Programming Style Guidelines
ArangoDB Edition

Dr. Frank Celler

Version 1.2.0

Contents

1 Introduction **5**
- 1.1 Layout of the Recommendations . . . 5
  - 1.1.1 Guideline short description . . . 5
- 1.2 Recommendation Importance . . . 5

2 General Recommendations **6**
- 2.1 General Naming Conventions . . . 6
  - 2.1.1 Names must be chosen by forming an english description . . . 6
  - 2.1.2 Names representing types must be in upper camel case . . . 6
  - 2.1.3 Variable names must be in lower camel case . . . 6
  - 2.1.4 Named constants must be all uppercase . . . 7
  - 2.1.5 Names representing methods or functions must be in lower camel case . . . 7
  - 2.1.6 Names representing namespaces must be all lowercase . . . 7
  - 2.1.7 Abbreviations and acronyms must not be uppercase when used as name . . . 7
  - 2.1.8 Generic variables must have the same name as their type . . . 8
  - 2.1.9 Non-generic variables should have a role . . . 8
  - 2.1.10 All names should be written in English . . . 8
  - 2.1.11 The length of a name should corresponde to the scope . . . 8
  - 2.1.12 The maximal length of a name must be 40 . . . 8
  - 2.1.13 The name of the object should be avoided in a method name . . . 8
- 2.2 Specific Naming Conventions . . . 9
  - 2.2.1 Getters and setters . . . 9
  - 2.2.2 Should use `compute` for methods which compute and store . . . 9
  - 2.2.3 Should use `find` and `lookup` for look ups . . . 9
  - 2.2.4 Should use `initialize` for initialisation . . . 10
  - 2.2.5 Should use GUI component type name as suffix . . . 10
  - 2.2.6 Can use `List` suffix for lists . . . 10
  - 2.2.7 Should use prefix `n` or `number` . . . 10
  - 2.2.8 Should use suffix `Id` for identifier . . . 10
  - 2.2.9 Should use mathematical iterators names . . . 11
  - 2.2.10 The prefix `is` should be used for boolean variables and methods . . . 11
  - 2.2.11 Complement names must be used for complement operations . . . 11
  - 2.2.12 Abbreviations in names must be avoided . . . 12
  - 2.2.13 Naming pointers specifically should be avoided . . . 12
  - 2.2.14 Negated boolean variable names must be avoided . . . 12
  - 2.2.15 Enumeration constants can be prefixed by a common type name . . . 12
- 2.3 C++ Naming Conventions . . . 13
  - 2.3.1 Within a given module, class names must be unique . . . 13
  - 2.3.2 Exception classes should be suffixed with `Exception` or `Error` . . . 13
  - 2.3.3 Enumerations should be lowercase follwed by `_e` . . . 13
  - 2.3.4 Template names should are short and must be uppercase . . . 13
  - 2.3.5 Global variables should always be referred to using the `::` operator . . . 13

3 Files **14**
3.1 Source Files . . . 14
3.1.1 Must use ".h" and ".cpp" extensions . . . 14
3.1.2 Must use two files per class for public classes . . . 14
3.1.3 All computations must reside in source files . . . 14
3.1.4 File content should be kept within 80 columns . . . 15
3.2 Include Files and Include Statements . . . 15
3.2.1 Must use include guards . . . 15
3.2.2 Include statements should be sorted and grouped . . . 15
3.2.3 Include statements must be located at the top of a file only . . . 16
3.3 File Structure . . . 16
3.3.1 Header file structure . . . 16
3.3.2 Source file structure . . . 17

4 C++ Statements **18**
4.1 Namespaces . . . 18
4.1.1 Namespaces must not be included globally in the header . . . 18
4.2 Types . . . 18
4.2.1 Types that are local to one file only can be declared inside that file . . . 18
4.2.2 Type conversions must always be done explicitly . . . 18
4.3 Variables . . . 18
4.3.1 Variables should be initialized where they are declared . . . 18
4.3.2 Variables must never have dual meaning . . . 19
4.3.3 Use of global variables should be minimized . . . 19
4.3.4 Non-constant class variables must never be declared public . . . 19
4.3.5 Related variables of the same type can be declared in a common statement 19
4.3.6 `const` must go after the type . . . 19
4.3.7 Constant must be on the left hand side of a comparison . . . 19
4.3.8 You must not use implicit 0 tests . . . 20
4.3.9 Variables should be declared in the smallest scope possible. . . . 20
4.4 Loops . . . 20
4.4.1 Only loop control statements must be included in the `for` construction . . 20
4.4.2 Should initialise loop variables before the loop . . . 20
4.4.3 `do-while` loops can be avoided . . . 21
4.4.4 The use of `break` and `continue` in loops should be minimized . . . 21
4.4.5 The form `while (true)` should be used for infinite loops . . . 21
4.5 Conditionals . . . 21
4.5.1 Complex conditional expressions must be avoided . . . 21
4.5.2 Should put the exceptional case in the `else`-part . . . 22
4.5.3 The conditional should be put on a separate line . . . 22
4.5.4 Executable statements in conditionals must be avoided . . . 22
4.6 Miscellaneous . . . 22
4.6.1 `goto` should not be used . . . 22
4.6.2 Functions must always have the return value explicitly listed . . . 23
4.6.3 `0` should be used instead of `NULL` in C++ . . . 23

5 Miscellaneous **24**
5.1 Miscellaneous . . . 24
5.1.1 The use of magic numbers in the code must be avoided . . . 24
5.1.2 The use of magic strings in the code should be avoided . . . 24
5.1.3 Use a decimal point for floating point constants . . . 24

5.1.4 Floating point constants should always be written with a digit before the decimal point . . . 24
5.1.5 Floating point comparison should be avoid . . . 24

6 Layout and Comments **25**
6.1 Layout . . . 25
6.1.1 Tabs must be eight characters wide . . . 25
6.1.2 Basic indentation should be 2 . . . 25
6.1.3 Fallthrough comment must be used . . . 28
6.1.4 Empty blocks or on-line blocks must use brackets . . . 28
6.2 White Space . . . 28
6.2.1 Conventional operators should be surrounded by a space character . . . 28
6.2.2 The method name should be followed by a space . . . 29
6.2.3 Logical units within a block should be separated by one blank line . . . 29
6.2.4 Methods should be separated by blank lines . . . 29
6.2.5 Variables in declarations can be left aligned . . . 30
6.3 Comments . . . 30
6.3.1 Tricky code should not be commented but rewritten . . . 30
6.3.2 All comments should be written in english . . . 30
6.3.3 Use `//` for all comments, including multi-line comments . . . 30
6.3.4 Must use development comments only during development . . . 31
6.3.5 You must use doxygen conventions for C++ . . . 31
6.3.6 Sectionise large files . . . 31

7 References **32**

1 Introduction

This document lists coding recommendations for the projects ArangoDB. They are based on C++ coding recommendations common in the C++ development community, on established standards collected from a number of sources, individual experience, local requirements/needs, as well as suggestions given in 1 - 5 of chapter 7.

There are several reasons for introducing a new guideline rather than just referring to the ones above. The main reason is that these guides are far too general in their scope and that more specific rules (especially naming rules) need to be established. Also, the present guide has an annotated form that makes it far easier to use during project code reviews than most other existing guidelines. In addition, programming recommendations generally tend to mix style issues with language technical issues in a somewhat confusing manner. The present document does not contain any C++ technical recommendations at all, but focuses mainly on programming style.

While a given development environment (IDE) can improve the readability of code by access visibility, color coding, automatic formatting and so on, the programmer should never rely on such features. Source code should always be considered larger than the IDE it is developed within and should be written in a way that maximize its readability independent of any IDE.

1.1 Layout of the Recommendations

The recommendations are grouped by topic and each recommendation is numbered to make it easier to refer to during reviews. Layout of the recommendations is as follows:

1.1.1 Guideline short description

Guideline longer description

```
Example if applicable
```

Motivation, background and additional information.

The motivation section is important. Coding standards and guidelines tend to start "religious wars", and it is important to state the background for the recommendation.

1.2 Recommendation Importance

In the guideline sections the terms *must*, *should* and *can* have special meaning. A *must* requirement must be followed, a *should* is a strong recommendation, and a *can* is a general guideline.

2 General Recommendations

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.

However, as readability is subject to personal taste, any violation should be discussed with the team. If a violation makes sense within a given context, a corresponding recomemendation should be added to this document.

2.1 General Naming Conventions

2.1.1 Names must be chosen by forming an english description

Names must be chosen by forming an english description with "of", "in", "to", "from" left out.

```
TypeLanguage  // NOT TypeOfLanguage or LanguageType
```

Words like "of" are automatically added when reading.

2.1.2 Names representing types must be in upper camel case

Names representing types (Classes, Interfaces) must be in upper camel case starting with upper case.

```
Line
SavingsAccount
```

Common practice in the C++ development community.

Standard names are allowed to violated the above rule. Names used for instance in the C++ STL do not always follow the above rule. In this case the established names should be used. Examples are `size`, `length`, `at`, `c_str`.

2.1.3 Variable names must be in lower camel case

Variable names must be in lower camel case starting with lower case. Private and protected member variables must start with a "_".

```
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.

2.1.4 Named constants must be all uppercase

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:

```
int maxIterations () const {
  return 25;
}
```

This form is both easier to read, and it ensures a unified interface towards class values.

2.1.5 Names representing methods or functions must be in lower camel case

Names representing methods or functions, which *do some work*, must be verbs and written in lower camel case starting with lower case. Names representing methods or functions, which *return something*, must be substantives and written in lower camel case starting with lower case.

```
w = totalWidth();
validateInput();
```

Common practice in the C++ development community. This is identical to variable names, but functions in C++ are already distinguishable from variables by their specific form.

Functions (methods returning something) should be named after what they return and procedures (void methods) after what they do. This increases 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.

One exception to these rules are getter and setter - see below.

2.1.6 Names representing namespaces must be all lowercase

Names representing namespaces must be all lowercase using underscore to separate words.

```
analyzer
io_manager
```

Common practice in the C++ development community.

2.1.7 Abbreviations and acronyms must not be uppercase when used as name

```
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.

2.1.8 Generic variables must 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 *x)

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.

2.1.9 Non-generic variables should have a role

Non-generic variables have a role. These variables can often be named by combining role and type.

```
Point startingPoint, centerPoint;
Name loginName;
```

2.1.10 All names should be written in English

```
FileName;     // NOT: filNavn
```

English is the prefered language for international development.

2.1.11 The length of a name should corresponde to the scope

Variables with a large scope should have long names, variables with a small scope can have short names.

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 a few lines of code. Common scratch variables for integers are i, j, k, m, n and for characters c and d.

2.1.12 The maximal length of a name must be 40

Very long names make the program harder to read because more line breaks are required. If the very long name is required to describe a method or variable it could be an indication that this method or variable is to complex and that the code should be refactored.

2.1.13 The name of the object should be avoided in a method name

The name of the object is implicit, and should be avoided in a method name.

```
line.getLength();  // NOT: line.getLineLength();
```

The latter seems natural in the class declaration, but proves superfluous in use, as shown in the example.

2.2 Specific Naming Conventions

2.2.1 Getters and setters

The terms `get`/`set` must be used where an attribute is accessed directly

```
employee.getName();
matrix.getElement(2, 4);
employee.setName(name);
matrix.setElement(2, 4, value);
```

In Java this convention has become more or less standard.

Note that this is only used when accessing a member variable directly. If any computation is involved the above rule does not apply and section 2.1 should be followed. A getter or setter must only consists of a few line of code.

The terms `is`/`set` must be used where a boolean attribute is accessed directly.

```
matrix.isDense();
```

In Java this convention has become more or less standard. There are exceptions to this rule in C++. The STL for example defines `empty` *instead* `isEmpty`*.*

If definition a setter `setSomething` use `newSomething` as variable name for the new value.

2.2.2 Should use `compute` for methods which compute and store

The term `compute` should be used in methods where something is computed and stored within the object.

```
valueSet->computeAverage();
matrix->computeInverse();
```

Give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

2.2.3 Should use `find` and `lookup` for look ups

The terms `find` and `lookup` should be used in methods where something is looked up.

```
vertex.findNearestVertex();
matrix.findMinElement();
vertex.lookupVertex(vertexId);
```

Give the reader the immediate clue that this is a simple look up method with a minimum of computations involved. `lookup` *should be used when no computation or minimal computation is done and no error is raised when the element is not found.* `find` *should be used when computations are required; it is allowed to create missing elements or to raise an error. Consistent use of the terms enhances readability.*

2.2.4 Should use `initialize` for initialisation

The term `initialize` should be used where an object or a concept is established.

```
printer.initializeFontSet();
```

The american initialize should be preferred over the british initialise. The abbreviation `init` *should be avoided.*

2.2.5 Should use GUI component type name as suffix

Variables representing GUI components should be suffixed by the component type name.

```
mainWindow, propertiesDialog, widthScale, loginText, leftScrollbar,
mainForm, fileMenu, minLabel, exitButton, yesToggle
```

Enhances readability since the name gives the user an immediate clue of the type of the variable and thereby the objects resources.

2.2.6 Can use `List` suffix for lists

The suffix `List` can be used on names representing a list of objects.

```
vertex (one vertex)
vertexList (a list of vertices)
```

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 the object. Simply using the plural form of the base class name for a list (matrixElement (one matrix element), matrixElements (list of matrix elements)) shoul be avoided since the two only differ in a single character and are thereby difficult to distinguish.

A list in this context is the compound data type that can be traversed backwards, forwards, etc. (typically an STL vector). A plain array is simpler. The suffix `Array` can be used to denote an array of objects.

2.2.7 Should use prefix `n` or `number`

The prefix `n` or `number` should be used for variables representing a number of objects.

```
nPoints, numberLines
```

The notation is taken from mathematics where it is an established convention for indicating a number of objects.

2.2.8 Should use suffix `Id` for identifier

The suffix `Id` should be used for variables representing an entity number.

```
tableId, employeeId
```

The notation is taken from mathematics where it is an established convention for indicating an entity number.

2.2.9 Should use mathematical iterators names

Iterator variables should be called `i`, `j`, `k` etc or `iter`.

```
for (int i = 0; i < nTables); i++) {
  ...
}

vector<MyClass>::iterator iter;

for (iter = list.begin();  iter != list.end();  ++iter) {
  Element element = *iter;
  ...
}
```

The notation is taken from mathematics where it is an established convention for indicating iterators. `iter` *should be used for STL iterators.*

An elegant alternative is to prefix such variables with an i or iter: iTable, iterEmployee. This effectively makes them named iterators.

2.2.10 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 fits better in some situations. These are the `has`, `can` and `should` prefixes:

```
bool hasLicense();
bool canEvaluate();
bool shouldSort();
```

2.2.11 Complement names must be used for complement operations

```
get/set
add/remove
create/destroy
start/stop
insert/delete
increment/decrement
old/new
begin/end
first/last
up/down
min/max
next/previous
open/close
show/hide
suspend/resume
```

Reduce complexity by symmetry.

2.2.12 Abbreviations in names must 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

Then there are domain specific phrases that are more naturally known through their abbreviations/acronym. These phrases must be kept abbreviated. Never write:

- HypertextMarkupLanguage instead of html
- CentralProcessingUnit instead of cpu
- PriceEarningRatio instead of pe

2.2.13 Naming pointers specifically should be avoided

```
Line * line;  // NOT: Line * pLine; or Line * linePtr; etc.
```

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 empahsize the type.

2.2.14 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.

2.2.15 Enumeration constants can be prefixed by a common type name

```
enum color_e {
  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.*

2.3 C++ Naming Conventions

2.3.1 Within a given module, class names must be unique

If class names are unique it is much easier to understand the code because mix-ups are avoided.

2.3.2 Exception classes should be suffixed with `Exception` or `Error`

```
class AccessException {
  ...
}
```

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.

2.3.3 Enumerations should be lowercase follwed by `_e`

```
enum color_e {
  ..
};
```

2.3.4 Template names should are short and must be uppercase

Names representing template types must be uppercase.

Names representing generic template types should be a single uppercase letter. Two uppercase letters at most. Names representing specific template types should be named after that type.

```
template<typename T> ...
template<typename C, typename D> ...
template<typename C, class VERTEX> ...
```

Common practice in the C++ development community. This makes template names stand out relative to all other names used.

2.3.5 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.

3 Files

3.1 Source Files

3.1.1 Must use ".h" and ".cpp" extensions

C++ header files must have the extension ".h". Source files must the extension ".cpp". The extensions ".C", ".cc" or ".c++" are not allowed.

```
MyClass.cpp, MyClass.h
```

These are all accepted C++ standards for file extension.

3.1.2 Must use two files per class for public classes

A class must be declared in a header file and implemented in a source file where the name of the files match the name of the class. Classes which are local to computation and have a file context, can be defined in the source file in an anonymous namespace.

The files must have the same name as the class including case. A class `MyHthml` must be declared in "MyHtml.h"" and implemented in "MyHtml.cpp".

```
MyClass.h, MyClass.cpp
```

Makes it easy to find the associated files of a given class. This convention is enforced in Java and has become very successful as such.

3.1.3 All computations must reside in source files

```
class MyClass {
  public:
    int getValue () {return value;} // YES
    int computeSomething () {...computation...} // NO!

  private:
    int value;
}
```

The header files must declare an interface of a class, the source file must implement it. When looking for an implementation, the programmer should always know that it is found in the source file. The obvious exception to this rule are of course inline functions and templates that must be defined in the header file. Trivial getter and setter are another exception.

3.1.4 File content should be kept within 80 columns

80 columns is a common dimension for editors, terminal emulators, printers and debuggers, and files that are shared between several people should keep within these constraints. It improves readability when unintentional line breaks are avoided when passing a file between programmers. Special characters like TAB and page break must be avoided.

These characters are bound to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment. The incompleteness of split lines must be made obvious.

```
totalSum = a + b + c
         + d + e;

function (param1, param2,
         param3);

setText ("Long line split"
         "into two parts.");

for (tableNo = 0; tableNo < nTables;
     tableNo += tableStep)
```

Split lines occurs when a statement exceed the 80 column limit given above. It is difficult to give rigid rules for how lines should be split, but the examples above should give a general hint. In general:

- Break after a comma.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line.

3.2 Include Files and Include Statements

3.2.1 Must use include guards

Header files must include a construction that prevents multiple inclusion. The convention is an all uppercase construction of the module/directory name, the file name and the `h` suffix.

Obeying 3.1 and 2.3 guarantees that the input guards is unique.

```
#ifndef MODULE_FILENAME_H
#define MODULE_FILENAME_H
...
#endif
```

The construction is to avoid compilation errors. The name convention is common practice. The construction should appear in the top of the file (before the file header) so file parsing is aborted immediately and compilation time is reduced.

3.2.2 Include statements should be sorted and grouped

Sorted by their hierarchical position in the system with low level files included first. Leave an empty line between groups of include statements.

```
#include <fstream>
#include <iomanip>

#include <Xm/Xm.h>
#include <Xm/ToggleB.h>

#include "ui/PropertiesDialog.h"
#include "ui/MainWindow.h"
```

In addition to show the reader the individual include files, it also give an immediate clue about the modules that are involved. Include file paths must never be absolute. Compiler directives should instead be used to indicate root directories for includes.

3.2.3 Include statements must be located at the top of a file only

Common practice. Avoid unwanted compilation side effects by "hidden" include statements deep into a source file.

3.3 File Structure

3.3.1 Header file structure

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. The ordering is "most public first" so people who only wish to use the class can stop reading when they reach the protected/private sections.

Use the following ordering within a header file.

1. header including copyright, author, and date
2. define-guard, see 3.2
3. include "Common.h" or header-file of base classes
4. include system C headers
5. include system C++ headers
6. include project C++ headers, use forward declarations where possible
7. forward declararions for C++ classes and structs
8. class declararion
    a) local classes and enumerations
    b) static constant variables
    c) static functions
    d) static variables
    e) constructors and destructors
    f) public methods
    g) public variables
    h) protected methods
    i) protected variables

j) private methods

k) private variables

3.3.2 Source file structure

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. The ordering is "most public first" so people who only wish to use the class can stop reading when they reach the protected/private sections.

Use the following ordering within a source file.

1. header including copyright, author, and date
2. corresponding header file
3. include system C headers
4. include system C++ headers
5. include project C++ headers
6. class implementation
    a) auxillary function hidden in an anonymous namespace
    b) static constant variables
    c) static functions
    d) constructors and destructors
    e) public methods
    f) protected methods
    g) private methods

4 C++ Statements

4.1 Namespaces

4.1.1 Namespaces must not be included globally in the header

A namespace must be included by `using` either within the source file or within another namespace. It must not be included globally in a header file.

Avoids conflicts with other libraries.

4.2 Types

4.2.1 Types that are local to one file only can be declared inside that file

Classes which are local to computation and have a file context, can be defined in the source file in an anonymous namespace.

Enforces information hiding.

4.2.2 Type conversions must always be done explicitly

Type conversions must always be done explicitly. Never rely on implicit type conversion.

```
floatValue = static_cast<float> (intValue);   // YES!
floatValue = intValue;                        // NO!
```

By this, the programmer indicates that he is aware of the different types involved and that the mix is intentional.

4.3 Variables

4.3.1 Variables should be initialized where they are declared

This ensures that variables are valid at any time. Sometimes it is impossible to initialize a variable to a valid value where it is declared:

```
int x, y, z;
getCenter (&x, &y, &z);
```

In these cases it should be left uninitialized rather than initialized to some phony value.

4.3.2 Variables must never have dual meaning

Enhance readability by ensuring all concepts are represented uniquely. Reduce chance of error by side effects.

4.3.3 Use of global variables should be minimized

In C++ there is no reason global variables need to be used at all. The same is true for global functions or file scope (static) variables.

4.3.4 Non-constant class variables must never be declared public

The concept of C++ information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C struct). In this case it is appropriate to make the class' instance variables public.

Note that structs are kept in C++ for compatibility with C only, and avoiding them increases the readability of the code by reducing the number of constructs used. You should use a class instead.

4.3.5 Related variables of the same type can be declared in a common statement

Related variables of the same type can be declared in a common statement. Unrelated variables should not be declared in the same statement.

```
float x, y, z;
float revenueJanuary, revenueFebruary, revenueMarch;
```

The common requirement of having declarations on separate lines is not useful in the situations like the ones above. It enhances readability to group variables like this. Howevery, in the case the type must not be a pointer.

Pointers and classes however must be declared on separate lines.

C++ pointers and references should have their reference symbol next to the type name rather than to the variable name.

```
float* x;    // NOT: float *x;
int&   y;    // NOT: int &y;
```

It is debatable whether a pointer is a variable of a pointer type (float* x) or a pointer to a given type (float *x). It is impossible to declare more than one pointer in a given statement using the first approach. I.e. float* x, y, z; is equivalent with float *x; float y; float z; The same goes for references.

4.3.6 `const` must go after the type

```
string const& ref = ...;      // NOT: const string& ref
```

The type declarations are read from right to left. The type should be the last.

4.3.7 Constant must be on the left hand side of a comparison

If a constant is on the left hand side then forgetting a = sign results in an error message.

```
if (a = 0)        // compiles, but always false
if (0 = a)        // compiler error
if (0 == a)       // correct
```

4.3.8 You must not use implicit 0 tests

Implicit test for 0 should not be used other than for boolean variables and pointers.

```
if (0 != nLines)        // NOT: if (nLines)
if (0.0 != value)       // NOT: if (value)
```

It is not necessarily defined by the compiler that ints and floats 0 are implemented as binary 0. Also, by using explicit test the statement give immediate clue of the type being tested.

It is common also to suggest that pointers shouldn't test implicit for 0 either, i.e. if (line == 0) instead of if (line). The latter is regarded as such a common practice in C/C++ however that it can be used.

4.3.9 Variables should be declared in the smallest scope possible.

Keeping the operations on a variable within a small scope, it is easier to control the effects and side effects of the variable.

4.4 Loops

4.4.1 Only loop control statements must be included in the `for` construction

```
sum = 0;

for (i = 0; i < 100; i++) {
  sum += value[i];
}

// NOT: for (i = 0, sum = 0; i < 100; i++) {
//        sum += value[i];
//      }
```

Increase maintainability and readability. Make it crystal clear what controls the loop and what the loop contains.

4.4.2 Should initialise loop variables before the loop

Loop variables should be initialized immediately before the loop.

```
bool isDone = false;

while (!isDone) {
  ...
}

// NOT: bool isDone = false;
//      ...
//      some more code
///     ...
//      while (!isDone) {
//        ...
//      }
```

4.4.3 `do-while` loops can be avoided

do-while loops are less readable than ordinary while loops and for loops since the conditional is at the bottom of the loop. The reader must scan the entire loop in order to understand the scope of the loop. In addition, `do-while` loops are not needed. Any `do-while` loop can easily be rewritten into a `while` loop or a `for` loop. Reducing the number of constructs used enhance readbility.

4.4.4 The use of `break` and `continue` in loops should be minimized

These constructs can be compared to `goto` and they should only be used if they prove to have higher readability than their structured counterpart.

4.4.5 The form `while (true)` should be used for infinite loops

```
while (true) {
  ...
}

for (;;) { // NO!
  ...
}

while (1) { // NO!
  ...
}
```

Testing against 1 is neither necessary nor meaningful. The form for (;;) is not very readable, and it is not apparent that this actually is an infinite loop.

4.5 Conditionals

4.5.1 Complex conditional expressions must be avoided

Complex conditional expressions must be avoided. Introduce temporary boolean variables instead.

```
if ((elementNo < 0) || (elementNo > maxElement)||
     elementNo == lastElement) {
  ...
}
```

should be replaced by assuming that short-cutting is not required:

```
isFinished      = (elementNo < 0) || (elementNo > maxElement);
isRepeatedEntry = elementNo == lastElement;

if (isFinished || isRepeatedEntry) {
  ...
}
```

By assigning boolean variables to expressions, the program gets automatic documentation. The construction will be easier to read and to debug.

4.5.2 Should put the exceptional case in the `else`-part

The nominal case should be put in the `if`-part and the exception in the `else`-part of an `if` statement.

```
isError = readFile (fileName);

if (!isError) {
  ...
}
else {
  ...
}
```

Makes sure that the exceptions don't obscure the normal path of execution. This is important for both the readability and performance.

4.5.3 The conditional should be put on a separate line

```
if (isDone) {            // NOT: if (isDone) { doCleanup(); }
  doCleanup();
}
```

This is for debugging purposes. When writing on a single line, it is not apparent whether the test is really true or not.

4.5.4 Executable statements in conditionals must be avoided

```
// Bad!
if (!(fileHandle = open (fileName, "w"))) {
  ...
}

// Better!
fileHandle = open (fileName, "w");

if (!fileHandle) {
  ...
}
```

Conditionals with executable statements are just very difficult to read. This is especially true for programmers new to C/C++.

4.6 Miscellaneous

4.6.1 `goto` should not be used

`goto` *statements violates the idea of structured code. Only in some very few cases (for instance breaking out of deeply nested structures or for special needs when dealing with an error)* `should` `goto` *be considered, and only if the alternative structured counterpart (for example, exceptions) is proven to be less readable.*

4.6.2 Functions must always have the return value explicitly listed

```
int getValue() { // NOT: getValue()
  ...
}
```

If not exlicitly listed, C++ implies int return value for functions. A programmer must never rely on this feature, since this might be confusing for programmers not aware of this artifact.

4.6.3 `0` should be used instead of `NULL` in C++

NULL is part of the standard C library, but is made obsolete in C++.

5 Miscellaneous

5.1 Miscellaneous

5.1.1 The use of magic numbers in the code must be avoided

Numbers other than 0 and 1 must be declared as named constants instead.

If the number does not have an obvious meaning by itself, the readability is enhanced by introducing a named constant instead. A different approach is to introduce a method from which the constant can be accessed.

5.1.2 The use of magic strings in the code should be avoided

Strings of length 3 or greater should be considered declared as named constants instead.

Using a constant reduces the errors due to typing mistakes. Strings used in text messages should only be declared as constants if used more than once.

5.1.3 Use a decimal point for floating point constants

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;
double sum;
...
sum = (a + b) * 10.0;
```

This empasize the different nature of integer and floating point numbers even if their values might happen to be the same in a specific case.

Also, as in the last example above, it emphasize the type of the assigned variable (sum) at a point in the code where this might not be evident.

5.1.4 Floating point constants should always be written with a digit before the decimal point

```
double total = 0.5;             // NOT: double total = .5;
```

The number and expression system in C++ is borrowed from mathematics and one should adhere to mathematical conventions for syntax wherever possible. Also, 0.5 is a lot more readable than .5; There is no way it can be mixed with the integer 5.

5.1.5 Floating point comparison should be avoid

Floating point comparison is dangerous due to rounding errors. It should therefore be handled with care.

6 Layout and Comments

6.1 Layout

6.1.1 Tabs must be eight characters wide

Since the beginnig of time.

6.1.2 Basic indentation should be 2

```
for (i = 0; i < nElements; i++) {
  a[i] = 0;
}
```

Indentation of 1 is to small to emphasize the logical layout of the code. Indentation larger than 4 makes deeply nested code difficult to read and increase the chance that the lines must be split. Choosing between indentation of 2, 3 and 4, 2 and 4 are the more common, and 2 chosen to reduce the chance of splitting code lines.

Block layout should be as illustrated in example 1 below (recommended) or example 2, and must not be as shown in example 3. Function and class blocks must use the block layout of example 1.

```
// example 1
while (!done) {
  doSomething();
  done = moreToDo();
}

// example 2
while (!done)
{
  doSomething();
  done = moreToDo();
}

// example 3
while (!done)
  {
    doSomething();
    done = moreToDo();
  }
```

Example 3 introduce an extra indentation level which doesn't emphasize the logical structure of the code as clearly as example 1 and 2.

The class declarations should have the following form:

```
class SomeClass : public BaseClass {
  public:
    void doSomething ();

  protected:
    ...

  private:
    ...
}
```

This follows partly from the general block rule above.

The function declarations should have the following form:

```
void someMethod () {
  ...
}
```

This follows from the general block rule above. Note that there is an extra space before the `()`. When calling a function there should be no space.

The `if-else` statements should have the following form:

```
if (condition) {
  statements;
}

if (condition) {
  statements;
}
else {
  statements;
}

if (condition) {
  statements;
}
else if (condition) {
  statements;
}
else {
  statements;
}
```

This follows partly from the general block rule above. An else clause should not be on the same line as the closing bracket of the previous if or else clause:

```
if (condition) {
  statements;
} else {        // NO
  statements;
}
```

This is equivalent to the Sun recommendation. The chosen approach is considered better in the way that each part of the `if-else` *statement is written on separate lines of the file. This should make it easier to manipulate the statement, for instance when moving else clauses around.*

A for statement should have the following form:

```
for (initialization; condition; update) {
  statements;
}
```

This follows from the general block rule above.

An empty for statement should have the following form:

```
for (initialization; condition; update) {
}
```

This emphasize the fact that the for statement is empty and it makes it obvious for the reader that this is intentional. Empty loops should be avoided however.

A while statement should have the following form:

```
while (condition) {
  statements;
}
```

This follows from the general block rule above.

A do-while statement should have the following form:

```
do {
  statements;
} while (condition);
```

This follows from the general block rule above.

A switch statement should have the following form:

```
switch (condition) {
  case ABC:
    statements;
    // Fallthrough

  case DEF:
    statements;
    break;

  case XYZ1:
  case XYZ2:
    statements;
    break;

  default:
    statements;
    break;
}
```

Note that each `case` keyword is indented relative to the switch statement as a whole. Note also that no extra space before the : character exits. The explicit `// Fallthrough` comment must be included whenever there is a case statement without a `break` statement. Leaving the `break` out is a common error, and it must be made clear that it is intentional when it is not there.

A `try-catch` statement should have the following form:

```
try {
  statements;
}
catch (Exception &exception) {
  statements;
}
```

This follows partly from the general block rule above. The discussion about closing brackets for `if-else` statements apply to the `try-catch` statments.

The function return type must be put in the immediately before the function name.

```
void MyClass::myMethod (void) {
  ...
}
```

6.1.3 Fallthrough comment must be used

See the example above.

6.1.4 Empty blocks or on-line blocks must use brackets

Even if a block contains only a single statement, it must be enclosed in brackets.

```
if (0 == a) {
  doIt();
}

if (1 == a) {
  // do nothing because ...
}
```

If a if-clause starts out is one-line and is later extended it is easy to forget the brackets.

It is a common recommendation (Sun Java recommendation included) that brackets should always be used in all these cases.

6.2 White Space

6.2.1 Conventional operators should be surrounded by a space character

C++ reserved words should be followed by a white space. Commas should be followed by a white space. Colons should be surrounded by white space. Colons in `case` statements should not be surrounded by white space. Semicolons in `for` statments should be followed by one or two (recommended) space characters.

```
a = (b + c) * d;            // NOT: a=(b+c)*d

while (true) {              // NOT: while(true) ...

doSomething(a, b, c, d);    // NOT: doSomething(a,b,c,d);

case 100:                   // NOT: case 100 :

for (i = 0;  i < 10;  i++) { // NOT: for (i=0;i<10;i++){
```

Makes the individual components of the statements stand out. Enhances readability. It is difficult to give a complete list of the suggested use of whitespace in C++ code. The examples above however should give a general idea of the intentions.

6.2.2 The method name should be followed by a space

Method or function names should be followed by a white space in the declaration or implementation, but not when calling.

```
void doSomething (FILE *currentFile) {
}

doSomething(currentFile);
```

Allows to search for `doSomething (` *to find the declaration or implementation.*

6.2.3 Logical units within a block should be separated by one blank line

A blank line should be inserted before and after a `if` or `while` block.

```
int f = 1;

if (f > e) {
  if (g > h) {
    int a = f + e + g + h;

    if (a == 1) {
      ...
    }
  }

  int x = f + 1;

  ...
}
```

Enhance readability by introducing white space between logical units of a block.

6.2.4 Methods should be separated by blank lines

They should be separated with three blank lines in larger files.

The methods will stand out within the file.

6.2.5 Variables in declarations can be left aligned

```
AsciiFile *  file;
int          nPoints;
float        x, y;
```

Enhance readability. The variables are easier to spot from the types by alignment. Use alignment wherever it enhances readability.

```
if      (a == lowValue)    compueSomething();
else if (a == mediumValue) computeSomethingElse();
else if (a == highValue)   computeSomethingElseYet();

value = (potential        * oilDensity)   / constant1 +
        (depth            * waterDensity) / constant2 +
        (zCoordinateValue * gasDensity)   / constant3;

minPosition     = computeDistance (min,     x, y, z);
averagePosition = computeDistance (average, x, y, z);

switch (value) {
  case PHASE_OIL:   strcpy (string, "Oil");   break;
  case PHASE_WATER: strcpy (string, "Water"); break;
  case PHASE_GAS:   strcpy (string, "Gas");   break;
}
```

There are a number of places in the code where white space can be included to enhance readability even if this violates common guidelines. Many of these cases have to do with code alignment. General guidelines on code alignment are difficult to give, but the examples above should give a general clue.

6.3 Comments

6.3.1 Tricky code should not be commented but rewritten

In general, the use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.

6.3.2 All comments should be written in english

In an international environment english is the preferred language.

6.3.3 Use // for all comments, including multi-line comments

```
// Comment spanning
// more than one line.
```

Since multilevel C-commenting is not supported, using // comments ensure that it is always possible to comment out entire sections of a file using / */ for debugging purposes etc.*

There should be a space between the // and the actual comment. Comments should be included relative to their position in the code.

```
while (true) {          // NOT:    while (true) {
  // Do something       //         // Do something
  something();          //           something();
}                       //         }
```

This is to avoid that the comments break the logical structure of the program.

6.3.4 Must use development comments only during development

The comments `FIXME` and `TODO` must be used to flag code blocks which must be refactored. The release code must be free of such comments and code blocks.

6.3.5 You must use doxygen conventions for C++

Class and method header comments must follow the Doxygen conventions.

```
////////////////////////////////////////////////////////////////////////////
/// @brief returns the maximum of two numbers
////////////////////////////////////////////////////////////////////////////
```

Regarding standardized class and method documentation the Java development community is far more mature than the C++. This is of course becuase Java includes a tool for extracting such comments and produce high quality hypertext documentation from it. There have never been a common convention for writing this kind of documentation in C++, so when choosing between inventing your own convention, and using an existing one, the latter option seem natural. Also, there are JavaDoc tools for C++ available. See for instance Doc++ or Doxygen.

For inherited methods use

```
////////////////////////////////////////////////////////////////////////////
/// {@inheritDoc}
////////////////////////////////////////////////////////////////////////////
```

6.3.6 Sectionise large files

Use section descriptions to split large files into sections.

```
// ////////////////////////////////////////////////////////////////////////////
// constructors and destructors
// ////////////////////////////////////////////////////////////////////////////

...

// ////////////////////////////////////////////////////////////////////////////
// public methods
// ////////////////////////////////////////////////////////////////////////////

...
```

7 References

1. Code Complete,
   Steve McConnel - Microsoft Press
2. Programming in C++, Rules and Recommendations,
   M Henricson, e. Nyquist, Ellemtel (Swedish telecom),
   http://www.doc.ic.ac.uk/lab/cplus/c
3. Wildfire C++ Programming Style,
   Keith Gabryelski, Wildfire Communications Inc.,
   http://www.wildfire.com/ ag/Engineering/Development/C++Style/
4. C++ Coding Standard,
   Todd Hoff,
   http://www.possibility.com/Cpp/CppCodingStandard.htm
5. Doxygen documentation system,
   http://www.stack.nl/ dimitri/doxygen/index.html