Source Code and Construction

CxOne Standard

CxStand_Code.doc

November 3, 2002

Construx

SOFTWARE

cxone

Advancing the Art and
Science of Commercial
Software Engineering

CONTENTS

Copyright Notice

0 Introduction

This standard provides conventions, rules, idioms, and styles for the creation of source code.

This document covers the topics of a traditional ‘coding standard’, but it also provides guidance for construction activities. This is the root coding standard for CxOne, covering general issues common across languages and technologies. Issues specific to languages and technologies are covered in separate standards contained in the *SpecificTechnologies* directory. The terms *coding standard* and *construction standard* are used interchangeably.

Software construction is a complex activity that does not lend itself to rigid application of precise rules. Coding standards provide a strong basis for efficient, high quality construction, but developers must exercise judgment about how the standards apply to unique situations.

0.1 Scope

This root standard along with the child standards for specific technologies define conventions for creating code and related construction artifacts. These construction standards work with CxOne code templates to provide a recognizable, consistent style across languages and technologies.

0.1.1 Relationship to CxOne Construction Checklists

There is some content overlap between the CxOne construction standards and the CxOne code checklists. These materials provide similar views into the same body of knowledge, but support different audiences. The checklists provide broad, descriptive summary guidelines for creating code. The coding standards provide specific, prescriptive, detailed rules for creating code. Since the underlying principles are the same, coding checklists and standards will almost always be in agreement, and will often overlap, albeit at different levels of detail.

Due to their detailed nature, construction standards are numbered like checklists. This supports easier use in review processes. Projects utilizing both CxOne coding standards and checklists should utilize the standards as their first level of issue identification, falling back to checklists items when an issue is not detailed in a standard.

0.1.2 Relationship to other CxOne Materials

CxOne construction standards do not directly address design or quality assurance activities, although some low level design and quality assurance issues are touched upon. Construction standards focus on issues that affect the structure, naming, organization, and content of individual source code modules and the elements that make up those modules.

0.2 Goals

Coding standards increase productivity through two mechanisms:

- Implementation issues are dealt with in a uniform fashion, avoiding common errors, unnecessary duplication, and multiple solutions for the same problem.
- Source code has consistent formatting and style, making it easy for engineers to read and understand, increasing efficiency, quality, and reducing maintenance costs.

Ideally, source code should appear as if only one person did all of the coding.

0.2.1 Enforcement

Creating code is not a deterministic activity. Although the term 'rules' is often used to describe the content of coding standard, many rules are often closer to 'guidelines'. The ultimate goal of project teams is to efficiently create code that meets the needs of a project – those needs may vary depending on the situation.

This standard intentionally doesn't clearly separate rules from guidelines, as the line between these should be drawn by experienced engineers based on the specific needs of the project.

Enforcement of construction standards should occur as part of formal and informal peer review processes, similar to the enforcement of checklist items. Collaborative construction helps to align a team around a consistent construction standard.

0.2.2 Education

Coding standards coupled with peer review processes provide an excellent framework to support the development of junior engineers' construction skills. The comments and explanations are particularly useful for this purpose, making the construction standards more valuable than checklists for this purpose.

0.3 Background

This standard cannot, and is not intended to, completely specify all considerations and points of style involved in writing good source code. In addition to a good understanding of the problem at hand and the language to be used, the books in *CxRef_Construction* are particularly helpful. Examples of an organization's best work can also be a powerful way to communicate, educate, and align teams to create consistent, high quality code.

Many of the rules in this standard improve code quality each time they are applied. Some rules—particularly those which cover style issues—derive their worth from consistent application throughout a project's code base. Placement of braces in C/C++ is a classic example. Normally the comments section will call out whether a rule is a style issue that has been chosen arbitrarily for consistency, or a quality-based rule that is a beneficial idiom.

Many guidelines in this standard are taken from Steve McConnell's *Code Complete*; those guidelines specifically refer to *Code Complete* for further explanation. A close reading of *Code Complete* is useful for understanding the entire standard.

0.3.1 Issues not covered

There are several topics not explicitly covered by these standards. Any issues greater in scope than a single source file or component are not covered here. Issues not covered include:

- **Style Minutia**. Not all matters of style and formatting are covered by this standard. The examples here are meant only to illustrate a particular rule, and do not implicitly define additional standards.
- **Design decisions**. Logical and physical design issues are too various and too specific to be dealt with by rules such as those presented here, and are beyond the scope of this standard. Some issues related to capturing the lowest level of design, i.e., source code comments and self-documenting code, are covered here.
- **Technology Recommendations**. The decision of when and how to apply particular technologies to a problem is beyond the scope of this document. Those choices are driven by the needs of the project and are part of the analysis and design process. This document defines how to use technologies after they are chosen.

0.4 Customization and Evolution

Like all of CxOne, the coding standards are based on Construx's industry leading knowledge and experience with software engineering. Since all projects and organizations are different, CxOne is intended as a starting point that organizations will customize and evolve to meet their unique needs.

Coding standards and templates are one of the more volatile areas of CxOne in this respect, due to close proximity to technology, and the fact that coding is very detailed work that has many dependencies on specific technologies, domains, environments, and situations.

If you already have mature coding standards, the CxOne materials can be used to fill any gaps. If you are adopting CxOne construction standards, ***CxOne Enterprise*** allow you to customize and evolve the materials to match your unique needs.

When starting from scratch and adopting CxOne coding standards, be on the lookout for modifications or additions that come up frequently, as that is an indication you should evolve your standard to best meet your current needs. Rules that are identified as primarily for style consistency are obvious candidates for immediate conversion to the prevailing style of your project or organization. The point of these types of rules is consistency, not some notion of absolute correctness.

0.5 Document Organization

0.5.1 Overview

This document contains a set of general coding and construction rules that are applicable when programming in any language or technology. The separate technology specific standards (such as *CxStand_Cpp*) cover issues specific to particular technologies, and will cross-reference each other as appropriate.

As much as possible, all rules and guidelines should be followed for all languages. Obviously this will not always be possible, and often language-specific guidelines will explicitly override general guidelines or contradict guidelines for other languages or technologies.

0.5.2 Conventions

Throughout this document, the terms 'rule' and 'guideline' are used interchangeably. Most rules have comment text to explain the rule and provide background. Many rules include example code to illustrate the implementation of the rule. Refer to the construction templates in the project source tree for additional examples of the formatting styles described by this document.

All rules are marked with a unique identifier. The first three letters indicate the section the rule is found in (General, C++, etc.), while the next two or three groups of numbers identify the position of the rule in the section.

ABC1.2.3 Rules are outlined in this style with a gray background.

Many rules are followed by comments and explanations, which are in sans-serif italics.

Some rules are followed by references, given in serif italics.

Some rules are followed by code examples, which are provided indented in the following style:

```
if ( y == x )
   {
   func1( param1, param2 );
   }
```

1 General

This section captures general construction issues.

1.1 Low-Level Design

Source code files should contain comments that document the low-level design of the module or component they implement. There will generally be several levels of low-level design in a source file: file comments, function or class comments, and implementation comments.

For any module or class, the project architecture document contains the highest-level design. Some sub-systems or components will require one or more intermediate-level design documents to further describe their design. Other areas may allow for coding to be done directly from the architecture.

In both cases, the lowest level of design documentation should be present in the code. It should be possible to follow the chain of documents between code and architecture, and not feel that there are any large holes in the understanding of the design.

GEN0 Use a Program Design Language (PDL) approach to writing code.

When implementing a module or class, define the low-level design first using comments, and then go back to add source code.

***DO NOT** write code without comments, with the intent of going back to comment it "later".*

See Code Complete, Chapter 4.

1.2 Mechanics

GEN1.1.1 Avoid use of code generation tools to create code that will be manually extended or maintained

GEN1.1.2 If utilizing code generation tools to create code that will be extended or maintained, format generated code to match the construction standards.

Code generation tools include everything from VC++ class wizards to Front Page's mangling of ASP pages. Code generation tools usually produce code that is poorly documented, poorly formatted, and poorly understood by the engineer who generated it.

GEN1.2 To generate a new module or component, start with the appropriate template. If no appropriate template exists, prefer starting from a copy of an existing module or component.

Starting from scratch with each module or component is inefficient and may introduce errors or style inconsistencies.

GEN1.3 When using copy and paste techniques, ensure comments are updated.

Nothing makes maintenance harder than incorrectly commented code.

GEN1.5 Differentiate between test code and production code.

Test code is created to exercise production code, but are not part of the shipping product. Test code may relax some of the construction standards described here, although it is still important to ensure that test applications are of high quality and easy to maintain.

An example of the type of rules that it makes sense to relax for test applications are the rules related to code generation. It may make sense to rely on code generation tools to create and maintain test applications. It does not makes sense to write test applications following a different formatting style than the construction standards just because an engineer prefers that formatting (they may not be the next person to work on it).

GEN1.6.1 Differentiate between throw-away prototype code and production code.

GEN1.6.2 If you are not certain the prototype code will be thrown away, treat as production code.

For the same reasons mentioned for test code, it may be appropriate to relax construction standards for prototype code. However, only do so if you are certain that the prototype code will not turn into the production code.

1.3 Naming

GEN1.10.1 All names should be clear and descriptive.

This includes variables, files, functions, classes, directories, etc. Don't artificially limit name length unless the technology demands it. When it comes to names, conciseness is a distant third to clarity and descriptiveness.

1.4 Formatting

GEN1.20.1 If no discernable style is present in existing code, reformat the file according to the Construction Standards.

GEN1.20.2 If a discernable style is present in existing code, but the style differs from the Construction Standards, consider reformatting to follow the Construction Standards.

Normally this will not be an issue, but code may be inherited into a project that was not produced by the current team. If there is time, reformat the entire code module to meet current coding standards. If there are reasons not to reformat the entire code module (e.g., the modifications involve two lines of a one-thousand-line module), attempt to replicate the existing formatting style as much as possible.

GEN1.20.5 Follow format examples used in the Construction Templates.

All source files should be based on the appropriate Construction Template. A Construction Template exists for each language and source file type in use on a project, and contains comment headers and preferred file layout. When starting work in a new language, create a Construction Template for that language.

GEN1.20.10 When it does not conflict with the Construction Standards, prefer following nuances of existing code format.

Anything not explicitly standardized in this standard is left to the discretion of the engineer; there are always formatting details that differ between individuals. If you are working in an existing file, make any additions or changes consistent with its formatting. Once you are done with the file, it should not be evident that two different engineers worked on it.

GEN1.20.15 Don’t create lines greater than 79 characters in length.

Lines greater than 79 characters are difficult to print without unwanted line breaks. They are also difficult to view on-screen without horizontal scrolling. If code can't be easily read, it can't be easily understood.

GEN1.20.20 Indent using groups of 4 spaces, not tabs.

Tabs are interpreted in varying ways by different applications. Using spaces avoids confusion. Most text editors, including the Visual Studio IDE, allow the tab key to be set to generate the appropriate number of spaces rather than the tab character. Some text editors will replace existing tabs with spaces when files are loaded or saved.

GEN1.20.25 Use blank lines liberally to separate and organize code.

GEN1.20.26 Prefer 1 blank line inside functions, 2 blank lines between functions, and 3-4 blank lines between file sections. Avoid more than 4 blank lines together.

Whitespace is the most effective way to separate sections of code within a block or function.

GEN1.20.30 Place a single space before and after each operator or control statement.

GEN1.20.31 Place a single space after each comma in a function parameter list.

GEN1.20.32 Place a single space after each open parenthesis and before each close parenthesis in a function call or function declaration.

GEN1.20.33 Place a single space before and after an array index.

Spaces visually separate the variables or functions being acted on from the operation being performed. C++ examples are below. In some languages (e.g., Visual Basic) the author does not have control over all spacing.

```
if ( 3 == x )
func1( param1, param2 );
int array[ MAX_NUM_THINGS ];
```

1.5 Debugging

GEN1.40.1 Use assertions to document and enforce pre-conditions, post-conditions, and invariants during testing and debugging.

In other words, use assertions to perform sanity checks on internal processes and data.

GEN1.40.5 Use trace statements to log notable events.

During initial development of a component or function, use many trace statements to make sure the code is behaving as expected. As the component or function matures, the number of trace statements should gradually decrease, so that the common application trace log isn’t filled with ‘noise’ from messages that no one uses.

GEN1.40.10 Don’t use assertions to detect and handle errors that must be handled in release code, such as errors in data from the user or external systems.

Assertions are generally disabled in release builds, so assertions won’t handle errors in release code. Besides, assertions generally halt the program, which is not a good way to handle expected run-time errors.

GEN1.40.15 Use the standard assertion and tracing implementations.

Using standard assertion and tracing implementations ensures that all assertions and tracing are handled consistently throughout the application.

GEN1.40.20 When possible use Debug and Release builds.

Debug builds allow for the use of many extra techniques to catch problems in code at run time, without causing performance problems or exposing debugging tools in release code.

1.6 Comments

GEN1.50.1 Use comments to describe the intentions of the programmer or provide a summary of a section of code.

GEN1.50.2 Do not use a comment to restate what the code does or explain the action of a section of code.

The difference between intent or summary comments and explanatory comments is generally a matter of describing 'why' rather than 'how' or 'what'.

See Code Complete, section 19.4 (p. 463).

GEN1.50.5 Use comments, especially file-level comments, to describe low-level design issues.

As described above, a source code file is also a low-level design document. Where possible, use a machine-parsable comment format such as JavaDoc to allow examination of code-level documentation separately from the code itself.

GEN1.50.10 Write comments in English.

GEN1.50.11 Use complete sentences if appropriate.

GEN1.50.12 Don't omit capitalization or punctuation.

The purpose of comments is to describe the intention of source code. The best way to do this is in English, not more code or pseudo-code.

GEN1.50.15 Remember the audience of a comment.

GEN1.50.16 Declaration comments should contain information useful to the user of the function.

GEN1.50.17 Definition comments should contain implementation details.

When interfaces or functions are split from their implementation (as in C++ .h and .cpp files), make sure the comments that someone using the code would naturally see (those in the interface declaration) describe how to use the code. Users should not see comments related to implementation of underlying interfaces, nor should they need to go to the implementation code to find a description of the use of an interface or function.

GEN1.50.20 Comment each major block of code with a brief description of its intention.

Don't say what the code is doing, say why it is doing it.

GEN1.50.25 Indent comments at the same indentation level as the code they're commenting.

Comments are just as important as the code they describe. They're not more important, so don't indent them less to call attention to them. They're not less important, so don't indent them more to invite the reader to skip over them.

GEN1.50.30 Avoid placing comments at the end of a line of code (assembly style).

End-of-line comments are less visible and more difficult to align vertically with other comments. Also, adding general comments to the end of a line often makes the line longer than the 79-character limit, forcing the use of unnecessarily terse comments.

GEN1.50.35 Prefer use of a blank line before a comment.

If an action is important enough to be commented, it's a good idea to use whitespace to highlight it.

GEN1.50.40 Use a standard marker to indicate incomplete sections of code.

Construx uses:

```
CxINCOMPLETE( <UserID>, <Description> )
```

The <UserID> parameter contains the version control ID of the person marking the code as incomplete. The <Description> parameter briefly describes the reason the code was marked incomplete. If access to a macro is not available in a language, use standard marker in a comment instead.

2 Modules and Files

This section captures issues specific to naming and structure of modules and files.

GEN2.1 Base new source files on the appropriate Construction Template.

GEN2.2 Follow the formatting styles indicated in the Construction Templates.

The standard source file templates include a comment header with placeholders for file purpose and description of contents.

GEN2.5 When naming a file, use only alphanumeric characters.

GEN2.6 Capitalize the first letter and the first letter of any words or acronyms within the filename; use lowercase for all other alphabetic characters.

This ensures consistent and easy-to-type file names.

```
CxFileName.cpp
CxFileName.h
```

GEN2.7 Use the file name to tie together related files.

Files with similar purposes should be named similarly; files with different purposes should be named differently. The directory that a file is in provides a context for the filename; an appropriate name for a file in the "DbInterface" directory is "Coordinator.cpp", but "DbInterfaceCoordinator.cpp" is redundant.

GEN2.10 Begin all source files with the standard comment header describing the purpose of the file and giving a brief synopsis of its contents.

See the Technology Specific templates for examples.

GEN2.15 A source file should contain a single module or component, at most.

Don't place more than one module or component into a single source file. There are, of course, exceptions; things like utility classes or inner classes in C++ will usually live in the same source file as their parent.

GEN2.20 When including files always use a relative path.

Using an absolute path prevents the file from being properly included in another environment, or even on another machine.

3 Variables

Issues specific to variable definition and use are covered here.

GEN3.1 Use descriptive variable names.

GEN3.2 Use variable names to describe the entity that the variable represents in the problem domain.

A variable name must be a readable, accurate, and unambiguous description of its purpose. Don't use one-character variable names. Clarity and descriptiveness is more important than conciseness.

See Code Complete, section 9.1 (p. 185).

GEN3.5 Use only alphanumeric characters in variable names.

GEN3.6 The first character of a variable name should be a lowercase alphabetic character.

GEN3.7 Capitalize the first character of words and acronyms within a variable name; use lowercase for all other alphabetic characters.

This ensures consistent and easy to type variable names.

```
balance          // One-word variable name
monthlyTotal     // Two-word variable name
exportProfile     // Variable name with acronym
```

GEN3.10 Avoid use of Hungarian notation to denote underlying variable type.

Using Hungarian style notation to tie variable type to variable name is contrary to the principles of data hiding and encapsulation. Using Hungarian notation can distract from the main purpose of the variable name, which is to describe what the data contained in the variable means. Hungarian notation can also make maintenance difficult when a variable type changes.

Circumstances where Hungarian notation can be useful are standardized in this and the language-specific construction standards.

GEN3.15 Declare a variable as close as possible to its first use.

GEN3.16 Initialize a variable when declared.

This limits variable scope and ensures a known value for the variable, and minimizes the number of variables in use at any point in the source code. It also improves readability by reducing the physical distance (in number of lines) between where a variable is declared and where it is used.

See Code Complete, section 10.1.

GEN3.20 Precede a variable declaration with a brief comment describing the variable's purpose.

Even the most descriptive variable names are terse. Use a comment to explain the variable's purpose more fully.

GEN3.25 Prefix global variable names with 'g_'.

This should not be construed as encouraging the use of global variables, which should be used sparingly if at all.

3.1 Constants

GEN3.40.1 Use upper-case alphanumeric characters in constant names; separate words within a constant name by underscores.

Identification of constants is important, since one cannot assume that any other type of identifier always has the same value.

```
THIS_IS_A_CONSTANT
```

GEN3.40.5 Never use a bare numeric constant (magic number), even if it is only used once. Always declare a symbolic name for numeric constants.

A 'magic number' provides no information about its purpose or meaning, and is essentially unmaintainable. In certain cases, such as initializing variables or boundary checking, the bare numbers '0' and '1' are acceptable.

GEN3.40.10 Use constants for multiple use text strings.

GEN3.40.11 Prefer use of constants for single use text strings.

Text strings that will be displayed to the user should always be pulled out into string constants or string tables. This allows them to be translated or updated as needed.

4 Expressions

Issues specific to use of expressions are covered here.

GEN4.1 Don't use assignment in conditional expressions.

In languages where the assignment and comparison operators differ, such as C++, an assignment in a conditional expression is often a typographical error--the programmer meant to use the comparison operator. Avoid this ambiguity by not performing assignment inside conditional expressions; that's not what they're meant for anyway. If this guideline is followed, an assignment in a conditional is unambiguously an error.

GEN4.5 When testing for equality, place the constant value on the left side of the comparison operator.

This protects against inadvertent assignment within a conditional expression.

GEN4.10 When testing for inequality, place the values to be compared in number-line order.

GEN4.11 If multiple inequality tests are being performed with the same variable, place the inequality tests in number-line order.

Number-line means that the greatest quantity is always placed on the right, as on a number line. In other words, always use < and <=, not > or >=. This provides an explicit ordering for comparison tests, and makes the intent of the test easier to visualize.

4.1 Statements

GEN4.15.1 Break statements longer than 79 characters after the last operator within the 79-character limit. Indent the next line one level. If the new line also exceeds 79 characters, insert another line break after the last operator within the 79-character limit, but do not indent the next line further. Continue until all lines of the statement are less than 79 characters in length.

This ensures a consistent style that is easily viewed and printed.

```
If (PreliminarySize = 0 And CalculatedMeanSizeInLoc(Module) <> 0) Or _
    (PreliminarySize = CalculatedMeanSizeInLoc(Module)) Then
    Call CalibrationData.GetProjectPhaseFromId( _
        ProjectCharacteristics.CurrentProjectPhase, ProjectPhase)
End If
```

5 Control Structures

Issues specific to control structures are covered here.

GEN5.1 Indent blocks using the 'begin-end block boundaries' style.

Place block delimiters (such as '{' and '}') on their own lines and indent them at the same level as the code contained in the block.

```
if ( true == success )
    {
    // Do something.
    }
```

See Code Complete, section 18.3 (p. 415).

GEN5.5 Use descriptive names for loop indices.

Name the loop index after the thing being iterated over, such as 'row' or 'col'. If the loop is simple and small, traditional loop indices such as 'i', 'j', and 'k' are acceptable, although not preferred.

See Code Complete, section 9.2 (p. 190).

GEN5.6 Avoid exiting a loop using a break or return statement.

Like multiple return statements in a function, break statements add complexity to a loop by adding additional code paths that are not readily apparent. Consider the issues carefully before using break or return statements to exit a loop.

See Code Complete, section 15.2 (p329).

GEN5.10 Prefer `for` loops when they're appropriate.

`for` *loops place all loop-control code in one place, which makes the loop more readable and maintainable.*

See Code Complete, section 15.2 (p. 331).

GEN5.11 Don't include statements which do not affect the loop duration or exit condition in a `for` loop control expression.

The statements and expressions in a `for` *loop head should either initialize the loop, terminate the loop, or move the loop towards termination. If you're including other statements in the* `for` *loop header, you should probably be using a while loop.*

See Code Complete, section 15.2 (p. 332).

GEN5.15 Don't use goto.

Using gotos violates structured-programming principles, prevents compiler optimizations, and makes the resulting code difficult to format. Don't use goto unless its use demonstrably reduces code complexity in a way not possible with other control structures.

See Code Complete, section 16.1 (p. 347).

6 Functions

Issues related to defining and using functions are covered here.

GEN6.1 Follow the same form as variable names when naming functions, with the distinction that the first character of a function name should be capitalized.

This distinguishes function names from variable names, while providing a consistent format for both.

```
CxFunctionName()
```

GEN6.5 For a function whose main purpose is to perform an action, use a verb-object function name.

Use specific, problem domain-centered verbs and objects. Note that an object method that performs an action on the object itself should just be a verb.

```
PrintDocument( document );
document.Print( );
```

GEN6.10 For a function whose main purpose is to return a value, use a name that describes the value returned.

Again, use specific, domain-centered names.

```
if ( FILE_OPEN == file.State() )
```

See Code Complete, section 5.2 (p. 80).

GEN6.15 Precede each function declaration with a comment describing the intent of the function, the purpose of each argument, and the function's return value.

Groups of small related functions may be grouped under the same comment block.

GEN6.20 Base new function declarations and definitions on the appropriate standard template.

The standard templates for function declarations include a comment header with placeholders for function intent, arguments, and function return value.

GEN6.25 Avoid using multiple return statements to exit a function, except for egregious pre-conditions

Multiple returns add complexity, making it more difficult to determine the code path. They prevent checking the return value or post-condition of a function at a single location. They also make debugging more difficult for the same reason: there is no longer a single place to set a breakpoint on function exit. Consider these issues before using multiple returns.

In some cases where there are 1 or more pre-conditions that result in immediate failure, the use of returns at the beginning of the function can improve readability by preventing potentially deep logic clauses that do not contribute to the main purpose of the function.

See Code Complete, section 16.2 (p. 360).

7 Quick Reference

0 Introduction

1 General

GEN0	Use a Program Design Language (PDL) approach to writing code.
GEN1.1.1	Avoid use of code generation tools to create code that will be manually extended or maintained
GEN1.1.2	If utilizing code generation tools to create code that will be extended or maintained, format generated code to match the construction standards.
GEN1.2	To generate a new module or component, start with the appropriate template. If no appropriate template exists, prefer starting from a copy of an existing module or component.
GEN1.3	When using copy and paste techniques, ensure comments are updated.
GEN1.5	Differentiate between test code and production code.
GEN1.6.1	Differentiate between throw-away prototype code and production code.
GEN1.6.2	If you are not certain the prototype code will be thrown away, treat as production code.
GEN1.10.1	All names should be clear and descriptive.
GEN1.20.1	If no discernable style is present in existing code, reformat the file according to the Construction Standards.
GEN1.20.2	If a discernable style is present in existing code, but the style differs from the Construction Standards, consider reformatting to follow the Construction Standards.
GEN1.20.5	Follow format examples used in the Construction Templates.
GEN1.20.10	When it does not conflict with the Construction Standards, prefer following nuances of existing code format.
GEN1.20.15	Don’t create lines greater than 79 characters in length.
GEN1.20.20	Indent using groups of 4 spaces, not tabs.
GEN1.20.25	Use blank lines liberally to separate and organize code.
GEN1.20.26	Prefer 1 blank line inside functions, 2 blank lines between functions, and 3-4 blank lines between file sections. Avoid more than 4 blank lines together.
GEN1.20.30	Place a single space before and after each operator or control statement.
GEN1.20.31	Place a single space after each comma in a function parameter list.
GEN1.20.32	Place a single space after each open parenthesis and before each close parenthesis in a function call or function declaration.
GEN1.20.33	Place a single space before and after an array index.
GEN1.40.1	Use assertions to document and enforce pre-conditions, post-conditions, and invariants during testing and debugging.
GEN1.40.5	Use trace statements to log notable events.
GEN1.40.10	Don’t use assertions to detect and handle errors that must be handled in release code, such as errors in data from the user or external systems.
GEN1.40.15	Use the standard assertion and tracing implementations.
GEN1.40.20	When possible use Debug and Release builds.
GEN1.50.1	Use comments to describe the intentions of the programmer or provide a summary of a section of code.
GEN1.50.2	Do not use a comment to restate what the code does or explain the action of a section of code.

GEN1.50.5	Use comments, especially file-level comments, to describe low-level design issues.
GEN1.50.10	Write comments in English.
GEN1.50.11	Use complete sentences if appropriate.
GEN1.50.12	Don’t omit capitalization or punctuation.
GEN1.50.15	Remember the audience of a comment.
GEN1.50.16	Declaration comments should contain information useful to the user of the function.
GEN1.50.17	Definition comments should contain implementation details.
GEN1.50.20	Comment each major block of code with a brief description of its intention.
GEN1.50.25	Indent comments at the same indentation level as the code they’re commenting.
GEN1.50.30	Avoid placing comments at the end of a line of code (assembly style).
GEN1.50.35	Prefer use of a blank line before a comment.
GEN1.50.40	Use a standard marker to indicate incomplete sections of code.

2 Modules and Files

GEN2.1	Base new source files on the appropriate Construction Template.
GEN2.2	Follow the formatting styles indicated in the Construction Templates.
GEN2.5	When naming a file, use only alphanumeric characters.
GEN2.6	Capitalize the first letter and the first letter of any words or acronyms within the filename; use lowercase for all other alphabetic characters.
GEN2.7	Use the file name to tie together related files.
GEN2.10	Begin all source files with the standard comment header describing the purpose of the file and giving a brief synopsis of its contents.
GEN2.15	A source file should contain a single module or component, at most.
GEN2.20	When including files always use a relative path.

3 Variables

GEN3.1	Use descriptive variable names.
GEN3.2	Use variable names to describe the entity that the variable represents in the problem domain.
GEN3.5	Use only alphanumeric characters in variable names.
GEN3.6	The first character of a variable name should be a lowercase alphabetic character.
GEN3.7	Capitalize the first character of words and acronyms within a variable name; use lowercase for all other alphabetic characters.
GEN3.10	Avoid use of Hungarian notation to denote underlying variable type.
GEN3.15	Declare a variable as close as possible to its first use.
GEN3.16	Initialize a variable when declared.
GEN3.20	Precede a variable declaration with a brief comment describing the variable’s purpose.
GEN3.25	Prefix global variable names with ‘g_’.
GEN3.40.1	Use upper-case alphanumeric characters in constant names; separate words within a constant name by underscores.
GEN3.40.5	Never use a bare numeric constant (magic number), even if it is only used once. Always declare a symbolic name for numeric constants.
GEN3.40.10	Use constants for multiple use text strings.
GEN3.40.11	Prefer use of constants for single use text strings.

4 Expressions

GEN4.1	Don't use assignment in conditional expressions.
GEN4.5	When testing for equality, place the constant value on the left side of the comparison operator.
GEN4.10	When testing for inequality, place the values to be compared in number-line order.
GEN4.11	If multiple inequality tests are being performed with the same variable, place the inequality tests in number-line order.
GEN4.15.1	Break statements longer than 79 characters after the last operator within the 79-character limit. Indent the next line one level. If the new line also exceeds 79 characters, insert another line break after the last operator within the 79-character limit, but do not indent the next line further. Continue until all lines of the statement are less than 79 characters in length.

5 Control Structures

GEN5.1	Indent blocks using the 'begin-end block boundaries' style.
GEN5.5	Use descriptive names for loop indices.
GEN5.6	Avoid exiting a loop using a break or return statement.
GEN5.10	Prefer `for` loops when they're appropriate.
GEN5.11	Don't include statements which do not affect the loop duration or exit condition in a `for` loop control expression.
GEN5.15	Don't use goto.

6 Functions

GEN6.1	Follow the same form as variable names when naming functions, with the distinction that the first character of a function name should be capitalized.
GEN6.5	For a function whose main purpose is to perform an action, use a verb-object function name.
GEN6.10	For a function whose main purpose is to return a value, use a name that describes the value returned.
GEN6.15	Precede each function declaration with a comment describing the intent of the function, the purpose of each argument, and the function's return value.
GEN6.20	Base new function declarations and definitions on the appropriate standard template.
GEN6.25	Avoid using multiple return statements to exit a function, except for egregious pre-conditions

7 Quick Reference