$\mathcal{MLC}$++ Coding Standards

Ronny Kohavi
Ronnyk@CS.Stanford.EDU

May 17, 1994

1 Introduction

The purpose of this document is to provide a description of coding standards and conventions used in $\mathcal{MLC}$++. While the ideas are general, they are written for use with C++. Related documents include the $\mathcal{MLC}$++ dictionary (dict.*) that defines the terms used throughout the documentation and code, the $\mathcal{MLC}$++ environment (environ.*) that describes the environment for working with $\mathcal{MLC}$++, and the general C++ coding standards (cc-coding.*).

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.

[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 General Structure

Efficiency is very important when considering algorithms to use. Code should be designed so that it can interface the best algorithms in the future. Current implementations may be inefficient, but the interface to classes should allow transparent exchange of classes with more efficient ones.

Code should be structured (few *goto*s if any), and should avoid hacks and micro-efficiencies (*e.g.*, no register declarations, no inlining of functions over 2-3 lines *etc.*). The idea is to have a *flexible* and an *extensible* code. Performance analysis can be done, and critical code can be made faster.

Files should contain logical units, usually a class or related functions. The goal is to have one big class, and possibly a few derived classes or helper classes per file. Each file has a file-header that describes the purpose of the file and functions it performs. This file-header serves the same purpose as a manual entry (see Section 3).

Code should be defensive (see Section4) and assume that the caller is unaware of assumptions that are made, or that the wrong parameters will be passed. Checks should be made whenever possible to ensure that the arguments are valid. Sanity checks, or internal consistency checks, are also encouraged (see Section 4. Checks may be taken out by preprocessing directives when speed is important.

Code should be tested by tester functions (see Section 5) to ensure correctness and leak-free code.

Terms from the $\mathcal{MLC}$++ dictionary file (dict.*) can be used freely in the code under the assumption that the user is familiar with them. Similarly, conventions set in this document do not have to be explained (*e.g.*, use of Pixes when interfacing GNU-lib).

There is one include file of *basic ontology* (basics.h) which every file must include. This defines objects that are available everywhere (*e.g.*, TRUE/FALSE, error handling, *etc.*).

3 Structure of Source Code

Every source code file starts with two lines that give the project name and a pointer for more information. Source files (".c" files) then give a description of the class or set of functions being implemented (file header), followed by include files that always start with "basics.h," and finally the actual function definitions, each preceded by a function header. Headers files (.h) should be "readable" assuming general knowledge of the class. Irregular calls or short explanations to functions should be made, but most documentation should go into the ".c" files themselves.

> The suffixes ".c" and ".h" are used because Cfront/ObjectCenter require the header and source file to be the same case. We preferred using source files with lowercase "c" than header files with uppercase "H." In order to force editors like Emacs to recognize the fact that the files are C++ and not C, the first line of each file will include the C++ mode setting string.

3.1 Stanzas

$\mathcal{MLC}$++ provides a few stanzas, or templated headers, that should be inserted and filled-in in the appropriate places (described below). There is a stanza for a file header (class header), for include files, for functions, and for testers.

Stanzas should be filled in by the programmer, and should adhere to the following strict guidelines. While $\mathcal{MLC}$++ does not impose many typesetting standards on the source code itself, the headers are more rigid because they are the basis for a manual and they may be used by automatic documentation generators.

1. All information filled in should be to the right of the colons.
2. The structure of the stanzas should not be changed, nor the order. If you have nothing to fill-in, leave the section-name blank and do not erase the line.

   [The ordering is sometimes critical. For example, the RCS string follows the `#include` lines. The reason is that if it came before the `#include`, the header-compilation mechanism of ObjectCenter would fail to ever use compiled headers because the RCS string generates code.]

   The RCS line may be commented out in templates, since it seems to cause problems in the automatically generated files. It is still useful to leave the line in, as RCS replaces the strings it uses, even if inside a comment.
3. If a sentence is more than one line, the next lines should be indented two spaces to the right.

   [This is especially useful if there are many points made under the same section, such as enhancements.]

3.2 Structure of Header Files

Header files (".h" files), sometimes called include files, are structured as follows.

1. Every file begins with two lines with the name MLC++ and a reference to a file describing how to get more information.

   ```
   // MLC++ - Machine Learning Library in -*- C++ -*-
   // See Descrip.txt for terms and conditions relating to use and distribution.
   ```

   [The -*- C++ -*- string is the mode setting string for Emacs.]
2. Header files then follow with the following information:

```
// This is an include file.  For a full description of the class and
// functions, see the file name with a "c" suffix.

#ifndef _CLASS_h
#define _CLASS_h 1
```

Where CLASS should be substituted with the class name. The file ends with `#endif` which closes of the `#ifdef`

> The purpose of the `#ifdef`s is to make sure a file contents to not get included more than once.
> The header stanza file is inc/include.inc

3.3 Structure of Source Files

Source files (".c" files) are structured as follows.

1. Every file begins with the same two lines as the header files.
2. Every file begins with a standard header describing the purpose of the class/functions and their relation to other classes. See Figure 1 for an example.

   > The stanza for this header is src/class.inc.
   > Comments should include essential information that cannot be automatically derived. For example, there is no need to give the whole ancestor hierarchy, since tools like the inheritance browser in ObjectCenter can handle these.

3. Every function, or a few related functions, begin with a standard header describing the input/output, purpose, and details. See Figure 2.

   > All constructors are considered related and should have one header. Related overloaded functions, or functions and their `const` variant should similarly be defined. Some common sets of functions like Pixes should have one header as their behavior is standard. *Interface functions* and *access functions* can also be clumped together if there is not much to say.
   > The idea in giving one header for a few functions is to save the programmer the need to type the same information over and over. If you can describe a set of three **related** functions in one header, feel free to do it. If, on the other hand, the header begins to be long, consider splitting it into more than one header.

```
// MLC++ - Machine Learning Library in -*- C++ -*-
// See Descrip.txt for terms and conditions relating to use and distribution.
/***************************************************************************
  Description  : The DT class provides operations on Decision Trees.
                 It is a subclass of RDG (rooted decision graphs) with added
                 operations for splitting and merging trees.
  Assumptions  : The class destructor assumes that node information should
                 be freed so node information must not have other
                 pointers pointing to it.
  Comments     : A general description of decision trees can be found
                   in "C4.5 : Programs for machine learning" by R. Quinlan.
  Complexity   : Merge and split take log(num-nodes).
  Enhancements : We may want to allow a lazy merge (see Tarjan / Data
                 structures and network algorithms).  Merge will then
                 take constant time, and other operations will take no
                 more than log(num-nodes) time amortized over all
                 operations until the next merge.
  History      : Richard Long                                     9/11/93
                   Added foo to bar.
                 Richard Long                                     7/18/93
                   Initial revision (.c)
                 Ronny Kohavi                                     7/13/93
                   Initial revision (.h)
***************************************************************************/
```

Figure 1: Example of class header.

```
/***************************************************************************
  Description : Merge two subtrees.  Tree B will be a child of node N
                of tree A with the connecting edge testing the given
                test T.
  Comments    : Tree A is modified so that it points to tree B (no copying).
***************************************************************************/
```

Figure 2: Function header.

The source file header

Here is a short description of the fields in the file header:

Description Describe what the conceptual operations on the class are. Don't list all operations as they will be fully described in each function's header.

Assumptions Describe any assumptions you make about the caller, especially about ownership of objects (*i.e.*, who is responsible for deallocating them).

Comments Include important comments for operating with this class and special initialization or termination operations. Mention important warnings or misunderstandings that may arise, and invariants which are expected to be kept by the caller. For top level classes, a reference to some paper may be relevant.

If some parts of the class are unimplemented, mention it here.

Complexity Give basic complexity of **public** and **protected** operations on class (*i.e.*, do not mention private member functions or static functions), with references to special algorithms used. Routines not mentioned are assumed to take constant time.

[Complexity analysis assumes debugging code is turned off.]

Complexity for private and static functions should be mentioned in the appropriate function header if it is non-constant.

[The purpose of this is not to clutter the header with little routines.]

Enhancements Mention any ideas for enhancing the code. This may include ideas for improving specific code, generalizing it, speeding operations, *etc.* Enhancements should be prioritized according to the expected ratio of benefit to amount of work.

History History of changes in reverse chronological order. Only important changes should be logged, as all changes are kept with RCS anyway. Each revision should give the author name, the date, right justified, and the changes indented 2 characters to the right. The initial revision should state the author name and say "Initial revision."

4 Defensive Programming and Error Handling

Functions should test the validity of their input and other assumed invariants. Error reporting should be done using ASSERT() or fatal_error() described below.

4.1 Error Handling

There are two types of error-reporting in $\mathcal{MLC}$++:

Internal consistency check This type of error-reporting is generally for programmers of $\mathcal{MLC}$++and not for people using it. It is intended to be used for "sanity checks" and should be triggered when the class itself has a bug, or when there is some really subtle bug in a class that the code relies on.

To do such tests, use `ASSERT(cond)` where `cond` is some condition to be tested. These statements should be inserted freely into the code. They are easy to add, and can be taken out by a compiler directive.

For example, if two lists were checked to be of the same length, and pointers are used to step through both of them in parallel, it is a good idea to assert that the second is NULL when we exit out of the loop because the first was NULL. It could be that we one pointer was incremented twice, or that the length of a list does not match the actual contents. Both are rare bugs, but some asserts like this actually do get triggered once in a while.

Testers can (and should) use `ASSERT`s to check consistency. There is rarely a need to use fatal_error().

The purpose of asserts is three-fold. The first is to make sure that some constraints and assumptions about how routines behave, hold. The second is to make it clear to readers of the code that such constraints hold at various places of the code. A reader who is reading the code and does not see why the assert holds, might reread portions and better understand the code. Third, it is a way of catching compiler bugs, memory corruptions, and similar problems. The closer an assert is executed to a programmer related to memory corruption, the higher the probability that it will be easier to identify it. Recompilations of code are needed whenever the interface changes, but in many cases the programmers are "smart" and know that a change should not require recompilation. Sometimes the smart programmers are wrong, and such violations tend to cause many assertions to fail.

> One famous case, having to do with the way temporaries are handled, was caught when a copy routine asserted that it is not being give the `this` pointer as an argument. A deeper investigation showed that the routine is called by the copy constructor, and therefore the argument can not be the `this` pointer. Tracing the problem led to a complicated expression that created a temporary that was destroyed prematurely. Since only lately was a standard adopted that temporaries must not be destroyed prior to the largest expression they appear in, this was valid compiler behavior to our version which does not support this new ANSI resolution.

Fatal errors This type of error-reporting is for "users" of the class. They are not interested in looking at the code and want to know what they did wrong. The error should be informative as possible and should contain any information that caused the error to be triggered. Another advantage to this type of error is that it can be caught if "expected." This is used by testers to test correctness of the class.

The standard for error reporting using fatal_error() is the following (note that no newline or period is needed at the end):

```
err << "Class::member: " << info << ... << fatal_error;
```

By informative, we mean that if the size cannot be negative, the following is a good error:

```
err << "Class::init: Negative size (" << size << ") for array"
```

Sometimes there are two functions which only differ on the fact that one is const and the other is not. In such cases, the member identification should include the word const. For example:

```
err << "Class::member const: " << info << ... << fatal_error;
```

If a function is a global function and not inside a class, the file name should be given instead of the class.

4.2 Optional Error Handling

Some routines have optional error-handling behavior. If a routine is called and a fatal_error should be generated when it returns some value (usually false), it is useful to allow it to do the error handling.

Such routines should have an extra parameter called "fatal_on_false" (or similar for other values) which calls fatal_error if the check is false. For example, callers to `InstanceInfo::equal` would like to abort on unequal result. Adding "fatal_on_false" takes the burden off the caller.

[In such cases, `operator==` can be defined to call `equal` with the argument for "fatal_on_error" set to false, so it behaves exactly like `operator==` should.]

4.3 Expensive Checks

Code which can drastically affect execution time may be conditionally included using preprocessor directives. The preprocessor macro DBG() is defined to include the code given as argument only if FAST is not defined. To compile the code for fast execution, you can compile with `-DFAST`. The usage of DBG() is as follows:

```
DBG(if (cond) err << "Class::func: cond failed" << fatal_error);
```

Multiple statements can be given, but they have to be separated by semicolons. If DBG() appears inside an if it must be balanced (note the trailing else), or the statements should be included in braces:

```
if (y == 0)
   DBG(if (x==0) err << "Class::func: X is zero" << fatal_error; else);
else
   y++;
```

4.4 Guideline for Catching Errors

Error handling should be as early as possible in a function. Not only does the actual code have less error-checks, but it aids the reader who is aware that functions begin with error checks. Expensive checks should be enclosed in `DBG()` so that they can be removed by a compiler switch (described above).

It is a good idea to put DBG(OK()) as the first line of the destructor. This ensures that the class is in a good state, and this is the last point where we can check this, since the class will be destroyed soon.

It is rarely the case that an error should be in the **else** part of an **if**. Always try to move it above the code. For example, instead of

```
if (t) {
   statements;
} else
   err << ...  <<  fatal_error;
```

the code should be

```
if (!t)
  err << ... <<  fatal_error;

statements;
```

[One case where the error is handled "last" is in switch statements where an error is caught using the `default` catch-all.]

Since a fatal_error() call never returns, do not put an else after a check that calls fatal_error. This avoids unnecessary indentation.

5 Testers

Each testable class should have a file that tests it, with a "t_" prefix to the class name. This test file should be run whenever changes are made to ensure back compatibility and that no new errors are introduced. The standard $\mathcal{MLC}$++ **put** command causes recompilation of the tester automatically.

[The stanza file for tests is in src/tests/test.inc]

Test programs (testers) should test all public members of the class they are testing, except those specifically listed in the file header (members mentioned in the header could be unimplemented members for example).

Test programs should also test the ability of the tested class/function to handle errors. The "errorUnless.h" file provides macros that allow testing for expected fatal error messages. The macro:

```
TEST_ERROR(msg,stmts)
```

executes the given statements which are expected to generate a fatal error that contains the given message *msg.* If the fatal error is indeed generated as expected, execution continues; otherwise, an automatic fatal error is generated saying the test failed.

Testers should test for boundary conditions, and should conduct many extensive tests defined by constants appearing in the beginning of the file. Try to avoid symmetry when testing, and use "strange" strings and numbers. Remember that testers may and should contain many bizzare constants.

A good example of a bad tester was the tester for the Array2 class (two dimensional arrays), which tested it on a square matrix. The `operator()` was buggy and the element was accessed by `[(row-startRow)*numRows + (col-startCol)]` instead of by `[(row-startRow)*numCols + (col-startCol)]`. The tester did not catch this mistake since NumRows was equal to NumCols.

It is very important to test for memory leaks during tests. A tester which is known to leak (because it tests fatal errors) should have `#ifdef` statements to allow testing only the parts that do not leak. For example,

```
#ifndef MEMCHECK
   TEST_ERROR("<msg>", code);
#endif
```

Memory leaks are the hardest to find and since long experiment using $\mathcal{MLC}$++ should activate many routines many times, leaks can really cause degradation of performance due to increase in page faults.

Test runs should be repeatable, so runs that use random number generators should use a fixed seed.

It would be a real surprise if a class is tested with a tester and fails because the seed is different, independent of the change that was done. If there is no call to srandom() or some similar seeding routine, all runs will be the same.

A tester should have a file with the same name and a ".exp" extension (expected output). The makefile runs the test and direct the output to a file with a ".out" suffix. A diff then compares the two files to ensure consistency.

Testers may optionally have ".exp#" where # is 1, 2, or 3. The tester must then generate ".out#" files which will be compared to the corresponding expected output file. Similarly, if a ".cin" file exists, input to the tester will be taken from this file.

The returned status from the test should be zero if (and only if) everything is OK. Thus the `main()` function should return 0 in Unix systems.

6 Subtle Problems

This section describes some subtle problems which are only partially resolved. Ideas for improvements will be appreciated.

6.1 Static Objects

The constructors for static classes are called in an undefined order if they are in different files. If a constructor uses the standard error-handling convention, it may have something like the following code:

```
err << msg << fatal_error;
```

before the `err` stream, defined as

```
ostrstream err(err_text, max_error_message_size);
```

is allocated. This causes core-dumps in most cases.

The solution is to declare all static objects in one file, basics.c. Since initialization order within a file is guaranteed to be in order of declaration, we can enforce `err` to be initialized first.

6.2 Fatal Errors

The fatal_error() function uses other functions before actually exiting. If one of those functions calls fatal_error(), we will have an infinite loop in most cases.

The solution to this problem is that fatal_error() sets a flag when it is called, and checks that flag just before setting it. If it discovers that the flag was set, it dumps the error stream without any manipulations, and aborts. Interestingly, since we use an error stream, both error messages will be displayed!

6.3 Exhausting Memory

If memory is exhausted, the $\mathcal{MLC}$++ out_of_memory_handler() function is called (this is an option that can be set by set_new_handler()).

Since we want to call fatal_error(), but are aware of the fact that to exit gracefully fatal_error() will need some memory, a block of memory is allocated when $\mathcal{MLC}$++ starts and deallocated when memory is exhausted. This ensures that fatal_error() can exit gracefully.

6.4 Name Conflicts

At some stage or another we will conflict with names of software we use. We already had one conflict where X-windows uses `typedef char* String`, but they have an optional `#define` to avoid using it.

If we do have conflicts, one possible idea is to do what InterViews did, that is, to define a "scope.h" and "unscope.h" files which gives some classes a new name, say with an MLC prefix. Unscope undefines those in case you need access to the original names.

The best thing to do is to avoid name conflicts with anything we are aware of. To resolve obvious conflicts, a prefix could be added, making it $\mathcal{MLC}$++ specific. For example, $\mathcal{MLC}$++ strings may be MString. Less common classes may have the `MLC` prefix.

END