Programming Style Guidelines

Version 1.1, April 2001 Copyright © D-Bross

Table of Content

Introduction

During the years, we tryed different coding styles in our development. Finally, we decided to collect all the best and put them together in one document.

This document describes coding standards we use in our C++ programs. But we use the exactly the same rules when using other programming languages.
For example: C, Java, JavaScript, C#, VB, PHP etc.

Standardization is Important

Good Points

When a project tries to adhere to common standards a few good things happen:

- program source codes look like they are written by one person. This improves team productivity.
- programmers can go into any code and figure out what's going on
- new people can get up to speed quickly
- people new to C++ are spared the need to develop a personal style and defend it to the death
- people new to C++ are spared making the same mistakes over and over again
- people make fewer mistakes in consistent environments
- programmers have a common enemy :-)

Bad Points

Now the bad:

- the standard is usually stupid because it was made by someone who doesn't understand C++
- the standard is usually stupid because it's not what I do
- standards reduce creativity
- standards are unnecessary as long as people are consistent
- standards enforce too much structure
- people ignore standards anyway

Discussion

Are standards necessary for success? Of course not. But they help, and we need all the help we can get! Be honest, most arguments against a particular standard come from the ego. Few decisions in a reasonable standard really can be said to be technically deficient, just matters of taste. So be flexible, control the ego a bit, and remember any project is fundamentally a team effort.

Interpretation

The use of the word "shall" or "must" in this document requires that any project using this document must comply with the stated standard.

The use of the word "may" or "should" designates optional requirements.

Naming Conventions

General Naming Conventions

Names representing types must be in mixed case starting with upper case, each next word starting with upper case and no underbars ('_').
Line, SavingsAccount

Class names must be in mixed case starting with upper case, each next word starting with upper case and no underbars ('_').
DownaloadAgent, FileNotFoundException
• Name the class after what it is. If you can't think of what it is that is a clue you have not thought through the design well enough. • Compound names of over three words are a clue your design may be confusing various entities in your system. Revisit your design. See if your objects have more responsibilities than they should. • Avoid the temptation of bringing the name of the class a class derives from into the derived class's name. A class should stand on its own. It doesn't matter what it derives from. • Suffixes are sometimes helpful. For example, if your system uses agents then naming something *DownloadAgent* conveys real information.

Variable names must be in mixed case starting with lower case, each next word starting with upper case and no underbars ('_').

Line   line;
Account savingsAccount;

Makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration Line line;

Named constants (including enumeration values) must be all uppercase using underscore to separate words.
MAX_ITERATIONS, COLOR_RED, PI

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.

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

getName(), computeTotalWidth()

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

Usually every method and function performs an action, so the name should make clear what it does: c*heckForErrors()* instead of *ErrorCheck()*, d*umpDataToFile()* instead of *DataFile()*. This will also make functions and data objects more distinguishable.

Names representing namespaces should be all lowercase.

analyzer, iomanager, mainwindow

Names representing template types should be a single uppercase letter.

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

This makes template names stand out relative to all other names used.

Abbreviations and acronyms must be uppercase when used as name.

```
exportHTMLSource();    // NOT: exportHtmlSource();
openDVDPlayer();       // NOT: openDvdPlayer();
```

This improves readability.

Prefix pointer variables with 'p'

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

This way the programmer will always keep in mind the variable is a pointer and the memory it points to should be deallocated.

Global variables and functions should always be referred to using the :: operator.

::g_mainWindow.open(), ::g_applicationContext.getName(), ::create_window()

In general, the use of global variables and function should be avoided. Consider using singleton objects instead.

Global variables should have prefix 'g_'.

```
Logger  g_log;
Logger* g_pLog;
```

In general, the use of global variables should be avoided. Consider using singleton objects instead.

Class variables should have prefix 'm_' starting with lower case and each next word starting with upper case.

```
class SomeClass
{
  private:
    int          m_length;
    long         m_initialSize;
    StatusInfo*  m_pStatus;
}
```

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.

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

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

Static class variables should have prefix 's_'.

```
class SomeClass
{
  private:
    static  int         s_length;
    static  StatusInfo*  s_pStatus;
}
```

Apart from its name and its type, the *scope* of a variable is its most important feature. Indicating class scope by using 's_' prefix for static class members makes it easy to distinguish static class variables from non static ones.

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 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.

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

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

All names should be written in english.

```
fileName;    // NOT:   ImeNaFajl
```

English is the prefered language for international development.

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*.

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.

Specific Naming Conventions

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 );
```

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 potential time consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

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.

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.

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.

The suffix *List, Array, Map* etc. should be used on names representing a list, array or map 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)) must 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.

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.

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

tableNo, employeeNo, orderID

Iterator variables should be called *i*, *j*, *k* etc. and STL iterator variables *it* .

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

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

The prefix *is* should be used for boolean variables and methods.

isSet, isVisible, isFinished, isFound, isOpen

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();
```

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, etc.

Reduce complexity by symmetry.

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.
```

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.

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 or if the enum is defined inside a class: Color::RED, Airline::AIR_FRANCE etc.

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.

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.

Files

Source Files

C++ source files should have the extension *.cpp*. Header files should have the extension *.h*.

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

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.

class MyClass defined in MyClass.h and implemented in MyClass.cpp

Makes it easy to find the associated files of a given class.

All definitions should reside in source files.

```
class MyClass
{
  public:
    int getValue () {
        ...    //    some code - NO !
        ...
    }
    ...
  private:
    int m_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. The obvious exception to this rule is of course inline functions that must be defined in the header file.

Inline methods should be max 2 statements long.

```
inline
int MyClass::getValue() {
   return  m_value;
}

inline
void  MyClass::computeAverage() {
//  This line is 3 statements long that's why this method should not be inline.
    m_size++; m_total = computeTotal(); m_average = m_total/m_size;
}
```

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.

Special characters like 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.

Include Files and Include Statements

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

```
#ifndef MOD_FILENAME_H
#define MOD_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.

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 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.

Statements

Types

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

Enforces information hiding.

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.

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

```
floatValue = static_cast<float> (intValue);    // YES!
floatValue = (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.

Variables

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.

Variables must never have dual meaning.

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

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.

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 variables public.

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.

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;
```

A pointer is a variable of a pointer type (float* x). Do not declare more than one pointer or reference in one statement because it is easily to make a mistake: float* x, y, z; is equivalent with float* x; float y; float z;   // NOT float* x; float* y; float* z;

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

```
if( nLines != 0 )       // NOT:   if( nLines )
if( pLine  !=  NULL )   // NOT:   if( pLine )
```

Do not declare multiple variables in one statement.

```
int      nLines, x, y, i, j;       // NO !
Line* pLine, pColumn;      // NO !
```

It is a common mistake as in the second definition. The programmer wanted to declare *pColumn* as a pointer.

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.

Loops

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 it crystal clear what controls the loop and what the loop contains.

Loop variables should be initialized immediately before the loop.

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

***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) should goto be considered, and only if the alternative structured counterpart is proven to be less readable.

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

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

Continue and break like goto should be used sparingly as they are magic in code. With a simple spell the reader is beamed to god knows where for some usually undocumented reason.

The two main problems with continue are:

- It may bypass the test condition
- It may bypass the increment/decrement expression

Consider the following example where both problems occur:

```
while( TRUE )
{
   ...
   // A lot of code
   ...
   if( /* some condition */ )
   {
      continue;
   }
   ...
   // A lot of code
   ...
   if( i > STOP_VALUE )
   {
      break;
   }
}
```

Note: "A lot of code" is necessary in order that the problem cannot be caught easily by the programmer.

From the above example, a further rule may be given: Mixing continue with break in the same loop is a sure way to disaster.

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.

Conditionals

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

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

should be replaced by:

```
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.

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.

Executable statements in conditionals must be avoided.

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

// Better !
fileHandle = open( fileName, "w" );
if( !fileHandle )
{
    :
}

// Good
if( checkState( fileHandle ) != -1 )
{
    :
}

// Absolutely forbidden !!!
if( bFlag == true || (checkState( fileHandle ) != -1 ) )
{
    :
}
```

Conditionals with executable statements are just very difficult to read. Exception of this rule is appropriate if this will save additional local variable and will increase readability. **But only if the statement in the conditional do not contain logical expressions.**

Miscellaneous

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.

One statement per line.

```
i++;
m_size = m_len*10;
m_average = m_total/m_size;

// NOT: getValue()
i++; m_size = m_len*10; m_average = m_total/m_size;
```

There should be only one statement per line.

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.

Layout and Comments

Layout

Basic indentation should be one TAB.

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

Block layout should be as illustrated in example 1 below, must not be as shown in example 2 and example 3.

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

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

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

}		}

The class declarations should have the following form:

```
//  NO !  class SomeClass : BaseClass
class SomeClass : public BaseClass, protected B2, private B3
{
  public:
    ...
  protected:
    ...
  private:
    ...
}
```

This follows partly from the general block rule above. Order scope specifiers from most public to most private. Explicitly specify the scope of the base classes in the same order.

The function declarations should have the following form:

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

This follows from the general block rule above.

Methods should be no more than one page long.

All the code of a method should be kept inside a single or up to 2 pages on the screen. Longer methods shoud be converted to call functions on logical units and thus to reduce their size. Besides readability this also improves code maintenance.

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 from the general block rule above. A little break of the rule is the opening brace of *else* which must be on the same line. This is because *else* is a short word and this will not reduce readability of the code. 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.

A for statement should have the following form:

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

This follows from the general block rule above.

An empty statement should have the following form and should be documented:

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

Always document a null body for a *for* or *while* statement so that it is clear that the null body is intentional and not missing code. 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;
        // FALL THROUGH

    case DEF :
        statements;
        break;

    case XYZ :
    {
        statements;
    }
    break;

    default :
        statements;
        break;
}
```

- Each case keyword is indented relative to the switch statement as a whole. This makes the entire switch statement stand out.
- Extra space before the *:* character
- Falling through a case statement into the next case statement shall be permitted as long as a comment is included. Leaving the break out is a common error, and it must be made clear that it is intentional when it is not there.

- The *default* case should always be present and trigger an error if it should not be reached, yet is reached.
- If you need to create variables put all the code in a block.

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.

Single statement if, for or while statements can be written without brackets but in this case must be on a separate line.

```
if( condition )
    statement;

while( condition )
    statement;

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

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

White Space

- Conventional operators should be surrounded by a space character.
- 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.

Method open brace should be followed by a white space when there is at least one argument. Method closing brace should follow a white space when there is at least one argument.

```
doSomething( currentFile );    // NOT:   doSomething (currentFile);
```

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

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

Methods and functions should be separated by a comment line of slashes and one blank line.

```
void function1()
{
}
////////////////////////////////////////

void function2()
{
}
////////////////////////////////////////
```

This way the methods and functions will stand out within the file. Exception of this rule is permitted only in header files, where inline methods are defined within a group of logically connected methods.

```
inline
int SomeClass::getSize()
{
}

inline
void SomeClass::setSize( int size )
{
}
//////////////////////////////////////////
```

Variables in declarations should 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 enhanbces 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.

Comments

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.

All comments should be written in english.

In an international environment english is the preferred language.

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.

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.