1. Introduction 3

1.1 Purpose 3

1.2 Scope 3

1.3 Rules 3

1.4 Organization of this document 3

2. Naming Conventions 4

2.1 Meaningful Names 4

2.2 Naming Conventions 5

3. Readability and Understandability 6

3.1 Comments 6

3.1.1 Indentations 6

4. C/C++ Coding 7

4.1 Organizing the code 7

4.2 Class Interface 8

4.2.1 Encapsulation 8

4.2.2 Argument Passing and return Values 8

4.2.3 const Correctness 8

4.3 Structs 9

4.4 Object Life Cycle 9

4.4.1 Constructor and Destructor 9

4.4.2 Copying Objects 9

4.5 new, delete and other Resources 10

4.6 Declaration and Definition of Variables 10

4.7 Constants 11

4.8 Static and Global Objects 11

4.9 Control Flow 12

4.10 Conversions 14

4.11 Error Handling, Assertions, and Exceptions 14

4.12 Parts of C++ to Avoid 15

4.13 Maintainability 15

4.14 Portability 16

4.15 Minimize Coupling 16

# Introduction

## Purpose

The purpose of this document is to define a C/C++ coding standard that should be adhered to when writing C/C++ code. The standard is intended to allow multiple programmers to produce code that is readable and understandable by all programmers. This is accomplished with consistent style and certain guidelines listed is this document.

## Scope

This guidance applies to all handwritten C/C++ code generated specifically for programming the Dynamic GUI Application

## Rules

A keyword will be used in each of the rules to indicate the type of rule that it is. The three rule types are:

* Should - Guideline, use judgment when not to follow.
* Shall - Mandatory.
* May - Permissible if requirements of the exception are met.

## Organization of this document

The document is organized as follows:

* Chapter 1: Introduction - this chapter
* Chapter 2: Naming - items on naming identifiers, with explanation and examples
* Chapter 3: Readability and Understandability – items on readability and consistent look of the code
* Chapter 4: C++ Coding - items regarding the syntax and semantics of C++ code

Each item stated in chapter 2 to 4 comprises at least two entities, an identifier and an item title. The identifier is formed by two or three letters and a number (e.g. NM3); the first letter indicates to which section (Naming, Readability or Coding) the item belongs, the second and third letter indicates the subsection, while the number simply represents the order within the subsection. This kind of identification should allow a minimal impact on the item’s numbering during the maintenance of the standard, i.e. in the possible cases in which items are added or removed. Whenever possible and appropriate, an explanation or an example has been added to the individual item, which expands the item title and clarifies its meaning and scope.

# Naming Conventions

This section contains a set of conventions on how to choose, write and administer names for all entities over which the programmer has control. This guarantees that programs are easier to understand, read and maintain.

Meaningful Names

NM1 Pronounceable names should be used

NM2 If abbreviations are to be used, a standard abbreviation list shall be generated

Any abbreviation used shall be captured in an abbreviation list within the software module using Doxygen such that the list may be generated as a document.

NM3 Names should be in English and self-descriptive.

NM4 Names of functions, structures, and variables should be meaningful and reflect their intended use.

Use “Is” as a boolean prefix, example: IsReady, IsFaulted, IsDone

Use nouns for variable names such as PatientRecord or GraphicsManager.

Use verbs for functions such as PlanFileOpen or StatisticsShow.

NM5 Similar names should be avoided

NM6 Namespaces may be used to avoid name conflicts.

You can avoid name clashes by declaring and defining names (that would otherwise be global) inside namespaces.

NM7 File scope may be used to avoid name conflicts. (C only)

Use “static” to keep file only functions and “global” variables local to the file

Naming Conventions

NC1 The following naming convention should be followed

|  |  |  |
| --- | --- | --- |
| ***Catagory*** | ***Convention*** | ***Example*** |
| Function | Pascal Case | FuntionName() |
| Variable | Camel Case | variableName |
| Class | \_C post fix | MyClass\_C |
| Interface | \_I post fix | MyInterface\_I |
| Struct  typedef (non-basic) | \_T post fix | MyStruct\_T  MyType\_T |
| #define  Constants  enum content  Macros | All caps with ‘\_’ separating words | MY\_DEFINE  MY\_CONST  STATE\_1, STATE\_2  MY\_MACRO |
| Files | Alphanumeric (upper and lower case) and ‘\_’ are all allowed | MyFile\_1.cpp |
| File Extensions | .c – C files  .cpp – C++ files  .h – header files | Cfile.c  CppFile.cpp  CppHeader.h |

typesdef of basic types of int, short, long, float and double do not need the \_T post fix

NC2 C++ member variable shall be post fixed with a member indicator

Example: “\_” for \_MemberVariable or mMemberVariable

NC3 Code file suffixes shall be “.h”, “.c” and “.cpp” for header, “C” source and “C++” source files

Readability and Understandability

Comments

RC1 Doxygen tags should be used with the intent of utilizing Doxygen output to contribute to Software Design Description

The author has general discretion to use the tags required to properly document, however the following rules describe some minimal Doxygen guidelines

RC2 Files should begin with a /brief description of the file’s purpose and or category of functions

RC3 Descriptions should reside in the .c/.c++ file as opposed to the .h file when possible

Exceptions may include:

* Pure virtual functions
* #defines
* Class members variables
* Struct members variables
* Enum values

RC4 Revision detail, author and dates should NOT be documented in the files

Source control tracks these details with the best accuracy.

RC5 The following Doxygen tags should be used to document a typical function

/params Function argument descriptions

/return Return value description

/brief Function high level description of what the function will accomplish

/detail Description of how the function accomplishes its mission

///< Inline comments

/\*\* \*\*/ Comment blocks

RC6 Doxygen extensions may be used to enable better document output

/dot graphviz figures

/msc mscgen figures

RC7 Trivial and Obviously named functions may be exempt from full documentation

If comments do not add value, don’t add them.

Example: PrintErrorString(int errorNum) is self descriptive

Indentations

RI1 Indentations shall be consistent within the file, spaces will be inserted for tabs

# C/C++ Coding

Organizing the code

CO1 File content should contain functions related in purpose or scope

CO2 Each header file shall include all other header files necessary to fully define the items declared within it

CO3 Header files shall be “stand-alone” in that they do not require a special order of inclusion

CO4 Header files shall begin and end with multiple-inclusion protection.

Example

|  |
| --- |
| **#ifndef** NAME\_OF\_HEADER\_FILE\_H  **#define** NAME\_OF\_HEADER\_FILE\_H  // file content  **#endif** // NAME\_OF\_HEADER\_FILE\_H |

CO4 Each file should be compiled without generating compiler warnings.

If the source of the warning is the author’s code, author should make changes necessary to remove the warning or comment why the warning occurs.

CO5 Development Tool configurations shall be saved in source control repository for consistent deployment and recreation

This includes saving the specific project, make file, and other build configuration files or using a tool like CMAKE to control the distribution of build environments.

CO6 Forward declarations may be used instead of including a header file if a header files is impractical.

This practice is not preferred

|  |
| --- |
|  |

Class Interface

Encapsulation

CI1 Data members should be declared private or protected.

Author should use get/set inline methods for public access of data

Argument Passing and return Values

CI2 A function’s interface should not contain more than 6 parameters.

CI3 All parameters in a function prototype shall be given an identifier and the same names shall be used in the function definition

CI4 Arguments of built-in types should be passed by value unless the function modifies them.

A good practice is to pass built-in types such as UInt8, Int32, Float64 by value because it is cheap to copy these variables. This recommendation is also valid for some objects of classes that are cheap to copy, such as simple aggregates of very small number of built-in types.

CI5 Class and structure type arguments should be passed by reference or pointer

const Correctness

CI6 A pointer or reference argument passed to a function shall be declared as const if the function does not change the object bound to it.

CI7 Class methods / member functions that do not modify the state of the object should be marked as const.

Labeling a member function as “const” states that it provides “read-only” access to the object.

Structs

CS1 structs shall only contain public attributes and shall not provide methods.

Object Life Cycle

This section contains rules regarding how objects are created, initialized, copied, assigned and destroyed.

Constructor and Destructor

CL1 Non-static class data members shall be initialized through the member initialization list or by the constructor.

CL2 The initializer list order should be the same as the order of the declaration in the header file: first base classes, then data members

CL3 A base class shall declare a virtual destructor.

Copying Objects

CL5 The copy constructor and the copy assignment operator shall be declared protected or private in an abstract base class.

Prevents creation or copying of an object that only represents the interface and doesn’t include the underlying implementation.

CL7 If you provide a copy constructor then you should provide an assignment operator also and vice versa.

Implementation of a copy constructor is straight forward once you have implemented an assignment operator.

Example

|  |
| --- |
| // Implements a copy constructor using the assignment operator.  Line\_C::**Line\_C**(**const** Line\_C &line)  {  **this**->**operator**=(line);  } |

CL8 The argument to a copy constructor and to an assignment operator shall be a const reference.

This ensures that the object being copied is not altered by the copy or assign.

CL9 The assignment operator shall return a const reference to \*this.

Both the standard library type and the built-in types behave this way. This ensures that:

a = b = c;

will assign c to b and then b to a as is the case with built in objects.

CL10 The assignment operator shall work correctly when the left and right operands are the same object (self assignment).

a = a must function correctly. This requires some care when writing assignment code, as the case when left and right operands are the same may require that most of the code is bypassed.

Example

|  |
| --- |
| Line\_C &Line\_C::**operator**= ( **const** Line\_C &Line )  {  **if** (**this** != &Line) // be aware of self-assignment  {  // implementation of operator=  }  } |

new, delete and other Resources

CN1 C++: new and delete should be used instead of malloc, calloc, realloc and free.

C++ should avoid all memory-handling functions from the standard C-library (malloc, calloc, realloc and free) because they do not call constructors for new objects or destructors for deleted objects.

CN2 C++: Wherever possible, classes should be instantiated off the stack without using new/delete so that they are automatically cleaned up when they go out of scope.

Declaration and Definition of Variables

CD1 Each variable should be declared or defined on a separate line.

Declaring multiple variables on the same line is not recommended. The code will be difficult to read and understand.

CD2 Each variable shall be defined with the smallest possible scope and immediately initialized.

CD3 Variables shall not be given the same name as variables with a larger scope.

CD4 Objects or functions should not be defined in a header file.

Header files exist to declare objects, functions, typedefs, macros and macro functions. Only C/C++ files should contain executable source code.

Exceptions:

* Simple single line C++ inline member functions
* Template functions

Constants

CC1 Symbolic values (const or #define) with a meaningful name shall be defined instead of using numeric values or string literals if the value is used in multiple locations.

Singular, hard coded values shall be commented.

CC2 An enum should be used for related constants rather than const or #define.

Example

|  |
| --- |
| **enum** MyState { HALTED, STARTING, RUNNING, PAUSED }; |

CC3 An “F” suffix shall be used for floating point constants that are intended to be used in objects of underlying type float

C assumes a floating point constant should be a double unless otherwise specified. This requires that a constant be converted from a double back to a float.

Example

|  |
| --- |
| **#define** MY\_FLOAT\_CONST 15.62F  F32\_T fileScopeVar = 0.0F; |

CC4 An “U” suffix shall be used for unsigned constants that are intended to be used in objects of underlying unsigned type

Example

|  |
| --- |
| **#define** MY\_UINT\_CONST = 4000000000U  UInt32 MyUint32 = 4000000000U; |

Static and Global Objects

CG1 The number of global variables should be minimized.

If necessary, encapsulate those variables in a class or in a namespace.

CG2 Use of global functions should be avoided.

In C++ code global functions are needed only for rather few specialized problems.

*Examples*:

You need a global function for implementing symmetric operators like operator+. This is the only way to get conversions of the left operand of binary operations to work. It is common in implementing the symmetric operator to call the corresponding asymmetric binary operator.

Global functions are also needed for implementing callbacks, if the caller is based on C (e.g. X11/MOTIF).

CG3 File scope functions and variables shall be declared as static.

Otherwise name clashes are likely. (CG4 in **Error! Reference source not found.**)

Control Flow

CCF1 Functions should not be overly complex.

The number of possible paths through a function, which depends on the number of control flow primitives (if, while, for, etc.), is the main source of function complexity. Therefore, you should be aware that heavy use of control flow primitives will make your code difficult to maintain.

Static code analysis should be used on all source files. Functions with high cyclomatic complexity should receive efforts to simplify

CCF2 The statement forming the body of an if, else, while, for, do…while or switch statement shall be a compound statement ( { ... } ), even if it is empty.

CCF4 if … else if constructs shall be terminated with an else clause

Example

|  |
| --- |
| **if** ( condition )  {  statement;  }  **else** **if** ( condition )  {  statement;  }  **else** **if**  {  statement;  }  **else**  {  // comment why no action  } |

CCF5 K&R style brackets shall not be used

Example

|  |
| --- |
| **// Good**  **if** ( condition )  {  ConditionTrueStatement;  }  **else**  {  ConditionFalseStatement;  } |
| **// Bad**  **if** ( condition ) {  ConditionTrueStatement;  }  **Else {**  ConditionFalseStatement;  } |

CCF6 All switch statements shall have a default clause as the last clause.

CCF7 All non-empty switch clauses that fall through or are fallen into from another non-empty clause shall be commented

Example

|  |
| --- |
| // Good  **switch** ( state )  {  **case** INITIAL:  statementA;  // This falls through to the RUNNING case  **case** RUNNING:  // The INITIAL case falls through to this case  statementB;  **break**;  **default**:  statementC;  **break**;  } |

CCF8 The number of return statements in a function should be minimized.

Where practical, have only one return from a function or method as the last statement. Otherwise minimize the number of returns.

CCF10 Floating point expressions shall not be tested for equality or inequality

Floating point variables are subject to rounding and truncation errors and an exact match may never occur

CCF11 Assignment operators shall not be used in expressions that yield a Boolean value.

Poor readability

Example

|  |
| --- |
| // Bad  **if** ( x = y )  {  **foo**( );  };  // Good  x = y;  **if** ( x != 0 )  {  **foo**( );  } |

Conversions

CV1 Narrowing conversions; conversions between signed and unsigned; and conversions from floating point to integers shall be type cast.

Explicit type conversion provide a visible indication to the reader that a conversion takes place and suggests that the programmer was aware of the conversion.

CV2 A pointer shall not be converted to an integer or pointer to a different type without a type cast.

Indicates the programmer understands what is happening and makes it easier to inspect for issues.

CV3 The new cast operators (dynamic\_cast and static\_cast) should be used rather than the C-style casts (C++ only).

The new cast operators give the user a way to distinguish between different types of casts. The behavior is well-defined.

CV4 The const or volatile qualification shall not be cast away

Error Handling, Assertions, and Exceptions

CH3 New exceptions should not be created

Exceptions can be tricky to use reliably and the benefits do not outweigh the risks.

CH4 All exceptions thrown by a library function shall be caught by the calling function.

The use of library functions is often unavoidable, and any exceptions they throw need to be dealt with as soon as possible. Unhandled exceptions could well cause program termination.

Parts of C++ to Avoid

Here below a set of different items are collected. They highlight parts of the language that should be avoided, because there are better ways to achieve the desired results.

CA1 iostream functions should be used rather than those defined in stdio.

scanf and printf are not type-safe and they are not extensible. Use operator>> and operator<< instead.

CA2 Functions should not be defined using ellipsis notation (“…”)

Ellipsis notation allows a variable number of arguments to be passed to a function. Functions with an unspecified number of arguments should be avoided because they are a common cause of bugs that are hard to find and bypass standard type-checking. Note that this only rules out creating new functions with this format, and does not rule out the use of library functions (like printf, scanf, etc.) that make use of this

CA4 NULL shall be defined as the integer constant 0.

CA7 The bool type of C++ shall be used for for booleans.

CA9 goto shall not be used.

CA10 The unbounded functions of *<cstring>*/*<string.h>* should not be used.

strcpy, strcmp, strcat, strchr, strspn, etc. functions within this library can read or write beyond the end of a buffer resulting in undefined behavior. Used the bounded versions instead (e.g. strncpy, strncmp, strncat, etc.)

Maintainability

CM1 Commented out code shall be deleted

Dead code can be confusing, especially during maintenance of the software. If there is a need to keep some code around, check a version with the software in it into version control.

Portability

CP1 Headers supplied by the implementation (system or standard libraries header files) shall go in < > brackets; all other headers shall go in “ ” quotes.

CP2 Absolute directory names should not be used in include directives

It is better to specify to the build environment where files may be located.

Relative to the build environment is okay and provides structure to large projects

CP3 Include file names shall always be treated as case-sensitive.

Some operating systems, Linux, have case-sensitive file names.

CP4 The basic types of int, short, long, float and double should not be used, descriptive typedefs for the specific length types should be created..

CP5 The order of evaluation of arguments to a function shall not be depended on.

The order of evaluation of function arguments is compiler dependent.

In particular never use ++, -- operators on arguments in function calls. The behavior of foo(a++, vec(a)); is platform dependent.

Example

|  |
| --- |
| // Bad  **func**( **f1**(), **f2**(), **f3**() ); // f1 may be evaluated before f2 and f3,  // but don't depend on it!  // Good  a = **f1**();  b = **f2**();  c = **f2**();  **func**( a, b, c ); |

Minimize Coupling

CMC1 Hardware and external interfaces should be isolated to a small number of classes or functions

CMC2 All global variables should not be extern’ed in one header file.

A single global variable header file creates confusion of ownership of the variable, makes it less likely that unneeded global variables will get removed.

CMC3 Global function prototypes shall not all be located in one header file.

Same reasoning as CMC3

CMC4 C functions that are internal / static to a file should be prototyped in the C file.

No reason to expose the function