**Programming Guide**

**For Topology Optimisation Programmers**

**Multidisciplinary & Structural Optimisation Group**

**(**<http://people.bath.ac.uk/enshak/MSORG/>**)**

**Lead: Dr. H Alicia Kim (**[h.a.kim@bath.ac.uk](mailto:h.a.kim@bath.ac.uk)**)**

**Department of Mechanical Engineering**

**University of Bath, UK**

**Acknowledgement**

The main objective of this document is to show a programming guide for those who would like to develop topology optimisation programs using Multi-Disciplinary, multi-Scale and Structural Optimistaion Framework. The practical aim is to introduce the way how developers can write a code in an appropriate manner with regards to coding conventions, styles and formats.

The main contents this document delivers is a set of recommendation with regards to programming styles and conventions with examples. Each recommendation consists of a title, a guideline, example and its description. This is seen as below:

1. This is a title of this guideline

|  |
| --- |
| **This row is a guideline** |
| This row shows example associated with the guideline above |
| This row addresses a description, justification and/or source of the guideline above |

The framework and its programming guide are now distributed/managed by Multidisciplinary and Structural Optimisation Group under the lead of Dr. H. Alicia Kim.

This programming guide is extended/revised from the “C++ Programming Style Guidelines”, published by Geotechnical Software Services. The original, whole guidelines are accessible via this website – <http://geosoft.no/development/cppstyle.html>.

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Ver** | **Author** | **Description** | **Date** |
| v0.0 | Khalid | Draft – simply downloaded from <http://geosoft.no/development/cppstyle.html#Naming%20Conventions> | 25 Nov 2014 |
| v0.0 | JeeHang | Add title, acknowledgement, revision history and contents | 08 Dec 2014 |
| v0.1 | JeeHang | Complete naming conventions | 09 Dec 2014 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Table of Contents**

[**1. General Recommendations** 5](#_Toc405905697)

[1.1. Violation of Recommendation – For Readability 5](#_Toc405905698)

[1.2. Violation of Recommendation – Towards Personal Preference 5](#_Toc405905699)

[**2.** **Naming Conventions** 5](#_Toc405905700)

[**3.** **Statements** 14](#_Toc405905701)

[**4.** **Layout and Comments** 14](#_Toc405905702)

[**5.** **Files** 14](#_Toc405905703)

[**6.** **References** 14](#_Toc405905704)

# General Recommendations

# Violation of Recommendation – For Readability

|  |
| --- |
| **Any violation to the guide is allowed if it enhances readability.** |
|  |
| The main goal of the recommendation is to improve readability and thereby the understanding and the maintainability and general quality of the code. It is impossible to cover all the specific cases in a general guide and the programmer should be flexible. |

# Violation of Recommendation – Towards Personal Preference

|  |
| --- |
| **The rules can be violated if there are strong personal objections against them.** |
|  |
| The attempt is to make a guideline, not to force a particular coding style onto individuals. Experienced programmers normally want to adopt a style like this anyway, but having one, and at least requiring everyone to get familiar with it, usually makes people start thinking about programming style and evaluate their own habits in this area.  On the other hand, new and inexperienced programmers normally use a style guide as a convenience of getting into the programming jargon more easily. |

1. **Naming Conventions**
   1. Names – Data Types

|  |
| --- |
| **Names representing types must be in mixed case starting with upper case.** |
| CName, Line, SavingsAccount |
| Common practice in the C++ development community. |

* 1. Names – Class Definition

|  |
| --- |
| **Names representing class must be in mixed case starting with upper case.** |
| class CClassName |
| Common practice in the C++ development community. |

* 1. Names – Class Initials

|  |
| --- |
| **Names representing class must have a following acronym (HAK) in mixed case starting with upper case.** |
| class CHakSolver, class CHakSensitivity, class CHakInput |
| Recommendation for the group. |

* 1. Names – Variables

|  |
| --- |
| **Variable names must be in mixed case starting with lower case.** |
| name, line, savingsAccount |
| Common practice in the C++ development community. Makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration Line line; |

* 1. Names – Types of Variables

|  |
| --- |
| **Variable names must start the initial of its data types with lower case.** |
| cName, bRes, fPoint, iNumber, cChar, sString |
| Complying Hungarian Notation which is a common practice in the C++ development community. Makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration Line line; |

* 1. Names – Variables

|  |
| --- |
| **Variable names must be in mixed case starting with lower case.** |
| name, line, savingsAccount |
| Common practice in the C++ development community. Makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration Line line; |

* 1. Names – Constants

|  |
| --- |
| **Named constants (including enumeration values) must be all uppercase using underscore to separate words.** |
| MAX\_ITERATIONS, COLOR\_RED, PI |
| Common practice in the C++ development community. In general, the use of such constants should be minimized. In many cases implementing the value as a method is a better choice: in modern C++, an inline function (like as below) returning a constant is more appropriate form instead of pre-definition using *#define*.  // NOT: #define MAX\_ITERATIONS = 25  inline int getMaxIterations() { return 25; }  This form is both easier to read, and it ensures a unified interface towards class values.  Enumeration cases simply follow the recommendation like an example below.  enum Color  {  COLOR\_RED,  COLOR\_GREEN,  COLOR\_BLUE  }; |

* 1. Names – Functions/Methods

|  |
| --- |
| **Names representing methods or functions must be verbs and written in mixed case starting with upper case.** |
| GetName(), ComputeTotalWidth() |
| Common practice in the C++ development community. |

* 1. Names – Namespaces

|  |
| --- |
| **Names representing namespaces should be all lowercase.** |
| model::analyzer, io::iomanager, common::math::geometry, topopt::sensitivity |
| Common practice in the C++ development community. |

* 1. Names – Templates

|  |
| --- |
| **Names representing template types should be a single uppercase letter.** |
| template<class T> ...  template<class C, class D> ... |
| Common practice in the C++ development community. This makes template names stand out relative to all other names used. |

* 1. Names – Abbreviations and Acronyms

|  |
| --- |
| **Abbreviations and acronyms must not be uppercase when used as name [2.4].** |
| exportHtmlSource(); *// NOT: exportHTMLSource();*  openDvdPlayer(); *// NOT: openDVDPlayer();* |
| Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type whould have to be named dVD, hTML etc. which obviously is not very readable. Another problem is illustrated in the examples above; When the name is connected to another, the readbility is seriously reduced; the word following the abbreviation does not stand out as it should. |

* 1. Names – Global Variables

|  |
| --- |
| **Global variables should always be referred to using the :: operator.** |
| ::mainWindow.open(), ::applicationContext.getName() |
| In general, the use of global variables should be avoided. Consider using singleton objects instead, which can provide both reference counting and interfaces in order to refer the global variables. |

* 1. Names – Member Variables in Class

|  |
| --- |
| **class member variables should have m\_ prefix.** |
| class CHakSomeClass {  private:  int m\_nLength;  } |
| Apart from its name and its type, the *scope* of a variable is its most important feature. Indicating class scope by using m\_ prefix makes it easy to distinguish class variables from local scratch variables. This is important because class variables are considered to have higher significance than method variables, and should be treated with special care by the programmer.  It should be noted that scope identification in variables has been a controversial issue for quite some time. It seems, though, that this practice now is gaining acceptance and that it is becoming more and more common as a convention in the professional development community. |

* 1. Names – Private Member Method in Class

|  |
| --- |
| **class private method should have ‘\_’ (underscore) prefix.** |
| class CHakSomeClass {  public:  bool Verify();  private:  bool \_IsTrue();  } |
| Apart from its name and its type, the *scope* of a variable is its most important feature. Indicating class scope by using ‘\_’ (underscore) prefix makes it easy to distinguish private methods from local Interfaces publicly opened.  It should be noted that scope identification in variables has been a controversial issue for quite some time. It seems, though, that this practice now is gaining acceptance and that it is becoming more and more common as a convention in the professional development community. |

* 1. Names – Generic Variables

|  |
| --- |
| **Generic variables should have the same name as their type.** |
| void setTopic(Topic\* topic) *// NOT: void setTopic(Topic\* value)*  *// NOT: void setTopic(Topic\* aTopic)*  *// NOT: void setTopic(Topic\* t)*  void connect(Database\* database) *// NOT: void connect(Database\* db)*  *// NOT: void connect (Database\* oracleDB)* |
| Reduce complexity by reducing the number of terms and names used. Also makes it easy to deduce the type given a variable name only.  If for some reason this convention doesn't seem to *fit* it is a strong indication that the type name is badly chosen.  Non-generic variables have a *role*. These variables can often be named by combining role and type:  Point startingPoint, centerPoint;  Name loginName; |

* 1. Names – Generic Variables

|  |
| --- |
| **All names should be written in English.** |
| fileName; *// NOT: filNavn* |
| (British) English is the preferred language for international development. |

* 1. Names – Scratch Variables

|  |
| --- |
| **Variables with a large scope should have long name, variables with a small scope may have short name.** |
|  |
| Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside of a few lines of code. However, programmers can see the meaning of the variables although shorter names are allowed.  In some case, common scratch variables for integers such as *i*, *j*, *k*, *m*, *n* and for characters such as *c* and *d* would be considered but not strongly recommended. |

* 1. Names – Object

|  |
| --- |
| **The name of the object is implicit, and should be avoided in a method name.** |
| cLine.GetLength(); *// NOT: cLine.GetLineLength();* |
| The latter seems natural in the class declaration, but proves superfluous in use, as shown in the example. |

* 1. Specific Names – Get/Set Properties

|  |
| --- |
| **The terms *get/set* must be used where an attribute is accessed directly.** |
| employee.GetName();  employee.SetName(name);  matrix.GetElement(2, 4);  matrix.SetElement(2, 4, value); |
| Common practice in the C++ development community. In Java this convention has become more or less standard. |

* 1. Specific Names – Use of ‘Compute’

|  |
| --- |
| **The term *compute* can be used in methods where something is computed.** |
| valueSet->ComputeAverage();  matrix->ComputeInverse() |
| Give the reader the immediate clue that this is a potentially time-consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability. |

* 1. Specific Names – Use of ‘Find’

|  |
| --- |
| **The term *find* can be used in methods where something is looked up.** |
| vertex.FindNearestVertex();  matrix.FindMinElement(); |
| Give the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability. |

* 1. Specific Names – Use of ‘Initialise’

|  |
| --- |
| **The term *initialize* can be used where an object or a concept is established.** |
| printer.initialiseFontSet(); |
| The British *initialise*should be preferred over the american *initialize*. Abbreviation *init*should be avoided. |

* 1. Specific Names – GUI Components

|  |
| --- |
| **Variables representing GUI components should be suffixed by the component type name.** |
| mainWindow, propertiesDialog, widthScale, loginText,  leftScrollbar, mainForm, fileMenu, minLabel, exitButton, yesToggle etc. |
| Enhances readability since the name gives the user an immediate clue of the type of the variable and thereby the objects resources. |

* 1. Specific Names – Collection of Objects

|  |
| --- |
| **Plural form should be used on names representing a collection of objects.** |
| vector<Point> vPoints;  int iValues[]; |
| Enhances readability since the name gives the user an immediate clue of the type of the variable and the operations that can be performed on its elements. |

* 1. Specific Names – Number of Objects

|  |
| --- |
| **The prefix *n* should be used for variables representing a number of objects.** |
| nPoints, nLines |
| The notation is taken from mathematics where it is an established convention for indicating a number of objects. |

* 1. Specific Names – An Entity Number

|  |
| --- |
| **The suffix *No* should be used for variables representing an entity number.** |
| iTableNo, iEmployeeNo |
| The notation is taken from mathematics where it is an established convention for indicating an entity number. |

* 1. Specific Names – Iterator Variables

|  |
| --- |
| **Iterator variables should be called *i*, *j*, *k, it, idx, iter* etc.** |
| for (int i = 0; i < nTables); i++) {  // do something  }  for (int idx = 0; idx < nTables); idx++) {  // do something  }  for (vector<MyClass>::iterator it = list.begin(); it != list.end(); it++) {  Element element = \*it;  // do something  } |
| The notation is taken from mathematics where it is an established convention for indicating iterators. Variables named *j*, *k* etc. should be used for nested loops only. Iterator variables should be called *it* or *iter* in order to distinguish index for loop. |

* 1. Specific Names – Use of ‘is’ prefix

|  |
| --- |
| **The prefix *is* should be used for boolean variables and methods.** |
| isSet, isVisible, isFinished, isFound, isOpen |
| Common practice in the C++ development community and partially enforced in Java.  Using the *is* prefix solves a common problem of choosing bad boolean names like status or flag. isStatus or isFlag simply doesn't fit, and the programmer is forced to choose more meaningful names.  There are a few alternatives to the *is* prefix that fit better in some situations. These are the *has*, *can* and *should* prefixes:  bool hasLicense();  bool canEvaluate();  bool shouldSort(); |

* 1. Specific Names – Complement Names by Symmetry

|  |
| --- |
| **Complement names must be used for complement operations [1].** |
| Get/Set, Add/Remove, Create/Destroy, Start/Stop, Insert/Delete,  Increment/Decrement, Old/New, Begin/End, First/Last, Up/Down, Min/Max, etc. |
| Reduce complexity by symmetry. |

* 1. Specific Names – Full Word in Names

|  |
| --- |
| **Abbreviations in names should be avoided.** |
| computeAverage(); *// NOT: compAvg();* |
| There are two types of words to consider. First are the common words listed in a language dictionary. These must never be abbreviated. Never write:  cmd    instead of  command cp     instead of   copy pt     instead of   point comp  instead of   compute init   instead of   initialize etc.  Then there are domain specific phrases that are more naturally known through their abbreviations/acronym. These phrases should be kept abbreviated. Never write:  HypertextMarkupLanguage  instead of   html CentralProcessingUnit     instead of   cpu PriceEarningRatio         instead of   pe etc. |

Specific Names – Pointers

|  |
| --- |
| **Naming pointers specifically should be avoided.** |
| Line\* line; *// NOT: Line\* pLine;*  *// NOT: LIne\* linePtr;* |
| Many variables in a C/C++ environment are pointers, so a convention like this is almost impossible to follow. Also objects in C++ are often oblique types where the specific implementation should be ignored by the programmer. Only when the actual type of an object is of special significance, the name should emphasize the type. |

* 1. Specific Names – Negated Variable Names

|  |
| --- |
| **Negated boolean variable names must be avoided.** |
| bool isError; *// NOT: isNoError*  bool isFound; *// NOT: isNotFound* |
| The problem arises when such a name is used in conjunction with the logical negation operator as this results in a double negative. It is not immediately apparent what !isNotFound means. |

* 1. Specific Names – Prefix of Enumeration Constants

|  |
| --- |
| **Enumeration constants can be prefixed by a common type name.** |
| enum Color {  COLOR\_RED,  COLOR\_GREEN,  COLOR\_BLUE  }; |
| This gives additional information of where the declaration can be found, which constants belongs together, and what concept the constants represent.  An alternative approach is to always refer to the constants through their common type: Color::RED, Airline::AIR\_FRANCE etc.  Note also that the enum name typically should be *singular* as in enum Color {...}. A plural name like enum Colors {...} may look fine when declaring the type, but it will look silly in use. |

* 1. Specific Names – ‘*Exception*’ Class

|  |
| --- |
| **Exception classes should be suffixed with *Exception*.** |
| class CAccessException  {  :  } |
| Exception classes are really not part of the main design of the program, and naming them like this makes them stand out relative to the other classes. |

* 1. Specific Names – Orders in Function Declaration

|  |
| --- |
| **Functions (methods returning something) should be named after what they return and procedures (*void* methods) after what they do.** |
|  |
| Increase readability. Makes it clear what the unit should do and especially all the things it is not supposed to do. This again makes it easier to keep the code clean of side effects. |

1. **Statements**
2. **Layout and Comments**
3. **Files**
4. **References**