C++ Programming Style Guide for `cXbase`

Éric Poirier

June 17, 2017

Version: 1 (Compiled June 17, 2017)
Document: cx_cpp_norm.pdf
Collaborations: Éric Poirier

Chapter 1

General

1.1 C++11 is the default

Throughout this document, you will notice that C++11 is used whenever possible. This same behaviour is expected from any programmer for the `cXbase` project. C++11 is becoming the new standard in the industry and it is important to create as modern a codebase as possible.

For now, C++14 and C++17 features are not to be used (unless there is no confortable way around it). When C++17 will become the new standard (that is, it will ship with most compilers), it will be possible to use its features.

1.2 Meaningful names

Try to produce selft-documenting code which requires a minimum analysis from the reader. In a perfect world, names given to variables, functions and so on should never add to the analysis a user must perform while reading the code. Names should be clear and unambiguous. For example:

```cpp
// Not OK: unclear, needs analysis from reader.
double mean(std::vector<double> V)
{
    double x{0};

    for(auto& v : V)
    {
        x += v;
    }

    return x / V.size();
}

// OK: very clear
double meanValue(std::vector<double> p_dataPoints)
{
    double sumOfAllPoints{0};
```

```
    for(auto& dataPoint : p_dataPoints)
    {
        sumOfAllPoints += dataPoint;
    }

    return sumOfAllPoints / p_dataPoint.size();
}
```

Don't fear long names, as long as they are clear!

1.3 Source code organization

The source code should always be organized in multiple files.

1.3.1 Classes and structs

A header file (with the extention `.h` named after the class it defines. This header file contains the class definition but not its implementation. The implementation should be in a source file named after the class it implements with a `.cpp` extension.

1.3.2 Templates

Template classes need to have the method definitions inside the header file. To keep declaration and implementation separated, do the following:

```
// Foo.cpp (excluded from build):
template<typename T>
T Foo<T>::doSomething()
{
    // code...
}

// Foo.h
template<typename T>
class Foo
{
public:

    T doSomething();

private:

    T anAttribute;
}

#include "Foo.cpp" // Important!
```

Keep **one and only one** class, struct or template should be defined in a given pair of files. Finally, include header guards for each header file created. For example, for a header file containing the declaration of a class `Foo`:

```
// Foo.h

#indef FOO_H_
#define FOO_H_

Foo
{
    // class definition...
};

#endif // FOO_H_
```

1.4 Language and characters

To avoid encoding issues, always use English ans ASCII characters either in your programming or your commenting.

IMPORTANT: tab characters should be replaced by four (4) space characters. Tab characters may be interpreted differently by different applications and therefore are a source of ambiguity which should be eleminated.

Chapter 2

Style

2.1 Identation

Allman's ident style was chosen for this project because of its clarity. This style puts the brace associated with a control statement on the next line, indented to the same level as the control statement. Statements within the braces are indented to the next level.

```
// OK: Clear and well spaced-out
void Foo::aMethod(int p_x, int p_y)
{
    if(p_x == p_y)
    {
        doSomething();
        z++;
    }
    else
    {
        doSomethingElse();
        z--;
    }
}

// Not OK: Less clear and more error prone, esppecially for brace matching.
void Foo::aMethod(int p_x, int p_y){

    if(p_x == p_y){
        doSomething();
        m_z++;
    }
    else{
        doSomethingElse();
        m_z--;
    }
}
```

From Wikipedia:

> Consequences of this style are that the indented code is clearly set apart from the containing statement by lines that are almost completely whitespace and the closing brace lines up in the same column as the opening brace. Some people feel this makes it easy to find matching braces. The blocking style also delineates the actual block of code from the associated control statement itself. Commenting out the control statement, removing the control statement entirely, refactoring, or removing of the block of code is less likely to introduce syntax errors because of dangling or missing braces. Furthermore, it's consistent with brace placement for the outer/function block.

2.2 Naming format conventions

The various formats presented here aim at marking a clear distinction between any piece of code used. They bring further distinctions among types of code structures and help avoid errors.

2.2.1 Variables

Variables need to follow a `camelCase` format:

```cpp
int main()
{
    // OK
    int aNewVariable{0};
    Foo someNewObject;

    // Not OK
    char MyChar{'a'};        // Not in camelCase.
    Bar  SOME_NEW_OBJECT;    // Not in camelCase and uses underscores.

    return 0;
}
```

Constant need to be in `UPPERCASE` format. If you want to separate words for a constant, only the underscore (_) character is allowed, as in `UPPERCASE_CONSTANT`:

```cpp
int main()
{
    // OK
    const double PI{3.1416};
    const std::string FRENCH_SALUTATION{"Bonjour"};

    // Not OK
    const char MyChar{'a'};      // Not in UPPERCASE.
    const Bar  SOME-NEW-OBJECT; // Does not use underscores.
    const Foo  SomeNewObject;   // Not in UPPERCASE and does
                                // not use underscores.

```

```
    return 0;
}
```

To differentiate parameters from regular variables or member variables, the prefix (p_) has to be added before the parameter variable name (which shoul be in `camelCase`. In other words, the format is `p_myParameter`:

```
double middle(double p_first, double p_second)
{
    return (p_first + p_second) / 2;
}
```

Like parameters, member variables often need to be distinguished from other variables, including parameters. The prefix `p_` needs to be added befor the variable name (in `camelCase`). The format is: `m_memberVariable`:

```
Foo::Foo(double p_number) : m_number{p_number}
{
    // Maybe some other work...
}
```

Notice in the last code snippet –in the constructor's initialization list– how the parameter/member variable distinction is clear.

2.2.2 Functions and methods

Function and method names follow the same rules as variables, that is they are written in `camelCase`. Of course, constructors and destructors are exceptions of this.

```
// Function:
void myFunction()
{
    // Some work...
}

// Method for class Foo:
void Foo::myMethod()
{
    // Some work...
}
```

2.2.3 MACROs

MACRO names follow the same rules as constants, that is they are written in `UPPERCASE`, possibly with underscores between seperated words. MACRO parameters, however, do not follow the same conventions as regular parameters. Rather, the format `__parameterName__` is used. These two rules combined make it very clear that the piece of code used is not a function or a method, but a MACRO.

```
#define PRINT_MACRO() std::cout << "macro"; // OK

#define PRINT_MESSAGE(__theMessage__) \
    std::cout << __theMessage__; // OK

#define printMessage(__theMessage__) \
    std::cout << __theMessage__; // Not OK, could be mixed with function name.

#define PRINT_MESSAGE(p_theMessage) \
    std::cout << p_message; // Not OK, same reason
```

2.2.4 Classes, structures and enumerations

A class name uses the `CamelCase` format. They always start with an uppercase letter. If more than one word is used to describe the class, uppercase letters for every first letter of a word are used to distinguish them. This helps avoid confusion between a class name and an instance name.

```
// OK
class Foo
{
public:
    Foo(int p_value);

private:
    int m_value;
};

// OK
struct Bar
{
    int m_firstValue;
    int m_secondValue;
};

// Not OK. Could be mistaken for a variable name.
class foo
{
public:
    Foo(int p_value);

private:
    int m_value;
};
```

To avoid confusion, respect the following order for member access when declaring an class interface:

1. `public`;
2. `protected`;

3. `private`.

Sometimes, this rule is hard to follow (for example with private `typedefs` that need to be know from the beginning by the interface). When it is not possible to follow this rule, duplicate as few access keyword as possible troughout the class interface.

```
// OK, everything is in order
class Foo
{
public:

    Foo();
    Foo(int p_value);

protected:
    void writeOverData(p_newValue);

private:
    int m_value;
};


// Not OK: private comes before public. Furthermore, it is useless here.
class Foo
{
private:

    int m_value;

public:

    Foo(int p_value);
};


// Not OK: public sections should be merged at the top.
class Foo
{
public:

    Foo(int p_value);

protected:
    void writeOverData(p_newValue);

private:
    int m_value;

public:
    Foo();
};
```

Interface classes names follow the same pattern as regular classes, but an 'I' character is added before the name. The pattern is IClass:

```
class IEnforceSuchAThing
{
    void such() = 0;
    int thing(char p_aChar) = 0;
};
```

Enumerations follow the same naming convention as regular classes. The actual enumeration can take place on one single line, but can also span on several lines if the layout seems clearer:

```
// OK
enum class Color : int {RED, GREEN, BLUE};

// Also OK. Often clearer.
enum class Color : int
{
    RED,
    GREEN,
    BLUE
};
```

A class that is tested using another class (for example by creating a test fixture in Google Tests) should always hold the original class name followed by the string Tests, clearly identifying its purpose. The format is ClassTests:

```
class FooTests
{
    // ...
}
```

2.2.5 Files

A class should be interfaced in an eponym header file and implemented in an eponym source file. The class should be tested in a source file whose name begins with test_ and ends with the class name. In other words:

Item	Name format
class	Foo
header file	Foo.h
source file	Foo.cpp
test file	test_Foo.cpp

```cpp
// ---------------------------------------------
// Foo.h: Foo class interface.
// ---------------------------------------------
class Foo
{
public:

    Foo();
    void aMethod();

private:

    int m_attribute;
};


// ---------------------------------------------
// Foo.cpp: Foo class method implementations.
// ---------------------------------------------
Foo::Foo() : m_attribute{0}
{
    // Other potential work here...
}

void Foo::aMethod()
{
    // Some stuff...
}


// ---------------------------------------------
// test_Foo.cpp: All Foo's unit tests.
// ---------------------------------------------
TEST(Foo, SomeTest)
{
    // Test here...
}
```

The only file extensions accepted are `.h` and `.cpp` for uniformity. Furthermore, these extensions are supported by nearly all (if not all!) compilers.

2.2.6 Namespaces

Namespace names should always be short. They are candidate to be typed often by the programmer and should not be a burden. For example, the Standard Template Library uses the namespace `std` for its members. Furthermore, it is good practice to use a `#define` to hide the namespace name. This way, if the namespace name changes (idealy, it should not), the program can sucessfully recompile with almost no work. For example, this is a set of `#define`s used for the Standard Template Library:

```
#define BEGIN_STD_NAMESPACE  namespace std {
#define END_STD_NAMESPACE    } // namespace std
#define USING_NAMESPACE_STD  using namespace std;
#define STD                  std
```

You can then use namespaces in the following manner in your code:

```
#include <iostream>

// No tricks:
std::cout << "No tricks: the scope must be added."; // or
STD::cout << "We can add it with a MACRO as well.";

// With using directives:
using namespace std;
cout << "Using directive"; // or

USING_NAMESPACE_STD
cout << "Using directive hidden in a MACRO";

// and so on...
```

The idea is to be consistent at least within a same module, and ideally within the whole code base.

2.2.7 Summary

The table 2.1 summarizes all naming conventions introduced in this section.

Item	**Format**
variable	`camelCase`
constant	`UPPERCASE_CONSTANT`
parameter	`p_camelCase`
member variable	`m_camelCase`
function & method	`camelCase()`
MACRO (w/o param)	`UPPERCASE_MACRO()`
MACRO (param)	`UPPERCASE_MACRO(__aParameter__)`
class	`CamelCase`
structure	`CamelCase`
enumeration	`CamelCase`
interface	`ICamelCase`
files	`[test_]CamelCase[.h/.cpp]`

Table 2.1 – Summary of the naming conventions

2.3 Spacing

The only spacing rule is to respect a **150 characters limit** for every line of code.

Chapter 3

Variables, classes and structures

3.1 `class` vs `struct`

A `struct` should be defined only when the data members that are considered form a data aglomeration without specific behaviour. For example, the following data collection is only a way to structure two `double`s but follow no specific behaviour and should be defined inside a `struct`:

```
struct Point2D // Notice how precise the struct name is!
{
    double m_abscissa;
    double m_ordinate;
};

typedef struct Point2D Point2D;
```

In the other hand, the following data agglomeration holds a specific behaviour – a way to calculate its length:

```
class Line2D
{
public:

    Line2D(Point2D p_firstVertex, Point2D p_secondVertex);
    double length() const;

private:

    Point2D m_firstVertex;
    Point2D m_secondVertex;
};
```

and therefore should be defined inside a `class`. Note that `struct`s can sometimes hold constructors and methods. Usually, the content of a `struct` should be completely public.

3.2 Access qualifiers

When thinking about access qualifiers, the principle of least privilege should be applied. Also, make sure the public part of your class is as stable as possible, especially if it is to be included in a public API!

3.3 Inheritance

Single inheritance is encouraged, multiple inheritance is discouraged other than for interface implementation. Consider the following classes and interfaces:

```
class Foo
{
    // Implementation... (abstract)
};

class Waldo
{
    // Some implementation... (abstract)
};

class IBar
{
    // Interface... (abstract)
};

class IBaz
{
    // Interface... (abstract)
};
```

The `Foo` and `Waldo` classes are regular classes and should be inherited one at a time (even if the class is abstract, but contains some implementation–that is, it is not an interface). In the other hand, both classes `IBar` and `IBaz` are interfaces an can be inherited simultaneously. For example:

```
// OK, single inheritance:
class Qux : public Foo
{
    // Implementation...
};

// Not OK, multiple inheritance:
class Quux : public Foo, public Waldo
{
    // Implementation...
};

// Not OK, multiple inheritance (only one interface):
class Garply : public Foo, public IBar
{
```

```
    // Implementation...
};

// OK, multiple inheritance from interfaces:
class Quux : public IBar, public IBaz
{
    // Implementation (or nor)...
};
```

Unless you really know what you are doing, all inheritance should be public.

Chapter 4

Control structures

4.1 Prefer iterators

Iterators in C++ provide a way to generalize a looping process to all `std`-like data containers. No random access operator is needed to perform the loop. For example:

```
std::string myString{"some text..."};

// Could be better...
for(int i = 0; i < myString.size(); ++i)
{
    std::cout << myString[i];
}

// Better...
string::iterator it;

for(it = myString.begin(); it != myString.end(); ++it)
{
    cout << *it;
}

// Awesome!
for(auto& letter : myString)
{
    cout << letter;
}
```

In the above example, the `std::string` could be replaced by an `std::list`, even if no random access operator is defined for it. Iterators should always be preffered to simple `for`-loops. They are welcomed allied for efficient refactoring!

4.2 Use range `for` loops

Whenever possible, prefer the range-`for`-loop. It is one of the best self documenting tool available in C++11. However, be careful with the keyword `auto` not to generate unwanted copies of objects (especially for large ones!). Consider the following example:

```cpp
// Creates a copy for each novel...
for(auto novel : library)
{
    novel.format();
}

// No copy.
for(auto& novel : library)
{
    novel.format();
}
```

The second example produces the expected result and is cheaper for no copy is done. The first loop is just wrong since the `format()` method is only applied to the copy!

4.3 Nesting is evil

Whenever you can, try to avoid loop nesting and even logic nesting. Instead, use more functions or `std::algorithms`. Nested structures are often hard to follow, allow more maistakes to be made and are poorly self documented.

4.4 Try to avoid loops

If you can, use an `std::algorithm` instead, especially if it makes reading the code clearer. Remember that algorithms are well tested allies!

Chapter 5

Functions, methods and MACROs

The following sections apply both to methods and functions.

5.1 Parameter passing

Parameters, when they represent objects (custom or not) should always be passed as constant references to avoid uneccessary copying. In other words, a parameter named `p_aParameter` of type `Object` should be passed as:

`const Object& p_aParameter`.

If the parameter is of a fundamental data type and is not an object (`int`, `bool`, `double`, etc), this rule does not apply since copy is cheap. For example:

```
// No need for references: default types
int max(int p_firstNumber, int p_secondNumber)
{
    int maximum{0};

    if(p_firstNumber < p_secondNumber)
    {
        maximum = p_secondNumber;
    }
    else
    {
        maximum = p_firstNumber;
    }

    return maximum;
}

// References needed: objects
std::string makeFullName(const std::string& p_firstName,
                         const std::string& p_lastName)
{
```

```
    return p_firstName + " " + p_lastName;
}
```

5.2 const correctness

`const` correctness is perhaps one of the most important rules introduced in this document. If a method does not modify the object it acts upon, mark it as `const`. Consider the following example:

```
// Person.h
class Person
{

public:

    Person(const std::string& p_name) : m_name{p_name}
    {
        //...
    }

    // code...
    bool checkName(const std::string& p_aName) const

private:

    std::string m_name;

};

// Person.cpp
bool checkName(const std::string& p_aName) const
{
    return m_name = p_aName; // Oops: forgotten "="! Compilation fails
}
```

which shows how `const` correctness helps avoid tricky mistakes by running checks at compile time!

5.3 Other specifiers

The `delete` keyword should be added to methods which should not exist in a class (usually because they don't make sense in the context of the class). For example, in the class `Person` below, the programmer has decided that instanciating am object without specifying a name should not be done. Therefore, he deleted the default constructor:

```
class Person
{

public:

```

```cpp
    Person() = delete;

    Person(const std::string& p_name) : m_name{p_name}
    {
        //...
    }

private:

    std::string m_name;

};
```

The `override` keyword is also very important. The following example shows how it can be used to avoid errors using the compiler:

```cpp
class Base
{

public:

    virtual int foo() const
    {
        // ...
    }
};

class Derived : public Base
{

public:

    virtual int foo() override   // Oops! No const, so compilation fails...
    {
       // ...
    }

};
```

In this case, the `override` addition makes the compiler complain that the method with the signature

`int foo();`

does not exist in the base class. This tells the programmer that the `const` keyword is absent from the signature instead of generating a "new" non `const foo()` method (Notice in the example how the keyword `virtual` is repeated in the overriden method signature. This is good practice!).

These two keywords should always be used, when applicable.

Other keywords such as `default` and `final` can be used but are not mandatory.

5.4 Virtual destructors

Destructor should always be made `virtual` to ensure correct behaviour if the class is eventually derived. This rule does not apply if a class is made `final`, of course.

5.5 Inlining

One line methods should be inlined to ensure correct optimization at compile time. If a method has more than one line, do not inline! Example:

```
class Person
{

public:

    Person(const std::string& p_name) : m_name{p_name}
    {
        //...
    }

    // code...

    std::string tellName() const {return m_name;} // preferred

private:

    std::string m_name;

};
```

Prefer `inline` to MACROs, whenever possible. This avoid tricky copy and paste errors which may be hard to find:

```
// Fails for max(a++, b++): a++ or b++ returned.
#define max(a,b) ((a<b)?b:a)

// Better: type checking, no copy/paste behaviour.
inline int max(int a, int b) { return ((a < b) ? b : a); }
```

Chapter 6

Documenting the code

6.1 Documentation tools

Connect X uses Doxygen to document APIs. Some madatory pieces of documentation have been designed to ensure complete documentation for the users (and the developpers!).

6.2 Public API mandatory documentation

All files should start with:

```
/***************************************************************************//**
 * @file    File name with extension
 * @author  John Doe
 * @date    Month YYYY
 * @version X.Y
 *
 * A description of the file contents.
 *
 ******************************************************************************/
```

All methods and functions should start with:

```
/***************************************************************************//**
 * Brief description of what the funtion does.
 *
 * Detailed description of what the function does, if necessary.
 *
 * @param[in] p_aParameter  Parameter descripton
 *
 * @pre     Precondition description
 * @post    Postcondition description
 *
 * @return  What the function returns.
 *
 * @todo    Task description.
```

```
 *
 ******************************************************************************/
```

Some private methods or functions may not be documented but it is good practice to do so. Doxygen will ignore this documentation, but it may help fellow developpers in the future.

All classes and structures should start with:

```
/***************************************************************************//**
 * class ClassName
 *
 * Brief class description.
 *
 * Detailed class description, if necessary.
 *
 * @invariant Class invariant description.
 *
 ******************************************************************************/
```

All attricutes should alwo be documented as follow:

```
int m_anAttribute;  ///< Short data description.
```

Other pieces of code (`namespaces`, `typedefs`, etc) should also be documented using the docygen tags, but a normalized documentation format has not been designed.

6.3 Internal documentation

One should only add comment to internal code (for example to explain a line inside the body of a function) after careful consideration. Usually, this practice is a sign of bad design (but not always) and poor self documenting code. The code should speak for itself. Here are some ideas on how to avoid comments in internal code:

1. separate the code in smaller functions with meaningful names;
2. code what you want to communicate (for example, use `const` to say that your method does not modify an object, instead of writting a comment);
3. use well know functions wherever possible (for example, in the excellent `algorithm` STL header.

For example avoid this:

```
// Useless:
std::string greeting = "salut!";

for(int i = 0; i < greeting.size(); ++i)
{
    cout << greeting[i]; // Print letters, one by one...
}
```

and write this instead:

```
// Better: self documenting code.
for(auto& letter : greeting)
{
    cout << letter;
}
```

Chapter 7

Testing

Testing the code is one of the most important part of software developpement. As the codebase grows, tests are added. At some point, the tests collection becomes huge and navigating through it a real pain, endangering its further use and maintenance. If it starts adding too musch overhead to the developper's work, there is always the rish that he (the developper) starts neglecting it.

7.1 Unit testing

Unit testing aims to test one indivisible entity (a unit). By definition, it is easy to know exactly what they do and this knowledge needs to be reflected in the name. The rule is that unit test names need to always have three parts:

Method name The name of the method (or function) that is tested.

State under test The state of the testes unit, at the moment it is tested.

Expected behaviour The way the unit is expected to react, or behave when tested in these conditions.

These Three fields should be concatenated using underscores, like so:

`MethodName_StateUnderTest_ExpectedBehaviour`

For examples, test names could look like:

1. `IsAdult_AgeLessThan18_ReturnFalse`
2. `WithdrawMoney_InvalidAccount_ExceptionThrown`
3. `AdmitStudent_MissingMandatoryFields_ReturnFailToAdmit`

Regarding methods `isAdult()`, `withdrawMoney()` and `admitStudent()`. With such a naming convention and a clear test body, unit test should not need a lot of documentation. Try to keep it minimal for there might be thousands of tests.