**Software Standards and Procedures Manual**

|  |  |
| --- | --- |
|  |  |
|  |  |
| **Version:** | 1.4 |
| **Date of Issue:** | 2014-06-23 |

**Software Standards and Procedures Manual**

**Identification of the document**

|  |  |
| --- | --- |
| **Identification** |  |
| **Version** | 1.4 |
| **Name** | Software Standards and Procedures Manual |
| **Authors** | Lynn Dabrowski, Jaroslav Mlejnek |
| **Project** |  |
| **Project Manager** |  |
| **File** | coding\_standard.docx |
| **System:** | MSS |

**Description of Content**

|  |
| --- |
| This document contains a definition of the standards and procedures that will be followed during development of new software components in the software development group. |

**Document History**

| **Revision** | **Description** | **Date** | **Author** |
| --- | --- | --- | --- |
| 1.0 | Creation of Document | 2011-03-11 | Lynn Dabrowski |
| 1.1 | Revision of coding rules. | 2012-07-05 | Jaroslav Mlejnek |
| 1.2 | Updated coding rules according to team discussion. | 2012-09-10 | Jaroslav Mlejnek |
| 1.3 | Coding rules split to mandatory and recommended. | 2012-11-01 | Jaroslav Mlejnek |
| 1.4 | Removed outdated rules, added recommendations. | 2014-06-23 | Jaroslav Mlejnek |

Table of Content

[1 Overview 4](#_Toc337638657)

[2 Reference Documents 4](#_Toc337638658)

[3 File Naming Conventions 5](#_Toc337638659)

[4 Source Code Naming Conventions 6](#_Toc337638660)

[5 Source File Commenting Conventions 8](#_Toc337638661)

[6 Statement Conventions 9](#_Toc337638662)

[6.1 Types 9](#_Toc337638663)

[6.2 Variables 9](#_Toc337638664)

[6.3 Loops and Conditionals 9](#_Toc337638665)

[6.4 Miscellaneous 10](#_Toc337638666)

[6.5 Source Code Layout 11](#_Toc337638667)

# Overview

This document contains a definition of the standards and procedures that will be followed during development of new software components in the software development group. These include file naming conventions, source code formatting, version numbering, and source control methods. The goal of these guidelines is to create uniform coding habits among software personnel in the software engineering department so that reading, checking, and maintaining code written by different people becomes easier. The intent of these standards is to define a natural style and consistency.

# Reference Documents

International Standard ISO/IEC 14882 *Programming Languages – C++*

# File Naming Conventions

Table 1 summarizes the naming conventions of software source files.

| **File Naming Conventions** | |
| --- | --- |
| **Rule** | **Example** |
| C++ header files should have the extension .h. It is not recommended to have header file without extension. | my\_class.h |
| C++ Source files should have the extension .cpp. | my\_class.cpp |
| C Source files should have the extension .c. | my\_source.c |
| Source C, C++ and header files should have names that clearly indicate the module functions that they represent. | rx\_serial\_data.cpp |
| A class should be declared in a header file and defined in a source file where the name of the files matches the name of the class.  It is possible to have multiple classes in one file, only if other classes are used only by main class for private usage. | my\_class.h my\_class.cpp |
| Files of the current project should be included in the sources using quotes syntax. | #include "my\_application.h" |
| Files of components and third party components should be included in the sources using brackets syntax. The name of the component should be included in the path as well. | #include <component/file.h> |
| The symbol / should be used as a path separator in include directives. | #include <com/private/file.h>  // NOT: // #include <com\private\file.h> |

Table 1 File Naming Conventions

# Source Code Naming Conventions

Table 2 summarizes the naming conventions of software components within source files.

| **Source Code Naming Conventions** | |
| --- | --- |
| **Rule** | **Example** |
| Names representing types must be in mixed case starting with upper case. | Line, SavingsAccount |
| Variable names must be in mixed case starting with lower case. | line, savingsAccount |
| Named constants (including enumeration values) must be all uppercase using underscore to separate words. | const int MAX\_ITERATIONS = 25; const Colors USED\_COLOR = Colors::Red; const double PI = 3.14; |
| Names representing functions must be verbs written in mixed cases starting with upper case. | ComputeTotalWidth() |
| Names representing methods must be verbs and written in mixed case starting with upper case. | GetName(), ComputeTotalWidth() |
| Names representing namespaces should be in mixed case starting with upper case. | Model::Analyzer, Io::IoManager, Common::Math::Geometry |
| Names representing template types should be in mixed case starting with upper case. Single uppercase letter is also allowed. | template<class Type> ... template<class C, class D> ... |
| Abbreviations and acronyms must not be uppercase when used as name | ExportHtmlSource(); // NOT: ExportHTMLSource();  OpenDvdPlayer(); // NOT: OpenDVDPlayer(); |
| Private and protected class variables must have \_ prefix. After \_ prefix follows an lowercase letter. | class SomeClass  { private:   int \_length;   double \_size; }; |
| Public class variables must be avoided. Use private variable and appropriate getter and setter. | class SomeClass  { public:  void SetSize(double value)   {   \_size = val;  }  double GetSize() const   {  return \_size;  } private:   double \_size; }; |
| Public structure variable names must be in mixed case starting with upper case. | struct Array {  int Size;  ::MyObject \* MyObject; }; |
| Generic variables should have the same name as their type.  It is possible to use well-known abbreviation, such as db for “database”, col for “column” etc. | void SetTopic(Topic \* topic) // NOT: void SetTopic(Topic \* value) // NOT: void SetTopic(Topic \* aTopic) // NOT: void SetTopic(Topic \* t)  void Connect(Database \* database)  // POSSIBLE: void Connect(Database \* db) // NOT: void Connect(Database \* oracleDB) |
| The name of the object is implicit, and should be avoided in a method name. | line.GetLength(); // NOT: line.GetLineLength(); |
| The name of the object should be full name without any abbreviations instead of well-known abbreviations listed below:  prev (for previous), col (for column), evnt (for event because event is usually keyword), db (for database) |  |
| Variables representing GUI components should be suffixed by the component type name. | mainWindow, propertiesDialog, widthScale, loginText, mainForm, fileMenu, minLabel, exitButton, yesToggle |
| The prefix n should be used for variables representing a number of objects. It is also possible to use count suffix. | nPoints, nLines  // POSSIBLE: pointsCount, linesCount  // NOT: // numberOfPoints, numberOfLines |
| The suffix count or size or length should be used for methods, which returns a number of objects. | int TargetsCount() const;  // NOT: // int NTargets() const;  // NOT RECOMMENDED: // int NumberOfTargets() const; |
| Prefixes p, i, d, c, a etc. (called Hungarian notation) for indicating type of variable is not allowed. | mainWindow // NOT: pMainWindow height // NOT: dHeight |
| Loop variables should be called i, j, k etc.  Iterator variables should be called it for one (non-nested) loop and itWithAppropriateName, itWithSomeOtherName for iterators used in nested loops. | for (int i = 0; i < nTables); ++i) { … }  for (IteratorType it = list.begin();  it != list.end(); ++it) {  Element element = \*it;  ... } |
| The prefix is or use or has should be used for boolean variables and methods. | isSet, isVisible, isFinished, isFound, isOpen, useTranslation, hasValue  IsEmpty(), IsConfigured() |
| Enumeration constants can be prefixed by a common type name. | enum Color {  COLOR\_RED,   COLOR\_GREEN,   COLOR\_BLUE  }; |
| Enumeration constants defined as enum class, should be in mixed case starting with upper case. | enum class Color {  Red,  Green,  Blue }; |

Table 2 Source Code Naming Conventions

# Source File Commenting Conventions

Class and method header comments should follow the Doxygen conventions.

Example of the comment block:

| **Class and Method Header Comment Block** |
| --- |
| /\*\*  \* Sample function.  \*  \* This function ...  \*   \* @param param1 Parameter1 description.  \* @param param2 Parameter2 description.  \*   \* @return Return value description.  \*/ int sample\_function(int param1, int param2) {  ... } |

Figure 1 Class and Method Header Comment Block

Table 3 summarizes the conventions for documenting code in header files.

| **Documenting code in header files** | |
| --- | --- |
| **Rule** | **Example** |
| Doxygen commands should use @ instead of \. | // @param myParam Param. // NOT: \param myParam Param. |
| Comments in header files should use JavaDoc format. | /\*\*  \* Brief description.  \*   \* Details.  \*/ int var1;  /// Brief description. Details. int var2;  int var3; ///< Brief description. Details. |

Table 3 Documenting code in header files

# Statement Conventions

## Types

Table 4 summarizes the conventions for type declarations.

| **Types** | |
| --- | --- |
| **Rule** | **Example** |
| Type conversions must always be done explicitly. Never rely on implicit type conversion. | floatValue = static\_cast<float>(intValue); // NOT: floatValue = intValue; |
| Type conversion should be done by static\_cast, dynamic\_cast, const\_cast and reinterpret\_cast instead of C-like "bracket" casting. | floatValue = static\_cast<float>(intValue); // NOT: floatValue = (float) intValue; |
| The parts of a class must be sorted public, protected and private. All sections must be identified explicitly. Not applicable sections should be left out. |  |
| Use enum class instead of (plain) enum for enumerations. | enum class Color {  Red,  Green,  Blue };  // NOT: // enum Color { // COLOR\_RED,  // COLOR\_GREEN,  // COLOR\_BLUE  // }; |
| Avoid usage of plain C array. Usage of STL containers is heavily recommended.  Always choose best container (vector, list, deque, queue, …) according to planned usage. Do not use “vector for everything”. | std::list<int> myArray; // NOT: // int \* myArray = new myArray[…];  std::array<int, 25> myArray; // NOT: // int myArray[25]; |

Table 4 Types

## Variables

Table 5 summarizes the conventions for variable declarations.

| **Variables** | |
| --- | --- |
| **Rule** | **Example** |
| Variables should be initialized where they are declared or in constructor. |  |
| Variables must never have dual meaning. |  |
| Use of global variables should be minimized. |  |
| Class variables should never be declared public. |  |
| Variables should be declared in the smallest scope possible. |  |
| Local variables should be declared close to their first occurrence, not at the beginning of the function or method. |  |

Table 5 Variables

## Loops and Conditionals

Table 6 summarizes the conventions for loops and conditionals.

| **Loops and Conditionals** | |
| --- | --- |
| **Rule** | **Example** |
| Only loop control statements must be included in the for() construction. | int sum = 0; for (int i = 0; i < NUMS; ++i) {  sum += value[i]; }  // NOT: // for (int i = 0, sum = 0; i < NUMS; ++i) { // sum += value[i]; // } |
| The form for(;;) should be used for infinite loops | for (;;) {  … } |
| Any omitted part in for loop definition (initialization, condition, increment) should be commented by nothing word. | for (int i = 0; i < COUNT; /\* nothing \*/) {  … } |
| The nominal case should be put in the if-part and the exception in the else-part of an if statement | bool hasError = ReadFile(fileName); if (!hasError) {  … } else {  … } |
| The conditional should be put on a separate line. Curly brackets are recommended even in single statement case. | if (isDone) {  DoCleanup(); }  // NOT: if (isDone) DoCleanup(); |
| For-loop variable should not be used after the loop. If is necessary, use while loop instead. | int i = 0; while (i < nItems) {  …  ++i; } Function(i);  // NOT: // int i; // for (i = 0; i < nItems; ++i) {  … // } // Function(i); |

Table 6 Loops and Conditionals

## Miscellaneous

Table 7 summarizes some miscellaneous programming rules that should be followed.

| **Miscellaneous** | |
| --- | --- |
| **Rule** | **Example** |
| The use of magic numbers in the code should be avoided. These numbers should be represented by typed constant. | static const int LOOP\_MAX = 20; for (int i = 0; i < LOOP\_MAX; ++i) {  … }  // NOT: // for (int i = 0; i < 20; ++i) {  … // }  // NOT: // #define LOOP\_MAX 20 // for (int i = 0; i < LOOP\_MAX; ++i) {  … // } |
| Floating point constants should always be written with decimal point and at least one decimal. | double total = 0.0; // NOT: double total = 0;  double speed = 3.0e8; // NOT: double speed = 3e8; |
| Floating point constants should always be written with a digit before the decimal point | double total = 0.5; // NOT: double total = .5; |
| Functions must always have the return value explicitly listed. | int GetValue() {  … }  // NOT: // GetValue() { // … // } |
| goto should not be used. |  |
| const keyword should be used on all appropriate places. |  |

Table 7 Miscellaneous

## Source Code Layout

Table 8 summarizes the layout conventions that should be followed when creating a C++ source file.

| **Source Code Layout** | |
| --- | --- |
| **Rule** | **Example** |
| Basic indentation must be done by tabulators instead of spaces. |  |
| Block layout must be as illustrated.  Curly brackets should not be omitted even in single statement case.  It is not possible to leave single statement on the same line as the control keyword is. | while (!done) {   DoSomething();  done = MoreToDo(); }  // NOT: // while (!done) // { // SoSomething();  // sone = MoreToDo();  // }  // NOT: // while (!done) OnlyOneStatement(); |
| The class declarations should have the form depicted in example. | class SomeClass : public BaseClass { public:  ... protected:  ...  private:   ...  };  // NOT: // class SomeClass : public BaseClass { // public:  ... // protected:  ... // private:   ... // }; |
| Method definitions should have the form depicted in example. | void SomeMethod() {  ... } |
| The if-else class of statements should have the form depicted in example. | if (condition) {  Statements(); }  if (condition) {  Statements(); }  else if (condition) {  Statements(); } else {  Statements(); } |
| All switch statements should have a default clause. A switch statement should have the form depicted in example.  Fallthrough comments should be mentioned if case does not have break. | switch (condition) { case ABC:   Statements();   // Fallthrough  case DEF:   Statements();   break;  case XYZ: {  Statements();   break;   }  default:   Statements();   break;  } |
| Symbols \* and & for pointer and reference types should be aligned in the middle of referenced type and variable name. | MyClass \* myClass; MyClass & myClass;  // NOT: // MyClass \*myClass; // MyClass& myClass; |
| Symbol \* and & for dereferencing and addressing should be aligned to the right, closer to variable name. | return \*this; gmtime(&time);  // NOT: // return \* this; // gmtime(& time); |

Table Source Code Layout

# C++ contructs

## Enumeration as a “class with static methods”.

Classic solution:

| **Classic usage of enumerations** |
| --- |
| enum MyEnum {  MYENUM\_VALUE\_A,  MYENUM\_VALUE\_B,  MYENUM\_VALUE\_C };  std::string MyEnumToString(const MyEnum & myEnum) {  switch (myEnum) {  case MYENUM\_VALUE\_A:  return "Value A";  case MYENUM\_VALUE\_B:  return "Value B";  case MYENUM\_VALUE\_C:  return "Value C";  default:  throw std::invalid\_argument("Unknown enum value.");  } }  MyEnum StringToMyEnum(const std::string & str) {  if (str == "Value A") {  return MYENUM\_VALUE\_A;  }  else if (str == "Value B") {  return MYENUM\_VALUE\_B;  }  else if (str == "Value C") {  return MYENUM\_VALUE\_C;  }  else {  throw std::invalid\_argument("Uknown string value");  } }  // Usage: MyEnum myEnum = MYENUM\_VALUE\_B; std::string myEnumStr = MyEnumToString(myEnum); MyEnum myEnum2 = StringToMyEnum(myEnumStr); |

Figure 2 Classic usage of enumerations

Classic solution rewritten as a “class with static methods” using namespace construction:

| **Enumeration as a class with methods** |
| --- |
| namespace MyEnum  {  enum EnumType {  ValueA,  ValueB,  ValueC  };  std::string ToString(const EnumType & myEnum)  {  switch (myEnum) {  case ValueA:  return "Value A";  case ValueB:  return "Value B";  case ValueC:  return "Value C";  default:  throw std::invalid\_argument("Unknown enum value.");  }  EnumType ToEnum(const std::string & str)  {  if (str == "Value A") {  return ValueA;  }  else if (str == "Value B") {  return ValueB;  }  else if (str == "Value C") {  return ValueC;  }  else {  throw std::invalid\_argument("Uknown string value");  }  } }  // Usage: MyEnum::EnumType myEnum = MyEnum::ValueB; std::string myEnumStr = MyEnum::ToString(myEnum); MyEnum::EnumType myEnum2 = MyEnum::ToEnum(myEnumStr); |

Figure 2 Enumeration as a class

The advantage of this solution is put enumeration type and related methods together.