Document Number: N0019
Date: 2008-04-07
Project: RPJBEYE
Reply-to: Manfred Doudar <manfred.doudar@rsise.anu.edu.au>

BIONIC-EYE

CODING STYLE, STANDARDS, GUIDELINES & PROCEDURES

Prepared By: Manfred Doudar
NICTA – Canberra Research Lab

Change History since N0016

- Added requirement that users always set `std::ios::sync_with_stdio(false);`

  REASON: Synchronization with C I/O streams slows down C++ I/O streams considerably. Turning off synchronization should provide C++ I/O performance on par with C I/O.

1 About this Coding Standard

The purpose of this standard is to make code developed under Project Bionic-Eye clean, consistent, and easy to install. This document can be read as a guide to writing portable, robust, and reliable programs; and is based upon GNU coding guidelines.

The rules and principles outlined are often stated with reasons of why we write code a certain way.

It is important that you stick to the conventions and rules laid herein. That way, your program will be more maintainable by others. If this sounds like an option, it is NOT, but rather strongly worded advice. Under code review, you shall be asked to rework your code and submit again if you fail to meet these guidelines.

Please remember, unmaintainable/ non-conforming code is equally as bad as non-working or buggy software. It costs your peers valuable time and effort they could better spend elsewhere. In short, one of the hallmarks of quality infrastructure is that you will not be able to point at any one unit and say who it was that wrote it.

2 LANGUAGE

For the purposes of this project, no language other than C++ shall be accepted. If you think C is a subset for inclusion, the answer is plainly, No! it is not.

If you need help writing C++ see me, or drop me a line by email: manfred.doudar@rsise.anu.edu.au

3 THIRD PARTY LIBRARIES

As is the case with language, our choice of libraries greatly impacts on the quality of project outcomes, our abilities to consolidate effort, and our means to communicate and contribute.

As a rule, you should not use a library or package without consultation. If a particular package meets a need that no other you can think of does AND is not in the list of permissible libraries, bring the issue to open group discussion first!

Inclusion or linking of third party libraries will not be accepted without appropriate review first. The reason for this is three-fold:

- We want to minimize dependencies and incompatibilities.
- We may indeed already have a reviewed library that does achieve the same result.
- We want to ensure that we only employ quality oriented, robust, mature, and platform-independent libraries into the project. –Otherwise we are just asking for much pain and trouble

down the road.

The libraries listed below are staples for the project Bionic-Eye. If you find a solution in two or more of the said libraries, you use the solution found in the library higher up in the list (you choose top-to-bottom):

1. In-house Bionic-Eye Libraries
2. C++ BOOST Libraries
3. C++ STL
4. uBLAS
5. Blitz++
6. NETLIB

OpenCV is shall be our candidate computer vision library, and wherever possible, shall be integrated with C++ BOOST Libraries, in particular GIL (Adobe's Generic Image Library), and wrapped in modern C++.

The libraries already developed in-house have precedence over all others, however this is limited only to those libraries developed under the guise of Project Bionic-Eye, and have undergone review (see next section).

4 REVIEWS & REPOSITORIES

Before any one module is committed to the code repository, it must satisfy four criteria:

- The code has undergone a scheduled code review, and passed against acceptance criteria; namely against design, and this standards document. Otherwise, the author will have to revise against criticisms brought forward, and submit for review again.

- The code must be submitted with full unit-tests, without which, code shall not be accepted.

- The code must come with sufficient in-line documentation. This being said however, every function, no matter how trivial, MUST also be accompanied with a comment after its opening brace stating its purpose.

- Submitted code must strictly remain platform independent, and adhere to the latest draft of the ANSI/ ISO C++ Standard.

5 UNIT TESTS

Unit Tests are not a hindrance, but rather, an enabling technology. We all make mistakes, with our code, and not to mention, the code of others. If we make a change, we want to ensure we have not

broken any other part of system and its dependencies. Unit tests go a long way in building confidence in the code we write, maintain, and adopt.

- Our Unit-Test framework by decree is CppUnit.
- All code written must be supported by unit tests unfailingly.

6 COMPILER & PLATFORM

While we are only advocating standards-compliant, platform-independent code, the official compiler and platform for the Project Bionic-Eye is the Linux platform, and the GNU GCC (g++) compiler.

7 BUILD ENVIRONMENT

As a means to uniformly communicate code, and assist all developers in producing code to standard, Project Bionic-Eye shall employ a unified build environment. This environment will be the means by which you build and release code, and additionally support with unit-tests.

8 CODING STANDARDS

01. Copyright notice to appear atop of each source file: **`Copyright (C) YYYY <Name>, NICTA Ltd.`**

02. File version information, akin to SCCS version control is strictly prohibited.

    Reason: This does not play well with other version control tools, and is unmaintainable clutter.

03. All code is to be compiled with the compiler flags: `-Wall -Werror -ansi -pedantic`

    Reason: We must build cleanly. Warnings will not be tolerated. Furthermore, we must always ensure our code is ANSI conforming, and we are not using any compiler specific extensions. We must mandate robust code, we have no other option.

04. Your directory organization must be structured in such a way as to produce one and only one executable per directory.

05. Librify your code wherever possible.

06. C++ translation units should have a **.cc** file extension, and header files a **.h** extension.

07. Each header file should contain the declaration of one and only one *class-type*.

08. File names should bear the same name as the class-types they declare/ define, all the while spelt in lower case.

09. There is a 178 column source limit. [I find `` `Monospace 9-point font' `` works well].

    Reason #1: For those of you still coding to 80 character limits, understand that computing has moved on considerably. Today you also have more screen real-estate – use it! Not to mention, the higher column width significantly aides readability.

    Reason #2: When writing template code, particularly template meta-programs, 178 columns is reasonable, without which code is near unreadable (and yes, template code can span many many lines of 178 column lengths)!

10. Your header-file guards should STRICTLY take the form: `NICTA_<NAMESPACE>_<FILENAME>_H`

    Reason: By prefixing with the name NICTA, we avoid potential name clashes with 3rd party libraries.

11. Always put C++ template definitions in the SAME header file as the their declarations, and segregate (non-inline) definitions and declarations like so:

    .. declarations

    `// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>`

    .. definitions

    `// <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<`

    that is to say, brace the beginning of definitions with **`// >>>`** and their end with **`// <<<`** where these "braces" should extend to the maximal column length permitted (178 column width).

    Reason #1: These "braces" clearly disambiguate where declarations and definitions begin and end in the one file.

    Reason #2: Template definitions and declarations in one file is more or less standard across most C++ communities, though some are known to split off template implementation into a separate unit (usually with a .txx or .tcc extension); though we will opt not do this.

12. All header file and translation unit `#include`*'s* should be listed in ALPHABETICAL order, and in the following prescribed order:

    [a.] project-level `#include`*'s*, first
    [b.] NICTA in-house library `#include`*'s*, second
    [c.] Third-party library `#include`*'s*, third
    [d.] System-level `#include`*'s*, last

    Reason #1: Alphabetical order aides readability.

    Reason #2: Includes can commonly span 20 or more lines, and searching for dependencies becomes difficult, unless there is a mandated order.

13. Do not use C-style headers. Use the C++ headers instead.

14. DO NOT `typedef` ATOMIC TYPES! Composited types are Ok to `typedef` (if deemed necessary).

    Reason: `typedef`'ing atomic types leads to code obfuscation, and indicates that you are trying to express a design choice, but failing miserably – use SFINAE instead and enable_if templates from C++ Boost.

15. All contributed NICTA code should live in the top-level namespace `nicta::`

    Reason: Avoids potential name-clashes with 3rd party libraries.

16. Under NO CIRCUMSTANCES should you use `using`-*declarations* inside of header files. They are equally forbidden inside of translation units too. This means you must be explicit as to which namespace a function, member, or type is lifted from.

    Reason #1: using-declarations subvert the purpose for having namespaces in the first instance, and leads to namespace pollution.

    Reason #2: being explicit about origins makes the code familiar with minimal searching.

17. All names: *functions, class-types, variables* should be in lower-case, and separated by underscores. Note: enumerations have a different convention (see later).

```
void free_function();

- NOT -

void FreeFunction();          // wrong

- NOT -

void Free_Function();         // wrong
```

    Reason: C++ language style, and witnessed by the STL is evidence to this.

18. If something comes from the global namespace, be sure to indicate that with the scope resolution operator.

```
void free_function()
{
```

```
    // blah ...
}

int main(int argc, char** argv)
{
    ::free_function();        // scope resolution op indicates global scope

    return 0;
}
```

Reason: being explicit about origins makes the code familiar and readable.

19. Do NOT use assertions. Use `try-catch` handlers and exception specifications, or alternatively, advanced template idioms.

    Reason #1: By using assertions, you force an outcome on your users, but your users should always be given the choice to handle an error for themselves. You are only obliged to notify them of the error.

    Reason #2: Assertions communicate nothing to the user of the error, and nor do they provide a core dump for debugging.

20. Throw exceptions by value, and catch them by (usually `const`) reference.

    Reason #1: You need to throw exceptions by value because the exception object will be destructed when it goes out of scope (at the end of the basic block where it is caught – unless you re-throw the original exception). If you throw a pointer to an allocated object, it will not be deleted, and you will leak memory!

    Reason #2: You should catch an exception by reference to avoid *slicing* – after all, exception classes are inherited from `std::exception`. Furthermore, catching by value will not catch the exception object itself, but a copy!

    Reason #3: Catching by `const` reference aides compiler optimization, unless you will be calling methods on the exception that are not declared `const`.

21. Do NOT use `pragma`.

    Reason: Not portable.

22. Do NOT use the `export` specification.

    Reason: Not portable.

23. Unless you are using the C++ BOOST Preprocessor Library, MACROS ARE STRICTLY FORBIDDEN, no matter how trivial your macro appears (even simple debuging macros)!

    Reason #1: Generally, macros are unmaintainable, poorly used, badly documented, and obfuscated.

    Reason #2: They cause much pain and difficulty to debug.

    Reason #3: In the author's decade plus experience, only two hand-rolled macros were ever written, and in retrospect, today the problem they sought to solve could have been achieved by better designed software.

24. Unless you are writing code to overcome portability issues for different platforms, or different compiler version, then pre-conditionals, aka `#ifdef`'s and their ilk are STRICTLY FORBIDDEN!

25. Unless you need a comment in preprocessor code, only use the C++ style comment: **`//`** and every comment to be separated by a space and the marker.

```
// this is a good comment          // OK

//this is a bad comment            // wrong
```

26. Indentation: 4 spaces is mandated. TABS ARE STRICTLY FORBIDDEN!

27. DO NOT use `#define`'s to declare constants! Use `const` instead.

    Reason: It is C++ style.

28. Global variables are STRICTLY FORBIDDEN!

29. Facilitate reentrant code - avoid static variables.

30. You should specifically list EVERY header dependency of a file, and bring it in with an `#include`

statement. Under NO circumstances should you omit a dependency because it comes through from another such dependency – you have been warned!

31. Avoid unnecessary build dependencies and long compile times by not including a header file when a forward declaration will suffice.

```
class foo;              // OK:  forward declaration
class ecky;             // OK:  forward declaration

struct bar
{
    bar(const foo& instance);

    ecky* ptr_;
};
```

- NOT -

```
#include "foo.h"        // wrong
#include "ecky.h"       // wrong

struct bar
{
    bar(const foo& instance);

    ecky* ptr_;
};
```

32. Every function, no matter how trivial must have a comment that details the purpose of that function immediately after the first opening brace of the definition. Comments must start with a capital letter, and should NEVER appear at the top of a function signature (either definition or declaration).

```
void classname :: ~classname() throw()
{
    // Destructor

}
```

33. Out-of-line function definitions of *class-type* should ALWAYS have a space either side of the scope resolution operator.

```
void classname :: foo()
{
```

- NOT -

```
void classname::foo()         // wrong
{
```

Reason: Makes searching for the beginning of a definition effortless.

34. If a member function of *class-type* does not modify class member data, declare the function `const`.

```
RT foo() const;
```

35. If you do not intend for a function parameter not passed by value, to be modified, declare the parameter `const`.

36. Unless passing through an atomic type, pass parameters by pointer or reference.

37. Pass non-atomic types by reference unless the object passed through is not guaranteed to exist; then and only then pass the parameter by pointer.

Reason: References are safer, and do not incur the overhead of checking for a valid pointer.

38. Know when to return by value, reference, and pointer.

39. Always call static members using full scope resolution.

```
classname::spod_;       // OK:  spod_ is a static member variable

classname::foo();       // OK:  foo() is a static member function
```

Reason: Clarity. Tells us that we are invoking a static member.

40. Be explicit with non-static members, use: `this->`

Reason #1: Koenig-lookup.

Reason #2: Allows us to pick up unconstrained types in class template hierarchies.

Reason #3: Allows us to immediately identify class members.

41. Do NOT specify member variables with access specifier *public*.

42. DO **postfix** all *private* & *protected* member data **AND** functions with a SINGLE underscore _

```
this->spod_;      // OK:  spod_ is either a private or protected member

this->foo_();     // OK:  foo_() is either a private or protected member

ecky;             // is a local variable

this->twang       // is a public member

this->bar();      // is a public member
```

Reason: Facilitates clarity & readability.

43. Be explicit regarding scope a member is called from.

```
class bar
{
public:

    RT foo();

protected:

    int data_;
};

class ecky : public bar
{
    RT spod() const
    {
        this->bar::foo();          // OK
        this->bar::data_;          // OK

        this->foo();               // wrong
        this->data_;               // wrong
    }
};
```

Reason: Facilitates clarity.

44. DO NOT use double underscores!

Reason: Can cause name clashes with compiler resulting in undefined behavior. Double underscored names are reserved for compiler vendors.

45. DO NOT use **`NULL`**. Use **`0`** to nullify pointers.

    Reason: It is C++ style.

46. When indicating a constrained type template parameter, DO NOT use *class*. Use `typename` instead.

```
template <typename T>          // OK
class bar;

template <typename T>          // OK
RT foo(T t);

- NOT -

template <class T>            // wrong
class bar;

- NOT -

template <class T>            // wrong
RT foo(T t);
```

47. DO NOT use C-style casts.

    Reason: They are not type-safe.

48. If you `dynamic_cast`, your design is probably wrong – re-think it!

49. Nine times out of ten, if you `const_cast`, you are hacking at code. Avoid it!

50. Avoid the `reinterpret_cast`. It leads to portability issues!

51. Braces. With the exception of namespaces, should **ALWAYS** be on a separate line by themselves.

```
RT foo()                      // OK
{
    blah;
}

- NOT -
```

```
RT foo() {                    // wrong
    blah;
}

- NOT -

RT foo() { blah; }            // wrong

- NOT -

RT foo()                      // wrong
    {
        blah;
    }
```

52. Namespaces. First brace on same line as name, last brace on a separate line (with multiple terminating braces on same line), followed by comment of the namespace scope that it terminates.

```
namespace ecky {          // OK

    blah;

} // namespace ecky


namespace foo {           // OK
namespace spod {

    blah;

} } // namespace foo::spod
```

53. One line statements of a condition or loop construct **MUST** be enclosed by braces.

```
if (condition)            // OK
{
    ++spod;
}
else
{
    --spod;
}

while (condition)         // OK
{
    ++spod;
}

for (;;)                  // OK
{
```

```
    ++ecky;
}
```

- NOT -

```
if (condition)          // wrong
    ++spod;
```

- NOT -

```
for (;;) ++ecky;        // wrong
```

Reason: Aides readability.

54. Always have a space between parentheses and a control-flow keyword.

```
if (condition)          // OK

while (condition)       // OK

for (;;)                // OK
```

- NOT -

```
if(condition)           // wrong
```

- NOT -

```
while(condition)        // wrong
```

- NOT -

```
for(;;)                 // wrong
```

Reason: Aides readability.

55. Always follow through with a space after a semi-colon in for-loop parameters.

```
for (int i = 0; i != 10; ++i)       // OK

for (int i = 0;i != 10;++i)         // wrong
```

56. Always separate expressions with spaces at either side of ALL operators:

```
= != < > <= >= + - / * % == && || & | ^ ^= += -=
```
... etc (and so on)

```
a += 2;                              // OK

b = (4 + a / 3) * (c % 7);           // OK

if (a && b)                          // OK

- NOT -

b=(4+a/3)*(c%7);                    // wrong
```

57. Always declare for-loop variables in for-loop scope.

```
for (int i = 0; i != 10; ++i)        // OK

int i;
for (i = 0; i != 10; ++i)            // wrong
```

58. Use prefix operators ++ and -- over postfix operators wherever possible.

    Reason: Postfix operators usually incur a copy overhead.

59. ALWAYS declare variable at the place of first use, NOT many lines before!

    Reason: Aides readability & maintenance.

60. DO NOT use `void` for a parameter type, when the function takes no arguments. Just have nothing.

```
RT foo();                // OK

RT foo(void);            // wrong
```

61. Function names and parentheses do NOT have space between them, and NO space on the inside of the parentheses.

```
RT foo();                // OK

RT foo(int a);           // OK

- NOT -

RT foo ();               // wrong

RT foo( );               // wrong
```

```
RT foo( int a );        // wrong
```

62. Pointers and references.

```
char* p = "flop"        // OK

char& c = *p            // OK

- NOT -

char *p = "flop"        // wrong

char &c = *p            // wrong
```

Reason: In C++, definitions are mixed with executable code. Here p is being initialized, not *p. This is near-universal practice among C++ programmers; it is normal for C hackers to switch spontaneously as they gain experience.

63. Class-type access-specifiers. Always align with the brace, and follow with empty line below.

```
class foo               // OK
{
public:

    // Types


private:

    // Types
};


- NOT -


class foo               // wrong
{
public:
    // Types
};


- NOT -


class foo               // wrong
{
    public:

        // Types
```

```
};
```

64. Enumerations. One line per enumerator, and names in camel case..

```
enum BlahWhat           // OK
{
    Foo       = 10 ,
    SpodTwang = 11 ,
    Ecky      = 12
};


- NOT -

enum blah_what { Foo = 10, SpodTwang = 11, Ecky = 12 };          // wrong
```

65. DO NOT inject anything into namespace `std::`

66. Member initialization lists. DO NOT use assignment in constructors (includes copy constructor).

```
ecky :: ecky()          // OK
: datum_(0)
, more_stuff_(-9)
, helper_(0)
{
    // Constructor
}

ecky :: ecky() : datum_(0), more_stuff_(-9), helper_(0)          // wrong
{
    Constructor
}

ecky :: ecky()          // wrong
{
    // Constructor

    this->datum_      = 0;
    this->more_stuff_ = -9;
    this->helper_     = 0;
}
```

Reason: Safe initialization, performance.

67. ALWAYS initialize your class-type member variables in the order in which they are declared.

Reason: Avoid unintended side-effects. The compiler will always initialize your types in the

order in which they were declared. However, if you use a variable thinking it already initialized because of an order of initialization you prescribed in the initialization list, then you will get a rude shock by actually ending up using an uninitialized variable!

68. Align your assignments for readability.

```
datum       = 0;        // OK
more_stuff  = 4;
something   = -7;

datum = 0;              // wrong
more_stuff = 4;
something = -7;
```

69. Spacing with respect to `return` statements. No extra spacing before returns, and no parentheses.

```
}                       // OK
return blah;

- NOT -

}                       // wrong

return blah;

- NOT -

}                       // wrong
return (blah);
```

70. DO NOT name a constrained template parameter by type. Use templates properly to generate compile time errors if a template is instantiated with an unintended type.

```
template <typename TReal>          // wrong, wrong and wrong!!
struct foo
{
    // blah ..
};
```

71. DO NOT derive from STL containers with the intention of using them as polymorphic base-classes.

Reason: STL containers do not declare their destructors `virtual`.

72. DO NOT override the default value of a function parameter in a polymorphic base-class.

    Reason: While virtual functions are dynamically bound, default parameter values are statically bound.

73. Keywords such as `template, extern, static, explicit, inline`, etc should go on the line above the function name.

```
virtual                                 // OK
int foo();

template <typename T>                   // OK
class foo;


- NOT -

virtual int foo();                      // wrong

- NOT -

template <typename T> class foo;        // wrong
```

74. Functions with more than two parameters should list their parameters on separate lines.

```
void foo(int var1, int var2);       // OK

void foo(int var1,                  // OK
         int var2,
         int var3);

void foo(int var1,                  // OK
         int var2,
         int var3
        ) const throw ();


void foo(int var1, int var2, int var3);                     // wrong

void foo(int var1, int var2, int var3) const throw ();      // wrong
```

75. Template parameters with more than two types should be listed on separate lines, lining up commas with angled brackets.

```
template <typename T, typename U>       // OK
class foo;

template < typename T                   // OK
```

```
          , typename U
          , typename V
          >
class foo;
```

- NOT -

```
template <typename T, typename U, typename V>        // wrong
class foo;
```

- NOT -

```
template <typename T,                                // wrong
          typename U,
          typename V>
class foo;
```

- NOT -

```
template <typename T,                                // wrong
          typename U,
          typename V
         > class foo;
```

- NOT -

```
template<typename T                                  // wrong
        ,typename U
        ,typename V
        >
class foo;
```

- NOT -

```
template<typename T,                                 // wrong
         typename U,
         typename V
        >
class foo;
```

76. DO NOT call non-static member functions in constructor initializer lists.

    Reason: Your object still has not been fully constructed yet.

77. DO NOT, repeat DO NOT return a raw pointer from a function! Use a BOOST smart pointer instead.

    Reason: A user of your code will not know who is responsible for deallocating the pointer.

78. ALWAYS ensure you compare your parameter to `*this` in your assignment operator.

79. If you want to prevent assignment, declare (not define!) the assignment operator `private`.

80. If you want to prevent copies, declare (not define!) BOTH copy constructor and assignment operator `private`.

81. DO NOT use `std::auto_ptr`. Use one of the smart pointers in BOOST.

    Reason: `std::auto_ptr` has transfer of ownership semantics, and consequently VERY dangerous.

82. Users must ALWAYS set

```
std::ios::sync_with_stdio(false);
```

    Reason: Synchronization with C I/O streams slows down C++ I/O streams consideratbly. Turning off synchronization should provide C++ I/O performance on par with C I/O (depends on QoI of your compiler as to the magnitude). This advice is further substantiated by the fact that we do not use C, and consequently are to avoid C I/O streams.

83. Facilitate *Return Value Optimization* (RVO).

```
ClassType foo()                                    // OK
{
    return ClassType(arg1, arg2);
}

- NOT -

ClassType foo()                                    // wrong
{
    ClassType instance(arg1, arg2);
    return instance;
}
```

    Reason: Performance.

84. Declare class-type member functions before member variables.

85. Declare your destructor before all other member functions for visibility.

86. Declare class-type static functions before your destructor, and consequently, before all other members too.

87. Wherever possible, manufacture a template solution, NOT a *runtime polymorphic/* `virtual` one. Remember, there is almost always a template solution to every runtime incarnation.

    Reason #1: We avoid the overhead of *virtual function tables* (vtables).

    Reason #2: We gain the security of compile-time (static) type checking.

88. Use *Object-Oriented* paradigms. Prefer *Functional Programming* paradigms over Object Oriented ones. Prefer *Template Meta-Programming* paradigms over Functional ones. Fuse all three paradigms successfully for best quality outcomes.

89. If it is in BOOST, use it!

    Reason #1: We do not want to reinvent the wheel.

    Reason #2: Owing to peer review, it has been done better than anything you are likely to match.

    Reason #3: It is robust, platform independent, and tested.

90. If you do not adhere to this standards document, expect to have your fingers chopped off. You have been warned!