to see the code structure.

3.6 Variable declaration comments

If not completely obvious from the choice of the variable name, describe how the variable will be used.
Example

```
long vectorLength = 0;
Window prevWin; //window that had the focus previously
```

3.7 Comments should be blended into the code

With the exception of function headers, most other comments should be as close to the code they refer to as possible. This “connects” them better with the code. More importantly, it reduces the risk of the comments going out-of-date when programmers make code changes (if comments are not close to the code they refer to, someone might change the code and forget to update the comment).

Example

Instead of:

```
// ... details ... blah blah blah blah blah
// ... more details... blah blah blah blah
// ... even more ... blah blah blah blah.
{
DoThis();
DoThat()
while(cond) {
Foo1();
Foo2();
Foo3();
MoreFoo();
}
CheckFoo();
}
```

<u>USE</u>:

```
// ... brief, overview comment with no details...
{
// ... local details ...
DoThis();
DoThat()
// ... local details ...
while(notDone) {
Foo1();
// ... local details ...
Foo2();
Foo3();
MoreFoo();
}
// ... local details ...
CheckFoo();
}
```

3.8 “TODO” comments

Use //TODO: comments to remind yourself and others about work that needs to be done in the code.

Example

```
//TODO: replace these calculations with calls to Screen library
drawGrid[n].width = ......
```

TODO comments also make easy to summarize what needs to be done, by running a global search or grep for “TODO”.

3.9 Function headers

Use function header comments in every function that you write. A function header should contain a brief description of what the function does. Avoid putting implementation details in the header (try to focus on WHAT the function does, not HOW it does it). Think of it as a black box, that you have to describe its usage. The header should be the “user’s manual” for whoever uses this function. Describe what every input parameter is (unless it’s obvious), and what this function returns. Keep function headers short. Long function headers reduce code visibility. Avoid having empty lines and redundant information. 3-5 lines must be enough in most cases. Do NOT put implementation details in the header, for example what are the local variables, or which subroutines are called from this function. This increases maintenance cost (function header comments must be modified along with the code; in addition, there’s risk that the comments will go out-of-date if programmers forget to update them). If someone wants to find out how the function works, it’s better to read directly the code. You can help by providing good and meaningful comments throughout the code (see 3.7 “Comments should be blended into the code”).

3.10 Pre-conditions – post-conditions

Do NOT write comments about pre-conditions and post-conditions in the function headers or in the function code. Instead, use TRAP statements to verify that you're pre- and post-conditions are true (see 8.5 "Error handling with TRAP and ERROR"). In this way pre- and post-conditions become "alive" and serve a meaningful purpose. Your code will be more reliable and more correct since bugs will be caught earlier.
Use TRAP to check input parameters, and check the results of your calculations. Use them even for things that you believe are "obvious". You will be surprised to discover that things can be much different than what you thought, especially after code modifications.

3.11 Don't overdo with comments

Too many comments are as bad as too little comments! In most cases, a comments-to-code ratio of 20%-
30% should be sufficient. If you find yourself writing as much comments (or more) than code, then probably you overdo it. Too many comments impair code readability, programmers most likely will quit reading them, and in turn, comments are likely to go out-of-date when code changes are made by programmers who simply ignored those huge comment sections. To be effective, comments should be brief and describe something that is not apparent from the code.

5. WRITING READABLE CODE

5.1 Function length

Try to write small functions, no longer than 2 pages of code. This makes it easier to read and understand them, and it also promotes code reusability and maintainability. It helps you avoid duplicating code by factoring out common code into small functions. It also results in code that is more self-documented since shorter functions usually contain mostly function calls, which presumably have meaningful names that can be a great help for a reader trying to understand that code.

5.2 Function declarations

Don’t omit dummy parameter names in function declarations unless the meaning is clear without them.
Also always include parameter names when you have more than one parameter of the same type; otherwise it’s impossible to figure out which one is which.
Example
Instead of

```
void DrawCircle(Color, long, long, double);
```

do:

```
void DrawCircle(Color, long centerX, long centerY, double radius);
```

5.3 Avoid having too many parameters in functions

Functions that take too many parameters often indicate poor code design and poor usage of data structures. If classes are used properly, then there is no need to pass many parameters to member functions. Think of a class as a storage place that stores data between function calls. In this way, its member functions should be able to find inside the class most of the data they need.
Too many parameters may also indicate limited/poor usage of data structures. For example, instead of passing separately the coordinates of a line as x1, y1, z1, x2, y2, z2, consider creating a class Line that contains these coordinates. Then you will only need to pass one parameter (a Line object) instead of six parameters. As another example, instead of having a function:

```
PrintImage(image, printer, pageWidth, pageHeight, inkColor,
tonerQuality, resolution, copies);
```

consider creating and using a class PrintSetup that contains all printing-associated parameters, and pass this to the function:

```
PrintImage(image, printer, printSetup);
```

5.4 Avoid deeply nested code

Avoid having too many levels of nesting in your code. As a rule of thumb, try to have no more than 5 nesting levels. If your code needs more nesting, consider creating a function for some of the nested code.

Code that looks like this:

```
}
}
}
}
}
}
}
}
}
}
```

should automatically raise a red flag in your mind. One way to limit yourself from using too many nesting levels is to use 3 or 4 spaces for indentation. In this way, every time you nest too deeply, your code will approach the right margin faster and so you'll know that it's time to think about organizing better the code.

5.5 Use 3 or 4 spaces for indentation

Using 3 or 4 spaces for indentation makes the code more readable that when only 2 spaces are used. You may be tempted to disagree with the argument that more indentation spaces would make it difficult to have many nesting levels in the code. *Exactly*! That's the point. The code shouldn't nest too deeply (see 5.4 "Avoid deeply nested code").
Another opposing argument might be that using 3 or 4 spaces for indentation makes it hard to fit function parameters and long expressions in one line. Well, think about it.
In code that is not deeply nested, using 2 vs. 4 spaces of indentation doesn't make much difference, anyway. On the other hand, having trouble fitting function calls and expressions in one line may indicate deeper problems that simply the indentation width. These problems include:

a) Function and/or parameters names that are too long (see 2.7 "Avoid extremely long names")
b) Functions that have too many parameters (see 5.3 "Avoid having too many parameters in functions")
c) Expressions that are too long

d) Code that is too deeply nested (see 5.4 "Avoid deeply nested code")
You should avoid having names that are extremely long, as this decreases writeability *and* readability of your code. Also a function that takes too many parameters may indicate poor code design and poor usage of data structures. Try to think about grouping parameters into data structures/classes that can be passed easily as one object. Finally, breaking long expressions into shorter ones makes the code more readable.
To summarize, 3-4 spaces for indentation is preferable than 2 spaces for the following reasons:
a) Code becomes more readable
b) It discourages deeply nested code
c) It discourages extremely long identifier names
d) It discourages having functions with too many parameters
e) It discourages long expressions

5.6 Curly braces

The starting brace should be placed at the end of the parent code line. The closing brace at the same indentation level as the parent code line. It's good to use braces even when there is only one line of code to be enclosed (this makes it easier to add additional lines of code if needed).
Example:

```
if (temperature>70) {
for(int n=0; n<100; n++) {
amount = x – y * n;
}
}
```

An exception to setting the starting brace at the parent code line is for function definitions. In this case, put both braces at the beginning of the line.
Example

```
void Foo(long x, double y)
{
//.....function body.......
}
```

Another exception is when the parent code line does not fit in one line. In this case, use whatever makes it more readable.

Example

```
if (veryLongExpression < anotherVeryLongExpression
|| veryLongExpression >= oneMoreVeryLongExpression)
{
//...do this...
//...do that...
}
```

Do not put unnecessary semicolons after a closing brace.

Example

```
while(...) {
...
}; //semicolon is unnecessary here
```

6. WRITING MAINTAINABLE CODE

6.1 Don't duplicate code

Duplicate code is one of the greatest evils in programming; it is the worst form of code reuse. If you find yourself having the same code in two or more functions, pull the duplicate code out of those functions and create a new function with it; then just call this new function. Sometimes it is difficult to create a function out of the duplicate code; in those cases you can put it in a macro. However you should first try hard to create a function, because functions are in many ways superior to macros.

A common reason that programmers produce duplicate code is time pressure: "now we don't have enough time, so let's cut-and-paste for the moment, and later on we'll come back to fix it". Don't fall into this trap! It's always better to do it right the first time, mainly for two reasons:

a) The chances are that you will never come back to fix this code! Probably you will always be too busy fixing bugs or developing new code. Going back and changing working code is always a low priority, so probably it will never make it high enough in the priority list.

b) If you don't have enough time, then cut-and-paste won't buy you more time, on the opposite, it will probably cost you more time, even

in the short term! Don’t forget that we are talking about code under development that changes constantly; changes will have to be repeated in all parts of the duplicate code, bugs found in one duplicate part will have to be corrected everywhere, code becomes lengthier and more difficult to work with, etc. So, *do it right the first time!*
The main advantages of avoiding duplicate code are the following:
a) Modifications are easier because you only need to modify code in one location
b) Code is more reliable because you’ll have to check only one place to ensure the code is right
c) Bugs are reduced. Programmers who will work your code in the future will have to fix bugs in only one place; but if there’s duplicate code, then it is very common that programmers fix the bug in one place and forget to fix it in others, so bugs remain in the system
d) When putting duplicate code is in its own function, the calling code becomes shorter and in turn, more readable and understandable. For example, compare this:

```
if (node != NULL) {
while (node->next != NULL) {
node = node->next;
}
leafName = node->leaf.GetName();
}
else {
leafName = “”;
}
```

With this:

```
leafName = GetLeafName(node);
```

e) The code can become more efficient because the size of your program becomes smaller. When you replace many lines of code with one function call, more code fits into the memory and page faults can be decreased. A trip to the disk may cost as much as 2,000 function calls, so avoiding even one extra page fault can make a difference

f) Code reuse is promoted, because it is much easier for a programmer to find a function he needs and simply call it, rather than searching the code and try to cut and reuse pieces of it.

6.2 Avoid global constants

Avoid global constants that clutter the global namespace. It is much better to limit the scope of constants by declaring them inside classes. If there is a good reason for wanting them to be globally visible, then you can declare them in the main section of the class.

6.4 Avoid having public data members

Avoid as much as possible public data members. All data members should be private or protected. This allows class implementations to change easily without breaking client code. Also functions that return or provide access to data members should also be avoided for the same reasons. Try to hide class implementation; its interface shouldn't depend/reveal how the class is implemented. The clients of your class should not be aware or depend of what algorithms or what data structures you have used.

7. CODE PERFORMANCE ISSUES

7.1 Avoid passing arguments by value

It is fine to pass built-in types and value-oriented classes by value. However, many classes, particularly those that perform heap allocation or those that are large in size, are inefficient to pass by value. Use instead pass by const reference.

Example

Instead of:

```
void Foo(long x, SomeClass obj) // NOT efficient
{
obj.CleanAll();
obj.SetValue(x);
}
```

do:

```
void Foo(long x, const SomeClass& obj) // better
{
obj.CleanAll();
obj.SetValue(x);
}
```

7.2 Beware of the order in constructor initialization lists

If you use initialization list in a constructor, then order the data members in this list in the same order that they are declared in the class. The reason is that C++ initializes the variables in this list in the order in which they are declared in the class, regardless of their order in the list. This may lead to a subtle bug if the variables actually must be initialized in a specific order.

Example

```
class SomeClass {
public:
SomeClass(int var1, int var2, int var3)
: m_var1(var1), m_var2(var2), m_var3(var3) {}
// here the initialization order will be: a2, a1, a3
private:
int m_var2;
int m_var1;
int m_var3;
}
```

7.3 Don't sacrifice code quality to improve performance

Don't try to increase code efficiency by sacrificing its clarity and readability. On the long term, this will create more problems than it solves. Code with poor readability is difficult to maintain; this means more bugs and more time needed for code maintenance. This in turn results in delaying new releases and neglecting performance tuning as the programmers are too busy fixing bugs and trying to put in new features in code that is difficult to understand.
Performance should be addressed separately and it should be more focused on the "hot spots" of the code.

Don’t forget the 80-20 rule: 80% of the execution time is usually spent on only 20% of the code.
Randomly applying tricks to improve performance is bad; instead, after the code is written and debugged, do a systematic profile analysis to reveal the bottlenecks and focus only on the parts that really matter.

7.4 Don’t tweak your code – try better algorithms

Better algorithms or smarter data structures generally buy you a lot more performance than tweaking code. Changing your code to use a O(logn) algorithm will generally pay much more than spending time trying to improve and fine tune an algorithm that is inherently O(n). Smarter data structures can also have a significant impact on the performance of your program (e.g. classes that use lazy evaluation). Try to address the system performance in a higher level; don’t simply tweak code.