C++ Programming Style Guidelines

Version 4.9, January 2011
Geotechnical Software Services

This document is available at http://geosoft.no/development/cppstyle.html

---

Table of Content

---

1 Introduction

This document lists C++ coding recommendations common in the C++ development community.

The recommendations are based on established standards collected from a number of sources, individual experience, local requirements/needs, as well as suggestions given in [1] - [4].

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. For guidelines on C++ programming *style* refer to the C++ Programming Practice Guidelines.

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 maximise 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:

n. Guideline short 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

1. 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.

2. 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.

3 Naming Conventions

3.1 General Naming Conventions

3. Names representing types must be in mixed case starting with upper case.
`Line, SavingsAccount`
Common practice in the C++ development community.

4. Variable names must be in mixed case starting with lower case.
`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;

5. 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 getMaxIterations() // NOT: MAX_ITERATIONS = 25
{
  return 25;
}
```

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

6. Names representing methods or functions must be verbs and written in mixed case starting with lower case.

```
getName(), computeTotalWidth()
```

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

7. Names representing namespaces should be all lowercase.

```
model::analyzer, io::iomanager, common::math::geometry
```

Common practice in the C++ development community.

8. 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.

9. Abbreviations and acronyms must not be uppercase when used as name [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.

10. 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.

11. Private class variables should have underscore suffix.

```
class SomeClass {
  private:
    int length_;
}
```

Apart from its name and its type, the *scope* of a variable is its most important feature. Indicating class scope by using underscore 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.

A side effect of the underscore naming convention is that it nicely resolves the problem of finding reasonable variable names for setter methods and constructors:

```
  void setDepth (int depth)
  {
    depth_ = depth;
  }
```

An issue is whether the underscore should be added as a prefix or as a suffix. Both practices are commonly used, but the latter is recommended because it seem to best preserve the readability of the name.

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.

12. 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;
```

13. All names should be written in English.

```
fileName;   // NOT: filNavn
```

English is the preferred language for international development.

14. Variables with a large scope should have long names, variables with a small scope can have short names [1].

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. Common scratch variables for integers are *i*, *j*, *k*, *m*, *n* and for characters *c* and *d*.

15. 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.

3.2 Specific Naming Conventions

17. 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.

18. 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.

19. 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.

20. The term *initialize* can be used where an object or a concept is established.

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

The american *initialize* should be preferred over the English *initialise*. Abbreviation *init* should be avoided.

21. 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.

22. Plural form should be used on names representing a collection of objects.

```
vector<Point>  points;
int            values[];
```

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.

23. 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.

24.The suffix *No* should be used for variables representing an entity number.

```
tableNo, employeeNo
```

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

An elegant alternative is to prefix such variables with an *i*: `iTable, iEmployee`. This effectively makes them *named* iterators.

25. Iterator variables should be called *i*, *j*, *k* etc.

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

for (vector<MyClass>::iterator i = list.begin(); i != list.end(); i++) {
  Element element = *i;
  ...
}
```

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.

26. 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();
```

27. 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,
next/previous, old/new, open/close, show/hide, suspend/resume, etc.
```

Reduce complexity by symmetry.

28. 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.
```

29. 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.

30. 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.

31. 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.

32. Exception classes should be suffixed with *Exception*.

```
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.

33. 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.

4 Files

4.1 Source Files

34. C++ header files should have the extension *.h* (preferred) or *.hpp*. Source files can have the extension *.c++* (recommended), *.C*, *.cc* or *.cpp*.

```
MyClass.c++, MyClass.h
```

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

35. A class should be declared in a header file and defined in a source file where the name of the files match the name of the class.

```
MyClass.h, MyClass.c++
```

Makes it easy to find the associated files of a given class. An obvious exception is template classes that must be both declared and defined inside a .h file.

36. All definitions should reside in source files.

```
class MyClass
{
public:
  int getValue () {return value_;}  // NO!
  ...

private:
  int value_;
}
```

The header files should declare an interface, the source file should implement it. When looking for an implementation, the programmer should always know that it is found in the source file.

37. File content must 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.

38. 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.

39. The incompleteness of split lines must be made obvious [1].

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

function (param1, param2,
          param3);

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

for (int 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.

4.2 Include Files and Include Statements

40. Header files must contain an include guard.

```
#ifndef COM_COMPANY_MODULE_CLASSNAME_H
#define COM_COMPANY_MODULE_CLASSNAME_H
  :
#endif // COM_COMPANY_MODULE_CLASSNAME_H
```

The construction is to avoid compilation errors. The name convention resembles the location of the file inside the source tree and prevents naming conflicts.

41. 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 <qt/qbutton.h>
#include <qt/qtextfield.h>

#include "com/company/ui/PropertiesDialog.h"
#include "com/company/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.

42. 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.

5 Statements

5.1 Types

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

Enforces information hiding.

44. The parts of a class must be sorted *public*, *protected* and *private* [2][3]. 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.

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

```
floatValue = static_cast<float>(intValue); // NOT: floatValue = intValue;
```

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

5.2 Variables

46. 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.

47. Variables must never have dual meaning.

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

48. 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.

49. Class variables should 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 [2].

Note that *struct*s 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. Use a class instead.

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

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

The *pointer-ness* or *reference-ness* of a variable is a property of the type rather than the name. C-programmers often use the alternative approach, while in C++ it has become more common to follow this recommendation.

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

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

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

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

54. 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.

5.3 Loops

55. Only loop control statements must be included in the `for()` construction.

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

Increase maintainability and readability. Make a clear distinction of what *controls* and what is *contained* in the loop.

56. Loop variables should be initialized immediately before the loop.

```
isDone = false;           // NOT: bool isDone = false;
while (!isDone) {         //      :
  :                       //      while (!isDone) {
}                         //        :
                          //      }
```

57. `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.

58. The use of `break` and `continue` in loops should be avoided.

These statements should only be used if they give higher readability than their structured counterparts.

60. 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.

5.4 Conditionals

61. Complex conditional expressions must be avoided. Introduce temporary boolean variables instead [1].

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
  :
}

// NOT:
if ((elementNo < 0) || (elementNo > maxElement)||
     elementNo == lastElement) {
  :
```

```
}
```

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

62. The nominal case should be put in the *if*-part and the exception in the *else*-part of an if statement [1].

```
bool isOk = readFile (fileName);
if (isOk) {
  :
}
else {
  :
}
```

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

63. 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.

64. Executable statements in conditionals must be avoided.

```
File* fileHandle = open(fileName, "w");
if (!fileHandle) {
  :
}

// NOT:
if (!(fileHandle = open(fileName, "w"))) {
  :
}
```

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

5.5 Miscellaneous

65. The use of magic numbers in the code should be avoided. Numbers other than *0* and *1* should be considered 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.

66. 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 emphasizes the different nature of integer and floating point numbers. Mathematically the two model completely different and non-compatible concepts.

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

67. 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.

68. 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.

69. `goto` should not be used.

Goto statements violate the idea of structured code. Only in some very few cases (for instance breaking out of deeply nested structures) should goto be considered, and only if the alternative structured counterpart is proven to be less readable.

70. "0" should be used instead of "NULL".

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

6 Layout and Comments

6.1 Layout

71. Basic indentation should be 2.

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

Indentation of 1 is too small to emphasize the logical layout of the code. Indentation larger than 4 makes deeply nested code difficult to read and increases 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.

72. Block layout should be as illustrated in example 1 below (recommended) or example 2, and must not be as shown in example 3 [4]. Function and class blocks must use the block layout of example 2.

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

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

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

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

73. The `class` declarations should have the following form:

```
class SomeClass : public BaseClass
{
  public:
    ...

  protected:
    ...

  private:
    ...
}
```

This follows partly from the general block rule above.

74. Method definitions should have the following form:

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

This follows from the general block rule above.

75. The `if-else` class of 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. However, it might be discussed if an `else` clause should be on the same line as the closing bracket of the previous `if` or `else` clause:

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

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.

76. A `for` statement should have the following form:

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

This follows from the general block rule above.

77. An empty `for` statement should have the following form:

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

This emphasizes 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.

78. A `while` statement should have the following form:

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

This follows from the general block rule above.

79. A `do-while` statement should have the following form:

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

This follows from the general block rule above.

80. A `switch` statement should have the following form:

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

  case DEF :
    statements;
    break;

  case XYZ :
    statements;
    break;

  default :
    statements;
    break;
}
```

Note that each `case` keyword is indented relative to the switch statement as a whole. This makes the entire switch statement stand out. Note also the extra space before the *:* character. The explicit *Fallthrough* comment should 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.

81. 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.

82. Single statement `if-else`, `for` or `while` statements can be written without brackets.

```
if (condition)
  statement;

while (condition)
  statement;

for (initialization; condition; update)
  statement;
```

It is a common recommendation that brackets should always be used in all these cases. However, brackets are in general a language construct that groups several statements. Brackets are per definition superfluous on a single statement. A common argument against this syntax is that the code will break *if* an additional statement is added without also adding the brackets. In general however, code should never be written to accommodate for changes that *might* arise.

83. The function return type can be put in the left column immediately above the function name.

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

This makes it easier to spot function names within a file since they all start in the first column.

6.2 White Space

84.
- 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.
- Semicolons in for statments should be followed by a space character.

```
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.

85. Method names can be followed by a white space when it is followed by another name.

```
doSomething (currentFile);
```

Makes the individual names stand out. Enhances readability. When no name follows, the space can be omitted (`doSomething()`) since there is no doubt about the name in this case.

An alternative to this approach is to require a space *after* the opening parenthesis. Those that adhere to this standard usually also leave a space before the closing parentheses: `doSomething( currentFile );`. This do make the individual names stand out as is the intention, but the space before the closing parenthesis is rather artificial, and without this space the statement looks rather asymmetrical (`doSomething( currentFile);`).

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

```
Matrix4x4 matrix = new Matrix4x4();

double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

matrix.setElement(1, 1,  cosAngle);
matrix.setElement(1, 2,  sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2,  cosAngle);

multiply(matrix);
```

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

87. Methods should be separated by three blank lines.

By making the space larger than space within a method, the methods will stand out within the class.

88. 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.

89. 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(phase, "Oil");   break;
  case PHASE_WATER : strcpy(phase, "Water"); break;
  case PHASE_GAS   : strcpy(phase, "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

90. Tricky code should not be commented but rewritten! [1]

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

91. All comments should be written in English [2].

In an international environment English is the preferred language.

92. 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, and comments should always start with an upper case letter and end with a period.

93. Comments should be included relative to their position in the code. [1]

```
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.

94. Class and method header comments should follow the JavaDoc conventions.

Regarding standardized class and method documentation the Java development community is more mature than the C/C++ one. This is due to the standard automatic Javadoc tool that is part of the development kit and that help producing high quality hypertext documentation from these comments.

There are Javadoc-like tools available also for C++. These follows the same tagging syntax as Javadoc. See for instance Doc++ or Doxygen.

7 References

[1] Code Complete, Steve McConnell - 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%2b%2b.rules/

[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

Acknowledgements

Thanks to Robert P.J. Day for valuable contributions.