# C++ Coding Standards in $\mathcal{MLC}$++

Ronny Kohavi
Ronnyk@sgi.com

June 18, 1996

## 1 Introduction

*If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization*
*—Gerald Weinberg*

The purpose of this document is to provide coding standards for writing C++ code in $\mathcal{MLC}$++. The description here can be used as a general guideline for programming in C++, independent of $\mathcal{MLC}$++, but it is a low-level guide that does not discuss important issues of design. The $\mathcal{MLC}$++ coding standards defines defines higher-level concepts used in $\mathcal{MLC}$++, including error-handling, defensive programming, and testing.

The conventions established here are designed to make the code more readable and more reliable. Every rule has exceptions, but deviations from the standards should be documented in the code and explained.

This document used to be longer, but it got shorter because of Meyer's book (Meyers 1994). Please read this book ASAP. His second book (Meyers 1996) is less basic and not as important initially (although still highly recommended).

[Annotations are enclosed in brackets and printed in a small font. They elaborate the actual text, giving reasons and explaining some of the decisions.]

## 2 Motivation and Overview

*Efficiency still matters, but it becomes secondary in that it is irrelevant unless the larger systems can be economically constructed and maintained*
*—Stroustroup (1994, p. 202)*

*Many programmers routinely ignore compiler warnings. After all, if the problem were really serious, it'd be an error, right?*
*—Meyers (1994, p. 183)*

The compiler should not generate any warnings for the code. Sometimes we know that the code is correct, but it is better to make sure everything is clean.

[We recommend using full warnings and suppressing those warnings that are truly unimportant or annoying. For example, SGI's MIPS7 compilers under `fullwarn` generate remarks that some code may not run well on a specific revision of some CPUs.]

Code should be structured (few *goto*s, *break*s, and *continue* statements), and should avoid hacks and micro-efficiencies (*e.g.*, no register declarations and no inlining of functions over 2-3 lines unless performance analysis show it helps). The idea is to have a *flexible* and an *extensible* code that is easy to maintain. Performance analysis can be done when it is needed, and the critical 10% of the code can be made faster. Make sure to get the algorithm right and leave the tweaks for later to lower the constants.

Classes should be written so that the code can be replaced without changing the interface. Avoid putting data objects as public members, thus exposing the internals.

Use advanced features of C++, such as templates. Non-standard options should not be used because they will make portability harder.

[C++ is an evolving language and many features are not widely available, or there is not enough experience working with them. The most obvious extension we do not recommend using is exception handling. We recommend reading Meyers (1996, Items 9-15) to get a sense of how hard it is to write correct code with exceptions.]

## 2.1 Naming Conventions

Naming conventions basically follow GNU-lib conventions, with some added definitions.

1. Classes and typedefs start with uppercase. Capitalize words in multi-word class (do not use underscores).

2. Instantiations of classes have the same naming conventions as the classes, except that the first letter is lowercase.

3. The underscore is used to separate components of long **function** names.

4. Include file names that define C++ classes begin with uppercase letters. A class should have the header file name identical (or at least very similar) to the class name with a ".h" suffix, and a source file name with a ".c" suffix. Regardless of the actual name, the prefix (without the extension) should be the same for the ".h" and ".c" extensions.

   [Some people use `c++` or `C` as extensions, but we many template instantiators assume that to find the template the `h` in the header file should be replace with `c` and so you must either name header files with a `h++` suffix or with a `H` suffix, both of which are rather ugly.]

5. File names should resemble the class name or function name they define.

6. Classes, types, and functions that are not self describing or that others should use caution when using them should have a trailing underscore.

   [Some people use the convention of leading underscores, but we decided to follow the recommendation in ARM (section 2.4): "Identifiers starting with a single underscore (_) should also be avoided by ordinary users since C implementations reserve those for their own use."]

   [For example, the `AttrValue_`class requires an `AttrInfo` class to parse it, so it is not self-describing.]

7. Macros are uppercase, `#defines` which look and act like types or classes look the same way as classes.

   [Avoid using macros where inline functions suffice, and use typedefs instead of `#defines` whenever possible.]

8. Constants are all uppercase, and should be defined in the beginning of the file. Words should be separated by underscores (since everything is uppercase).

   [Note that in some cases, like where a constant is used to dimension arrays, it cannot be external and must be defined either as "const" or as an enum. Defining something as enum can be done inside a class, which encapsulates it nicely. See Meyers (1994, item 32).]

9. Enumeration values (true ones, not declared for constants inside a class) look like variables.

   [Enumerated values are actually constants for most purposes, but there tend to be a lot of them, and we don't want too much uppercase. Moreover, they are not usually the type of constants that you ever want to look at to see their value.]

10. If you write a binary operator such as `operator==`, use `lhs` and `rhs` for the left-hand side and right-hand side.

    [This is a new convention following the suggestion made in Meyers (1996).]

## 2.2 Classes

*Virtual means magic*
*—Stroustrup (1994, p. 266)*

1. Use of public data members should be avoided. These should be accessed through *accessor* functions. Access functions may give only read access to the members, that is, they return references to const. Giving public read/write access is equivalent to making the data member public, and in those cases it is sometimes better to make the data member public, especially if it is an instantiation of a class.

   [A member that is a class by itself, may be in the public part if it saves duplicating the class interface. This should not be done too often, as many times class operations do not have the correct meaning inside another class. It also means "supporting" all the routines the class supports now or in the future. A better solution is to provide interface functions if not everything needs to be supported.]

2. Classes that have important invariants should have a function member "OK()" (following GNU-lib) which tests the invariants. OK has an optional argument for the level of test to do. Higher levels usually mean faster tests, but there is not necessarily a linear order on simplicity (*e.g.*, level 3 and 4 may do different checks). The default level defined in the header file is the suggested level and should include a very extensive check, but not necessarily the most extensive/costly test (*e.g.*, the default level could be level 2).

   The invariants will be described in the description of the function, and this will be the first function after the class header (before the constructors).

   [Invariants are an important part of the documentation for anyone wanting to add or modify the class, so this should be the first function. Putting the invariant description in the actual class header would distract "users" of the class who just read the header.]

3. Classes should not use the default copy constructor **unless** copying is fast, and it is documented that the default copy constructor is appropriate. The file error.h contains the macro `NO_COPY_CTOR(X)` This should be put in the **private** part of the class. This declares a private copy constructor. Unintentional usage outside the class will give a compile time error (it is private). Unintentional usage in the class will generate a link-time error, as there won't be a definitions for this function.

   If a class needs a copy constructor, it is better to force the caller to be aware of the fact that the copy constructor is being called, instead of automatic code generated by the compiler. This enforcement is done by defining the copy constructor with a second dummy argument of type `CtorDummy` which is an enumerated type with one value `ctorDummy`. Thus, to call the copy constructor, you would typically write `Array<int> a(b,ctorDummy)`.

   [The idea of ctorDummy is similar to the idea in overloading the postfix version of `++`]

   Similar reasons hold for `operator=` with the macro `NO_OPERATOR_EQ(X)` used to declare it but not define it.

4. Symmetric operators like `+` should be member functions, not friend functions.

   [There was a long discussion on the standard for this. Here are some pros and cons.
   The advantage to making `+` a friend function is that it is really a symmetric function. We want `a+b` to work exactly as `b+a`. One example where a member function may fail to do the job is when there is an automatic conversion going on. For example, suppose class X has an automatic conversion from (char *). Given instance x of type X, you can write `x+"Ronny"`, but not `"Ronny"+x`, since there is no `+` for `(char *)` that accepts an instance of X.
   The problem with the friend function can be illustrated as follows. Suppose Y is derived from X, the code `x1+x2` in a routine, where a pointer to Y was passed as x1, should call Y's `+`. If `operator+` is virtual, we get the right effect.
   One suggestion which was rejected is that the friend function will call a virtual function `plus` of the first operand. This will make the automatic conversions work, but the function becomes asymmetric again. Why should the `operator+` of the first operand be called and not the right one? The decision was to have `operator+` asymmetric, and let the programmer know that the left operand determines which virtual function is used.]

5. `operator==` should **not** be virtual, except if the class is known to be such that no class will be derived from it. See Stroustroup (1994, p. 295) for details.

   [The problem with making `operator==` virtual is that a function that does something on two base classes and requires only the base to be similar, does not really need to check equality of other information, and what's more, this may not allow using derived objects as objects. A nice example is AttrCategorizer which checks that the given instance contains attribute information that matches the attribute information given in the construction of the categorizer. Passing a labelled instance to Categorize() will not work if `operator==` is virtual, because the attribute information will not match (the original attribute information does not contain information about the label).
   A class like MString may declare `operator==` as friend because we do not expect any subclassing off String.]

6. Public member functions of classes should usually be virtual. This is doubly true for the destructor. See Meyers (1994, item 14). There are some exceptions listed in Koenig (1992)
   (a) When efficiency is a major concern. This should be determined by profiling, not in advance. Note that you can still declare a function virtual inline and the compiler may be able to inline it.
   (b) When non-virtual functions behave "correctly" (as in `operator=`).
   (c) When the class is not designed for inheritance.
   (d) When the functions are accessors, or delegators (*i.e.*, convenience functions) that are essentially calling another member to perform the action.

7. Destructors for classes must be declared (even if empty), and they should be declared virtual. Exceptions are made where efficiency is needed, or if there are no other virtual functions.

   [The destructors must be declared even if empty, because we want to make them virtual so that the destructor for the derived class will be called.
   Interestingly, if you want to make an abstract base class non-instantiable, you can make the destructor pure virtual, but implement it (you must!).]

8. Do not execute too many things in the constructor. Remember that virtual functions you call in the constructor will not call the derived class's functions because the virtual table is disabled during construction and destruction.

## 2.3 Functions

0.7If your favorite introductory C++ textbook doesn't discuss static members, carefully tear out all its pages and recycle them. Dispose of the book's cover in an environmentally sound manner, and then borrow or buy a better textbookMeyers (1994, p. 102)

1. Use of "const" is encouraged for function arguments. All functions that do not change their arguments should declare them const. All returned values which are pointers to data that should not be changed should be pointers to constants.

   [Declaring atomic types as const is usually useless, thus we do not declare `const int`. ]

2. Member functions which do not update the class or that behave as logically const[1] should be declared const functions. Some operators, most notably operator[] should be provided in two versions, the const and the non-const. The const version returns the const reference to the element, and the non-const returns an lvalue.

[1] The basic idea in logical constness is that for caching, or statistics (*e.g.*, counters), the function actually updates data members, but to the outside caller, it behaves as if it is a constant function.

Note that disambiguation is done by the class itself. It a const class is given, the const function is called, otherwise, the non-const version. There is no overloading on the returned value in C++. If there is a significant difference between the const and non-const version (*e.g.*, when reference counting is done), it should be stated in the headers so that callers can change to a const operation.

3. If a function does not change the argument, but becomes the owner, the argument should be a reference to a pointer, so that we can change the caller's value to NULL (see Section 2.4).
4. Arguments should be self describing (*e.g.*, int[] is bad). They should not rely on other arguments to give their length, size, *etc.*, as this is a sign that the abstraction level is too low (Stroustrup 1991, page 426).
5. Functions that are written to help the implementation of a class should be defined as "static" (their scope is the file scope), or they should be made private members.

   If you find yourself repeating code, this is often a good solution. The idea in using static declarations is to avoid conflicts with other classes that may define the same names. Static functions may be required if you wish to perform some operation at the constructor when giving an initialization list.

6. If a function does not use one of its arguments, do not give it a name in order to avoid a compiler warning. For example,

   ```
   Array(const Array& source, CtorDummy);
   ```

   Note how the second argument is unnamed. Sometimes, it is better to tell what we are ignoring, *i.e.*, what the argument is:

   ```
   void OK(int /* level */)
   ```

## 2.4 Ownership

*If you listen closely when you read this code, you can hear the sound of an airplane crashing and burning, with much weeping and wailing by the programmers who knew it*
*—Meyers (1994, p. 32)*

The owner of some memory is responsible for deallocating it. It is possible to allocate an area and transfer ownership to a different entity which will be in charge of deallocating it. Ownership is recursive to all levels unless specified differently, *i.e.*, ownership of a tree implies ownership of strings pointed to from the nodes.

When a function gets ownership of an object, the function should declare the argument as `ptr*&`. This allows the function to copy the pointer, but also set the caller's pointer to NULL, thus not allowing it to be used any more. This is intended to avoid mistakes more than a safe-proof method, as the caller can copy the pointer in advance.

One annoying case of the above usage is when you pass a derived class to a function that accepts a base class. In such cases you must pass a variable of the right type (base class), because otherwise a temporary is created and a warning is generated.

Ownership is recursive. If you own the array, you own the elements inside the array.

In general, it is a good idea to return a pointer when the pointer returned needs to be deallocated and to return a const reference when you don't give ownership (this fails if you need to return a NULL sometimes).

### 2.5 Reference Counting

*Most problems in Computer Science can be solved with an additional level of indirection*
*—Meyers (1996, p. 208)*

When ownership problems become a mess to handle because a class is ubiquitous or it needs to be copied many times, classes should be reference counted. The main advantage of reference counting is that there are no questions of ownership and that copies are cheap to make (at least until changes to the object need to be made).

The best way to make a reference counted class is to avoid special tricks with smart pointers (some suggested in Meyers (1996)) and to work hard on wrapping the class as shown in Coplien (1992, p. 58). While the initial wrapping around using the handle/body class idiom is harder on the class implementor, it is transparent to users of the class, which is more important.

It is also useful to add `RC` to the class name to make it clear that the class is reference counted. That way the user knows there are no deallocation problems and making copies is very cheap.

The methodology used in $\mathcal{MLC}$++ is to include `RefCount.h>` which defines the standard constructors and functions. Then each class member in the handle class parallels each member of the body class, except that the operation is preceded by `set_rep()` for construction operations, `read_rep()` for read-only operations, and `write_rep()` for write operations.

Some things to note about using reference counted classes:

1. Beware of using `operator[]` on non-const instances. Because of the way overloading works, any non-const function calls write_rep() to create a local copy. For example, the following code

   ```
   for (ILPix pix(*this); pix; ++pix) {
      InstanceRC instance = *pix;
      const AttrValue_& val = instance[attrNum];
      ...
   }
   ```

   will be slow because every instance will be copied even though the `operator[]` is used as read-only. A much better approach is to add a const, *i.e.*,

   ```
      const InstanceRC instance = *pix;
   ```

   this way no copy will be made.

2. A second question with reference counting is the use of references. If you know that the object will persist, there is no need to get a copy and you might be satisfied with a reference. This is especially common when the const object is given as a parameter to a function or used in a loop: there is really no need to pay for updating the counters. Thus the following is faster:

   ```
   for (ILPix pix(*this); pix; ++pix) {
      const InstanceRC& instance = *pix;
      const AttrValue_& val = instance[attrNum];
      ...
   }
   ```

   Of course any member that is stored in a class must not be a reference to a reference counted class. You should invoke the mechanism to make sure it will never be deleted before our class deletes it.

3. Since accesses through reference counted members require another indirection, you may sometimes want to get the actual body class. The function `read_rep())` will return a const pointer to the class. Use of this function should be limited to time-critical situations. Don't abuse this!

Finding leaks with reference counted classes is a bit tricky. Suppose function `f()` creates instance $x$, which is then copied in function `g()` into $y$. Function `f()` correctly deallocates $x$ but function `g()` leaks. Tools such as purify will point to `f()` as the leaking module, since it originally allocated the area, which was never deleted (because `g()` kept the reference count above zero). To help track such leaks, an option is available in $\mathcal{MLC}$++ to allocate one byte on every copy of a reference counted class and delete this on destruction. That way we will will see that `g()` makes a copy and never deallocates it. This option is automatically invoked when the `DEBUGLEVEL is set to 2` in $\mathcal{MLC}$++.

[There are some complex issues if reference counted classes are used in a way that creates cycles. Object $x$ has a reference to $y$ and $y$ has a reference to $x$, so neither is deallocated. One has to watch out that such a thing cannot happen during the design of reference counted classes.]

## 2.6 Be Defensive

*Use the last 'else' of a chain of 'else-ifs' to catch conditions that should "never" occur, but just might*
*—Kernighan & Plauger (1976)*

1. Be defensive when you write code. Do not trust the caller and try to check whatever you can. If a number must be in range, check for that. If an argument is size, check that it is positive. Some of these checks and asserts could be turned off for the final version.
2. Always ASSERT that a pointer is not NULL before dereferencing it. It is better to get an assertion failure than a core dump. The ASSERTs may be taken out using preprocessor defines.
3. Add integrity checks in an OK() function. Remember that tomorrow someone else may modify your class and will forget to carefully read your meticulously-written comments. It is always better to have the integrity checks done by the CPU. Calls to OK() can be removed using preprocessor defines.

## 2.7 Syntax Standards

*You don't need to understand this gobbledygook to use the `string` class, because even though `string` is a typedef for The Template Instantiation from Hell, it behaves as if it were the unassuming non-template class the typedef makes it appear to be*
*—Meyers (1996, p. 279)*

1. Use `@@` as a signal for something temporary or something that needs to be looked on again. When you finish a piece of code, these will pop out quickly and should be reviewed.
2. Lines in the source code should not exceed 79 characters.

   [This will make them printable, and also allow a standard window. Note that for long strings, C++ has an option to concatenate constant strings by closing the quotes and opening them again with no operator in between.]

3. Parentheses should be used when necessary to override precedence. Do not use them like in LISP. `if ((t1) && (v1 == v2))` should be `if (t1 && v1==v2)`. Parentheses should be added to long expressions, especially if they span more than one line, or when mixing ANDs (`&&`) and ORs (`||`).

   [An expression which is too big is usually a bad sign.]

4. Preprocessor macro definitions should enclose the arguments in parentheses when they are used. In general, try to avoid macro definitions whenever possible. Inline functions and templates are safer.

[Writing `#define SQUARE(x) x*x` will give unexpected results if you do `square(a+b)`. A better definition is `#define SQUARE(x) ((x)*(x))` Here you actually want to use many parentheses.]

5. There is no tabbing convention, but if tabs are used they should be the default tabs (8 characters).

   [We "encourage" indentation of 3 characters. Emacs has options to tabify and untabify regions, so this tabbing convention is not too important.]

6. The opening brace of functions should begin on a new line as is the standard in most C and C++ books. Opening braces for `if`s and loops should be on the same line. If there is no brace in an /if/, start a newline anyway.
7. `#include` statements that include files from the inc/ subdirectory should be done with angle brackets (as opposed to quotes).

   [It seems at first that quotes could only help, that is, if you have a local copy it would override the inc/ version, but this is not true due to nested includes. Suppose you include InstanceInfo.h which in turn includes Attribute.h. You have modified Attribute.h, so it is in the current directory, but InstanceInfo.h is not. The compiler finds InstanceInfo.h in the inc/ subdirectory, and when it sees `#include "Attribute.h"`, it also takes it from the inc/ directory which is a mistake since you have a local copy. The solution is that the -I flag is used to force the compiler to look in "." (current dir), and *then* in the inc/ subdirectory.]

8. Learn to read and write C++ as "English." A statement of the form

   ```
   return (a == b)
   ```

   Should be read as return TRUE if a is equal to b and FALSE otherwise. Avoid the following:

   ```
   if (a == b) return TRUE;
   else return FALSE;
   ```

9. Do not write "I" unless it is clear who you are.
10. Loops with empty bodies should have a small comment warning the reader. For example

    ```
    for (i=0; check(i); i++)
       ; // null
    ```

    Empty bodies in other constructs should be avoided (*e.g.*, never use them in `if` statements).

## 2.8 Miscellaneous Standards

*It is a well-known fact that programmers, as a breed, are sloppy*
*—Meyers (1994, p. 104)*

1. There should not be any important constants in the code. All constants should be defined in header files or at the top of the file where they are used. References to these name constants should be made in the class headers so that they can be given in the documentation.

   [For example, a limit on the length of a line that can be read from an input file containing the instances should be stated in the class doing the reading.]

2. Overloaded operators should obey the "natural" meaning and follow C++ conventions. For example, operator += should be the same as + and =.
3. Variables should be declared when first used, not at the top of the function.

4. No else after `if (t) return v;` or after a test that calls fatal_error(). If there is an if statement that does a return, do not put the rest of the function in an else.
5. When you have an if statement, try to put the easy/short condition first. That way you don't forget it when you write the code and the reader can see the easy cases first.
6. Avoid inlining functions in header files, except if they really need to be fast or they are one-liners.

   > Except when speed is crucial, this is usually a mistake. Changing the functions requires recompilation of everything, and compilation time grows for any function that includes the header file.
   > One-liners are usually interface functions or access functions for which we want to save the time of writing a header for them in the .c file.
   > Sometimes even a one-liner should be moved to the .c file if its function is not obvious and documentation can help.

7. Parameters should be references except when passing ownership or NULL has a special meaning (*e.g.*, when you want a default argument).
8. Avoid silly typedefs, such as `typedef Array<int> IntArray`. Users will just be confused when seeing IntArray. These are useful only if you added some important functionality.
9. Prefer using initialization list in constructors whenever possible. Initialization order should follow the class declaration order. See Meyers (1994, Item 12).
10. Avoid silly comments such as:
    (a) `i++; // increment i`
    (b) `// This function is intentionally private`
        `// because users do not need to use it.`

    As a rule of thumb, if a program could be written to generate such comments, you are making a fool of yourself.

# A Dropped Conventions

1. We had a convention of avoiding files over 15 characters. The reason was that the archiver on Sun truncated file of length greater than 15 characters. This restriction was lifted in Solaris and is not present on most modern implementations.

# References


Coplien, J. O. (1992), *Advanced C++ Programming Styles and Idioms*, Addison Wesley Pub Co Inc.

Kernighan, B. W. & Plauger, P. J. (1976), *Software Tools*, Addison-Wesley Pub. Co.

Koenig, A. (1992), 'When not to use virtual functions', *The C++ Journal* **2**(2), 31–34.

Meyers, S. (1994), *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, Addison Wesley Pub Co Inc.

Meyers, S. (1996), *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison Wesley Pub Co Inc.

Stroustroup, B. (1994), *The Design and Evolution of C++*, Addison-Wesley Publishing Company.

Stroustrup, B. (1991), *The C++ Programming Language*, second edn, Addison-Wesley Publishing Company.


E N D