LFC Coding Style

Leonard Moşescu
(lemo@metasoft.ro)

Contents

1 Introduction

This document describes the official LFC guidelines for writing/formatting C++ code. It is intended to help LFC developers in order to ease the creation of high quality, easy to read code.

These guidelines don't bring immediate advantages in code speed/memory usage, neither do they guarantee high quality code. But the benefit of following these guidelines is at least as valuable: we'll get clean, readable code that is easy to maintain and allows for extension. That is vital for a team!

As you may know, there are quite a few C++ coding styles out there. Every good team and programmer follows a coding style. There are not many objective arguments to compare different coding styles, and to decide that one style is better than another, and in any case, any style is better than no style. So the style presented here makes no claims to be better than other styles, it's a style on which LFC developers agree on; it tries to offer a set of guidelines to unify the way code 'looks' across the project, without placing a burden on the programmer.

The basic goal for LFC code regarding coding style is that all of the LFC code should use the same style (as opposed to: 'this is my module, I'll just use my own style').

This document will try to cover most of the coding style issues, but at the same time it should be reasonably brief, so it can be easy to use, by avoiding to list hundreds of pages with all the cases and details. When in doubt, take a look at the code already written and/or ask for advice from the other team members (and if it's the case, update this doc).

If your code doesn't look like the rest of LFC code or if other members are having trouble reading it, then consult this doc and update your code.

Happy coding!

1.1 Formating tools

There are various tools that can auto format source code to one or more coding styles. While those tools are useful to format external code (from other projects, etc.), the new code should be written from the start to conform to the coding style.

1.2 Conventions

In this document there will be examples of code. The good formatting example will be green and the bad formatting will be red.

2 General guidelines

- The code should be clear and simple to read. Avoid tricks and use special optimizations only when they are *really* needed
- The code formatting should follow and express the logical structure of the program (e.g. group similar code together)
- Public interfaces (everything that is exposed by a class, global functions, constants) should be clearly documented (and should not require reading through the implementation to understand basic functionality)
- Try to avoid abusing C++ features to create an 'extended language' - e.g. try to avoid macros (use a better alternative when possible: inline functions, C++ constants, etc.) - LFC C++ should be *C++* code!
- Avoid creating an unjustified number of synonymous names for the same entity
- Coding style should be consistent - all LFC code should look the same (no matter who wrote it)
- Try to fit the code in 80 columns (break extra long lines, see the guides for breaking lines)
- Others should enjoy reading your code as much as you enjoy writing it

3 Identifiers naming

- Names should be suggestive (BAD: `a23, bjXY, xwv`) and concise (BAD: `theMaximumValueForTheInterval`)
- When an identifier is composed of more words, each intermediary word will be capitalized (OK: `lastEvent, ComboBox, m_windowList, setFixedSize`) and not using underline, or without any form of separation (BAD: `last_event, setfixedsize`)

3.1 Variables names (global, member, auto, ...)

- Variables identifiers should start with a lowercase letter (OK: `totalTime, i, cmdOk`, BAD: `TotalTime, CmdOK`)

- In some cases (pointers, bool) identifiers should be prefixed according to their type, but without abuse. Use the following prefixes:

Type	Prefix	Examples
Pointer (regardless of base type)	p	`pObject, pItem, pNext`
Bool	b	`bFlag, bExclusive`

- Non-static protected/private data members will start with '`m_`' and static data members with '`s_`' (OK: `m_value, m_pParent, m_range, s_timersMap`)
- Global data that is not exposed to the LFC user should be prefixed with '`g_`'
- Public data members and public global data will not be prefixed with '`m_`', '`s_`' or '`g_`' (although public data should be avoided in general)

3.2 Types (class, enum, typedef)

- Type names begin with uppercase letter with no prefix (OK: `Object, DWord, GridLayout`, BAD: `CObject, TList, byte`)
- Template arguments are spelled with all letters uppercase (OK: `template<class TYPE> class List`)
- LFC interfaces classes (see LFC design specs) start their name with _ (underline)

3.3 Functions (global, methods, ...)

- Function (global or members) names begin with lowercase letter and no prefix (OK: `read(), isRunning(), initInstance()`, BAD: `Read(), bIsRunning(), m_initInstance()`)
- Methods pairs that set/get the values of an 'attribute' should be named as follows:

Attribute name	`TextColor`
Set method	`void setTextColor(const Color &color)`
Get method	`Color textColor() const`

(BAD: `getTextColor()` or `textColor()` overloaded for both set/get)

3.4 Constants

- Constants names could be spelled in uppercase (OK: `BUFFER_SIZE`)
- Constants declared within a class (like constants or enum members) could be spelled like normal variables, with an extra prefix to group set of flags together (OK: `File::flReadAccess, File::flWriteAccess`)

3.5 Namespaces

- Namespace names follow the same naming conventions as identifier names (starting with lowercase and each intermediate word is capitalized). There are some exceptions, for example PAL namespaces (OK: `lfc, posixPAL`, BAD: `LFC, Lfc`)

4 Source code organization

4.1 Source file names

- Source file names follow the same naming conventions as variable names (starting with lowercase and each intermediate word is capitalized). There are a few exceptions, for example lfc headers and PAL files (OK: `thread.cpp, windowManager.cpp`, BAD: `Thread.cpp, window_manager.cpp`)
- LFC header files follow an hierarchical naming style, starting with lfc and adding the name of each group separated by . (dot) and they have no 'extension' (OK: `lfc.streams.file`, BAD: `lfcFile.h, lfc.hpp`)
- Source files will use the following suffixes (extensions):

C++ source file	.cpp
C++ header *except main lfc headers*	.hpp
C++ source intended to be included/inlined	.inl
LFC headers	none

4.2 Location (hierarchical structure)

LFC core headers	core/include
LFC core sources	core/src
LFC core inline sources	core/include/inline
PAL headers	core/include/*platform* (platform is win32 or posix)
PAL sources	core/src/*platform* (platform is win32 or posix)
Samples	samples/*sample*
Tests	tests/*category*

4.3 Source file structure

- Lines should not be longer than 80 columns (see line breaking guidelines)
- Every source file (C++ code or header) should begin with the following header:

```
////////////////////////////////////////////////////////////////////////
//
// project       :  LFC2
// file          :  fileName.cpp
// description   :  implementation of class File methods
//
////////////////////////////////////////////////////////////////////////
//
// notes:
//       - source file notes here (optional)
//
////////////////////////////////////////////////////////////////////////
```

4.3.1 C++ headers

C++ header files should use the following technique (to avoid multiple inclusions of the same header):

```
#ifndef __LFC_INTERFACES_SCANABLE__
#define __LFC_INTERFACES_SCANABLE__

...  the rest of the header ...

#endif // __LFC_INTERFACES_SCANABLE__
```

4.3.2 C++ sources

No specific observations.

4.3.3 C++ inline sources

No specific observations.

5 Comments

- Comments should add extra information to the code (for a human reader) and not duplicate what the code does (and should be brief, easy to read)
- Comments should describe the logic of classes, methods, or blocks of code with the same purpose, and not each individual line
- In LFC code all comments will be C++ style (//) (single line or multiple line)
- Comments will be indented with the code, and the comments should be on separate lines preceding the code (and not on the same line with the code)

- Each function (global or member) should be documented in a function comment block before the function (comments used as metainfo for doxygen might take this role):
  - Brief description of the function
  - Description of each parameter
  - Description of return value
  - List potential exceptions that the function might throw
- Additional information for a function could be:
  - Pre-conditions, post-conditions and invariants
  - List of related entities (functions, classes, ...)
  - Description of the algorithm used (if non trivial) - including time and space complexity classes
  - Side effects (should be avoided!)
  - Thread safe or not

6 Indentation guides

- Indentation should use spaces (and not TAB chars - to be easy to use across different text editors). You may configure your favorite editor to replace TAB chars with soft tabs, that is with spaces
- Indentation size is 4 spaces
- A C++ block is indented as follows:
  - Enclosing pair of brackets should be on the same column and should be the only thing in that line (with the exception of do..while statement, see below)
  - The block contents should be indented as follows:

OK:
```
if(condition)
{
    // block code
    ...
}
```

BAD:
```
if(condition) {
    // block code
    ...
}
```

BAD:
```
if(condition)
    {
    // block code
    ...
    }
```

- goto labels are starting at the same column as the enclosing function brackets

7 Functions (methods)

Functions and methods will be formatted as follows:

```
Type foo(T1 arg1, T2 arg2, ...)
{
}

and

Type Class::foo (T1 arg1, T2 arg2, ...)
{
}
```

8 Classes

Classes will have the following general organization:

```
class ClassName :  access Base
{
      ...  friends declaration ...

access:
      ...  member types ...

access:
      ...  constructors, destructors ...

access:
      ...  LFC signals ...

public:
      ...  public data members ...

public:
      ...  public methods ...

protected:
      ...  protected methods ...

private:
      ...  private methods ...

protected:
      ...  protected data ...
```

```
private:
      ...  private data ...

};
```

8.1 General class formatting guidelines

- The body of methods defined inside the class should not be followed by semicolon
- Don't rely on default access (exception: structures declared with 'struct' keyword)
- Methods should be grouped into logical groups

8.2 Class templates

```
template<class T1, class T2, ...>
class ClassName :  access Base
{
};
```

8.3 Special class members

- Constructors/destructors (as you can see, base classes are initialized before data members):

```
X::X(T1 arg1, T2 arg2) :
    Base(arg1),
    m_data(arg2)
{
    ...
};
```

- Operators should be spelled as '`operator+(...)`'

8.4 Multiple base classes and LFC 'interfaces'

In LFC, all classes, defined in the library or in an application using LFC, should inherit from Object class (directly of indirectly) and from zero or more 'interfaces' (see LFC design doc about 'LFC interfaces'; you can recognize them because their name starts with an _ (underscore)).

When you inherit from a base class + one or more 'interfaces' you should use this formatting:

```
class File :
    public Object,
    public virtual _Input<char>,
    public virtual _Output<char>,
    public virtual _Named
{
    ...
};
```

9 C++ statements

9.1 If (if...else if...else)

```
if(expression)
    statement


if(expression)
    statement1
else
    statement2
```

```

if(expression1)
    statement1
else if(expression2)
    statement2
else
    statement3
```

9.2 Switch

```
switch(expression)
{
case const1:
    ...  statements ...

case const2:
    ...  statements ...

default:
    ...  statements ...
}
```

9.3 While, do..while, for

```
while(expression)
    statement

do
    statement
while(expression);

do
{
    ...  statements ...
} while(expression);

for(exp1; exp2; exp3)
    statement
```

9.4 Try..catch

```
try
{
    ...  statements ...
}
catch(T1 e1)
{
}
catch(...)
{
}
```

9.5 Goto

```
{
    goto label;
    ...
label:
    statement
    ...
}
```

10 Declarations, expressions

- In expressions, binary operators will be separated from their operands by a space (with the exception of operator ',' which will have a space only after it (OK: `i = 10, j = 0;`))

- (), [] and unary operators will not be separated with space from their operands (OK: `t[a + b] = foo(*x)`, BAD: `t [ a+b ]=foo ( * x )`)
- Don't abuse (). Use them only to force a different evaluation order or to enhance expression clarity
- When declaring data members or variables, each variable will be on a separate line
- When declaring variables with compound type (pointers, references, arrays), the operators (*, &, [], ()) will be attached to the identifier and not to the base type (OK: `int *v[100], double &ref`, BAD: `int* v[100], double & ref`)
- When declaring the prototype of a function/method use the names of the paramenters (although C++ allows to skip them)
- Definition of default values for functions arguments will be made in function's prototype (and not in function's definition)

11 Spacing

- **Breaking lines longer than 80 columns:**
  - If possible without affecting code size and performance, break the statement into multiple statements
  - Long string constants can be divided into multiple string constants concatenated (remember: "Hello " "world!\n" is the same with "Hello world!\n")
  - When there is a long expression, the expression can be breaked *after* an operator and indenting the rest of the expression. Try to avoid breaking an expression in the middle of an operand for a lower precedence operator:

```
lfcOut << hours << ", " << minutes << ", " << seconds <<
    " (" << (hours * 60 * 60 + minutes * 60 + seconds) << ")\n";
```

- **Vertical spacing**
  - Use 2 empty lines to separate classes and functions
  - Use at least one empty line at the source file end (but no more than 3)
  - You can use an empty line to separate logical groups of lines, method declarations, ...