OAK RIDGE
NATIONAL LABORATORY
MANAGED BY UT-BATTELLE
FOR THE DEPARTMENT OF ENERGY

ORNL/TM-2009/240

C++ Coding Standards for the AMP Project

September 2009

Prepared by
T. M. Evans
K. T. Clarno

UT-BATTELLE
ORNL-27 (4-00)

ORNL/TM-2009/240

Nuclear Science and Technology Division

C++ CODING STANDARDS FOR THE AMP PROJECT

T. M. Evans
K. T. Clarno

Date Published: September 2009

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6283
managed by
UT-BATTELLE, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

C++ Coding Standards for the AMP Project

Thomas M. Evans and Kevin T. Clarno

ORNL/TM-2009/240

Revision 0
September 30, 2009

1 Introduction

This document provides an initial starting point to define the C++ coding standards used by the AMP nuclear fuel performance integrated code project and a part of AMP's software development process. This document draws from the experiences, and documentation [1], of the developers of the Marmot Project at Los Alamos National Laboratory.

Much of the software in AMP will be written in C++. The power of C++ can be abused easily, resulting in code that is difficult to understand and maintain. This document gives the practices that *should* be followed on the AMP project for all new code that is written. The intent is not to be onerous but to ensure that the code can be readily understood by the entire code team and serve as a basis for collectively defining a set of coding standards for use in future development efforts. At the end of the AMP development in fiscal year (FY) 2010, all developers will have experience with the benefits, restrictions, and limitations of the standards described and will collectively define a set of standards for future software development.

External libraries that AMP uses do not have to meet these requirements, although we encourage external developers to follow these practices. For any code of which AMP takes ownership, the project will decide on any changes on a case-by-case basis.

The practices that we are using in the AMP project have been in use in the Denovo project [2] for several years. The practices build on those given in References [3–5]; the practices given in these references should also be followed. Some of the practices given in this document can also be found in [6].

2 Organizing and Writing Code

This section deals mainly with how code should be organized and written in terms of files, file names, code formatting, and documentation.

2.1 File Names

AMP will follow the Nemesis naming convention for file names [7]. Each class is defined within its own set of files, as summarized in Table 1. See Appendix B for examples. Note that multiple classes should not be defined in a single set of files, except in very special circumstances where the classes are closely related (for example, nested classes, or an iterator class for a container).

There may be other translation units outside of classes, such as for `main()` (for example, main.cc), and header files that contain free functions or constants (both of which must be in a unique `namespace`). Such file names should begin with a lowercase letter.

2.2 Template Model

Following Nemesis, there are two models that are used for template instantiation, based on usage:

1. *Automatic*: This is for templated classes for which it is unlikely that the template arguments are known beforehand. Examples are containers and smart pointers (which can contain or point to any data type). These classes include their function definitions within their header file (.hh), either explicitly or via a .i.hh file. Hopefully, to avoid code bloat and excessive compile times, these classes are small.

Table 1: File Naming Convention for Class `A`.

File	Required	Contents
A.hh	Yes	Header file. Contains definition of class `A`. It may also contain member function definitions, although preferably, function definitions should be in one or more of the files A.cc, A.i.hh, and A.t.hh.
A.cc	No	Implementation file. If `A` is non-templated, then contains non-templated member function definitions of `A`. If `A` is templated, contains member function definitions of specializations of `A`.
A.t.hh	No	Template implementation file. May be used if `A` is a templated class, or if `A` contains templated member functions. In these cases, contains the corresponding function definitions which will be explicitly instantiated by A.t.cc.
A.i.hh	No	Implementation file for member functions. This file should *always* be included at the bottom of A.hh. If `A` is templated, or has templated member functions, then this file can be used for an automatic instantiation model. For non-templated entities, this file may contain `inline` function definitions.
A.t.cc	No	Instantiation file. Used for explicit template instantiation of the definitions in A.t.hh. For specific template arguments, contains instantiations of `A` or its templated member functions.
test/test_A.cc	Yes	Unit test file. The unit test for `A`. Other files may also be used by the unit test.

2. *Explicit*: This is for templated classes for which the range of template arguments is known. An example is a finite-volume class that is templated on the mesh type (the number of mesh types one typically requires is known and fairly small). Here, the function definitions (.t.hh) are included by the template instantiation file (.t.cc). The instantiation file instantiates the class for each desired template argument.

See Appendix B for examples. Note that the STL follows the automatic instantiation model.

2.3 Write Unit Tests

The author of a class must write a unit test for that class. The unit test not only tests the functionality of the class but also helps serve as an example of how to use the class. Some authors actually prefer to write the test before the class. See §2.1 for where the unit test should reside.

2.4 Syntax Names and Code Formatting

Rules for names and code formatting can be onerous. We believe many such rules are based more on personal preference and do not significantly add value to a code's readability. If too many rules are specified, teams tend to ignore *all* the rules, or become unhappy with enforcement. The intent here is to have code that is quickly understandable by anyone on the AMP team. But individual team members may not find each other's style "pretty."

Consequently, we have narrowed the list of rules to those we believe are the most important:

1. Indent your code as follows:
   - Do not indent curly braces relative to their control statements.
   - Do not indent `namespace` blocks.
   - Indent `public`, `private`, `protected` statements two spaces relative to their `class`'s braces.
   - For other code blocks, indent four spaces relative to their enclosing braces.

- Comments on their own line should be indented to the same level as the code they are commenting.

2. Be consistent in spacing, bracket placement, formatting, etc. If modifying someone else's code, respect and attempt to follow their style (otherwise, you are introducing inconsistency).

3. Separate words and acronyms within a name with an underscore.

4. Use complete words or acronyms in names. For example, use `is_anal_retentive` instead of `is_anl_ret`.

5. Distinguish variable and function names from user-defined type names as follows: Begin all words in type names with a capital letter. Begin all words in variable and function names with a lowercase letter.

6. One should not have to search outside of a file, or parse an entire function definition, to determine the origin of a name used within that file (ideally, apply this down to function definitions). This rule implies the following:

   - A convention for member data names, such as using the prefix "`d_`", or the prefix `this->`. One could extend this idea to nested types, but one rarely defines new types within function scope, so the origin of types not fully scoped is usually clear.
   - Even within an implementation file, avoid "`using namespace`". One exception here is "`using namespace std`", as long as you are using very common stuff from `std` (`cout`, `endl`, `vector`,...). It is still preferable that you explicitly specify which names you are using (e.g., `using std::cout`). See also §3.10.

7. Limit the body of a function definition to approximately one page in length.

8. Declare only one variable per line, including within function declarations:

```
int i;
int j;

void blorp(int i,
           int j);
```

9. Do only one operation per line. For example, avoid

```
a = z; b = std::sin(a);
c = d = g = 0;
if ( a = b )
{
    // executes if 'a' is true; note there is only one '='
}
```

An example with the naming and format conventions described in this section is shown in Appendix A. Templates will be developed to ease the use of the source code indentation and comment formatting.

2.5 Code Comments and Documentation

Code must be reasonably commented and documented using Doxygen. Templates will be developed to provide uniform headers throughout the code that reconize the tri-laboratory development of the software and consistent formatting for documenting the code using Doxygen. Our preference is to put one-line comments for each member function in the header file and its documentation in the implementation file. See Appendix A for an example.

User documentation is separate from the code's documentation. Standards for user documentation are not covered by this document.

3 Use of Language Features

This section gives rules on language features to use and to avoid.

3.1 Use a `namespace`

All implementations must have their own `namespace`. The name of the `namespace` will be determined by the AMP project.

3.2 Enforce `const`-correctness

See *Effective C++* [3, Item 21]. When declaring a variable or function, use the `const` qualifier whenever possible. If unsure when declaring the variable, then go ahead and add the `const` qualifier. You can always remove the qualifier later. We discourage strongly the practice of adding `const`-correctness after major features have been implemented.

AMP may have to use external libraries that do not enforce `const`-correctness. Such libraries must be wrapped in the equivalent functionality that enforces `const`-correctness. Otherwise, if wrapping is not done, the external library potentially could force all of AMP to abandon `const`-correctness. Wrapping has other benefits, such as that the external library can be swapped out.

3.3 Use Design-by-Contract (DBC)

The design-by-contract macros defined in Nemesis's `rtt_dsxx::Assert`, or an equivalent, must be used extensively. Example 1 shows how DBC can be used.

```
#include <ds++/Assert.hh> // defines Require, Check, Ensure, ...

double pressure(const double temperature,
                const double density)
{
    Require(temperature >= 0.0); // use Require() to check input values
    Require(density >= 0.0);

    double p; // pressure that is returned

    // ... code that computes initial guess for p ...

    Check(p >= 0.0); // use Check() for intermediate calculations

    // ... code that computes final value of p ...

    Ensure(p >= 0.0); // use Ensure() for final values

    return p;
}
```

Code Example 1: Example that uses Design-By-Contract (DBC). There is other functionality in Nemesis's DBC not shown here, such as `Insist()` and `Remember()`.

3.4 Classes Should Hide Their Data

See *Effective C++* [3, Item 20]. Aside from pure data structures (i.e., a `struct`), class member data should be accessed only through member functions. There are several suggestions for accessors:

- If the class is a data container (for example, a container of cell-centered pressures on a mesh), then data container semantics may be used to access the underlying data. These semantics include `operator[]`, `operator()`, and iterator access.

```
class Solution
{
    // DATA

    typedef std::vector<double> CCF; // cell-centered field
    CCF d_pressure;
    CCF d_density;

  public:

    // ACCESSORS

    CCF &pressure() { return d_pressure; }
    const CCF &pressure() const { return d_pressure; }

    CCF &density() { return d_density; }
    const CCF &density() const { return d_density; }

    // ITERATORS

    CCF::iterator begin_pressure() { return d_pressure.begin() }
    CCF::iterator end_pressure() { return d_pressure.end() }

    // ... etc ...
};
```

Code Example 2: Example with accessors to large data structures; presumably, the `CCF` class contains a large amount of data. Care must be taken that `Solution` is not destroyed while handles it has returned are still available. We might want to revisit this issue.

- If the member data is a large data structure, then for efficiency, handles to that data may be returned. The handles may be in the form of iterators or references, as in Example 2.
- Otherwise, "get/set" semantics should be used. The "get" function must not return a non-`const` handle to the data.

3.5 Avoid `friend`

There are specialized cases where `friend` is useful (such as an iterator class for a container class), but generally, the use of `friend` is strongly discouraged. The use of `friend` violates data hiding, which was covered in the previous section.

3.6 Avoid Macro Functions

See *Effective C++* [3, Item 1]. Macro functions are not type-safe and should be avoided.

3.7 Avoid Raw Pointers

The use of raw pointers (for example, `double *x`) can be a major source of bugs. Often, the use of raw pointers can be avoided by substituting one of following techniques:

- Use a "smart pointer" [4, Item 28] instead (for example, see `rtt_dsxx:SP` in Nemesis).
- Use a reference. See *More Effective C++* [4, Item 1].

There are situations where using a raw pointer cannot be avoided. For example, raw pointers cannot be avoided when communicating with other languages, such as Fortran or C. In other cases, their use should be encapsulated. For example, container classes often use a pointer for their underlying data storage and may define its `iterator` type as pointer via a `typedef`. However, the pointer implementation in this case is encapsulated from the user of the container class.

Finally, pointers-to-functions are a relic of C and can be avoided through the use of virtual functions.

3.8 Avoid Circular Object Dependencies

See the discussion on levelized design in *Large-Scale C++ Software Deesign* [6]. A simple example of a circular dependency is shown in Example 3. Because both classes `A` and `B` refer to one another, they cannot be tested independently. Again, there may be special cases where circular dependencies are acceptable, such as between a container and its `iterator` type.

```
class B; // forward declaration

class A
{
    int do_b(const B &b); // refers to class B
};

class B
{
    int do_a(const A &a); // refers to class A
};
```

Code Example 3: Example of a Circular Dependency. Classes `A` and `B` both refer to one another.

3.9 Explicitly Use the `std-namespace`

For example, use '`#include <cmath>`' instead of '`#include <math.h>`'. In general, do not use the '`.h`' system header files, which pollute the global `namespace`. See also *Effective C++* [3, Item 2].

3.10 No `using` Statements in Header Files

Do not place `using` statements where they might pollute the global namespace. Unless within the scope of an `inline` function, `using` statements should not be placed within header files (.hh). Even within implementation files, `using` statements preferably should appear only within function scope.

You might argue that `namespace using_abuse` is "yours," and you are free to pollute it all you like. However, someone else might have to maintain your code in the future. Also, placing `using` statements with header files may affect those who want to use your class, as illustrated in Example 4.

References

[1] R. Lowrie and T. M. Evans, "C++ coding standards for the Marmot project," Research Note CCS-4:03–52(U), Los Alamos National Lab., 2002.

[2] T. Evans, A. Stafford, and K. Clarno, "Denovo—A new three-dimensional parallel discrete ordinates code in SCALE," *Nuclear Technology*, 2009. submitted for publication.

[3] S. Meyers, *Effective C++*. Addison Wesley, second ed., 1998.

[4] S. Meyers, *More Effective C++*. Addison Wesley, 1996.

[5] S. Dewhursh, *C++ Gotchas*. Addison Wesley, 2003.

[6] J. Lakos, *Large-Scale C++ Software Design*. Addison Wesley, 1996.

[7] T. Evans, "The Nemesis build system." In development, 2009.

File A.hh

```
#include <iostream>

namespace using_abuse
{

using namespace std; // Not here!!!

class A
{
  public:
    void print_something() { cout << "I am lazy.\n"; }
};

} // end of namespace using_abuse
```

File uses_A.cc

```
#include "A.hh"

int main()
{
    // The following using statement is OK, because it's in an implementation
    // file.  Unforunately, it asks for using_abuse, but got std too!!!
    using namespace using_abuse;

    A a; // using_abuse::A is OK too
    a.print_something();

    cout << "Where did cout come from???\n"; // Answer: via using_abuse.
}
```

Code Example 4: Example showing the improper placement of `using` in a header file. File uses_A.cc never explicitly asks to use `namespace std`.

A Sample Code

This appendix shows an example for a class named `My_Class`, which uses the preferred naming and formatting conventions given in §2.4.

File My_Class.hh

```
//---------------------------------*-C++-*----------------------------------//
/*!
 * \file    my_Component/My_Class.hh
 * \author Kevin Clarno
 * \date    Wed Oct 22 12:12:09 2003
 * \brief
 * \note    Cooperatively devleoped by Oak Ridge National Laboraotry,
 *                                     Idaho National Laboraaory,  and
 *                                     Los Alamos National Laboratory.
 */
//---------------------------------------------------------------------------//
// $Id: My_Class.hh,v 1.1 2003/11/20 17:27:37 tme Exp $
//---------------------------------------------------------------------------//

#ifndef rtt_my_component_My_Class_hh
#define rtt_my_component_My_Class_hh

#include <vector>
#include <ds++/Assert.hh>

namespace rtt_my_component
{

//===========================================================================//
/*!
 * \class My_Class
 * \brief This is a sample class.
 *
 * This is a sample class, showing the preferred naming
 * scheme, comment style, and Doxygen documentation.
 *
 * \sa My_Class.cc for detailed descriptions.
 *
 * \example my_Component/test/My_Class_test.cc
 *
 * description of example
 */
//===========================================================================//

class My_Class
{
  public:

    // NESTED CLASSES AND TYPEDEFS

    typedef std::vector<int> My_Vector_Int;

    // CREATORS

    //! default constructors
    My_Class();

    //! copy constructor
    My_Class(const My_Class &rhs);

    //! destructor
    ~My_Class();

    // MANIPULATORS

    //! Assignment operator for My_Class
    My_Class& operator=(const My_Class &rhs);

    //! Example of a public function
    int my_public_function(const double t,
                           const int j);
```

```
    // ACCESSORS

    //! Returns the courant number
    double get_courant() const { return d_courant; }

    //! Sets the courant number
    void set_courant(double courant)
    {
        Require(courant > 0.0); d_courant = courant;
    }

    //! Returns the iteration count
    int get_iteration_count() const { return d_iteration_count; }

  private:

    // NESTED CLASSES AND TYPEDEFS

    // IMPLEMENTATION

    //! Example of a private function
    double my_private_function();

    // DATA

    //! Iteration count of linear solver
    int d_iteration_count; // member data begins with d_

    //! Courant number for time step control
    double d_courant; // member data begins with d_
};

} // end namespace rtt_my_component

#endif // rtt_my_component_My_Class_hh

//---------------------------------------------------------------------------//
//              end of my_component/My_Class.hh
//---------------------------------------------------------------------------//
```

File My_Class.cc

```
//---------------------------------*-C++-*----------------------------------//
/*!
 * \file    my_Component/My_Class.cc
 * \author Kevin Clarno
 * \date    Wed Oct 22 12:12:09 2003
 * \brief
 * \note    Cooperatively devleoped by Oak Ridge National Laboraotry,
 *                                     Idaho National Laboraaory,  and
 *                                     Los Alamos National Laboratory.
 */
//---------------------------------------------------------------------------//
// $Id: My_Class.cc,v 1.1 2003/11/20 17:27:37 tme Exp $
//---------------------------------------------------------------------------//

#include "My_Class.hh"
#include <cmath>

namespace rtt_my_component
{

//---------------------------------------------------------------------------//
/*!
 * \brief Default constructor
 */
My_Class::My_Class()
    : d_iteration_count(0)
    , d_courant(0.0)
{
    return;
}

//---------------------------------------------------------------------------//
/*!
 * \brief Destructor
 */
My_Class::~My_Class()
{
    return;
}

//---------------------------------------------------------------------------//
/*!
 * \brief Sample public function
 *
 * \param t Time spent on marmot
 * \param j Number of complete pairs of socks
 * \return Your estimated lifespan, in years
 */
int
My_Class::
my_public_function(const double t,
                   const int    j)
{
    using std::floor;

    Require(t > 0.0);
    Require(j >= 0);

    int i = j * floor(my_private_function() * t); // local variable

    Ensure(i >= 0);

    return i;
}

//---------------------------------------------------------------------------//
// PRIVATE FUNCTIONS
```

```
//---------------------------------------------------------------------------//

//---------------------------------------------------------------------------//
/*!
 * \brief Computes the maximum nondimensional total time.
 *
 * \return The desired result.
 */
double
My_Class::
my_private_function()
{
    if ( d_iteration_count > 50 )
    {
        // ... encase in braces, even if it's a single statement
        d_courant *= 0.1;
    }

    Check(d_courant > 0.0);

    return d_iteration_count * d_courant;
}

} // end namespace rtt_my_component

//---------------------------------------------------------------------------//
//                 end of My_Class.cc
//---------------------------------------------------------------------------//
```

B Examples of Uses of the Various File Extensions

This appendix shows examples of the various file extensions covered in §2.1. Code Example 5 is a main program that uses a templated class `B`. Code Examples 6 and 7 show how the implementation of `B` may be organized using automatic instantiation. Code Example 8 shows the implementation using explicit instantiation.

File uses_B.cc

```
#include "B.hh"

int main()
{
    B<int> bi(1);
    B<double> bd(2.3);
    bi.print_data();
    bd.print_data();
}
```

Code Example 5: Example use of templated class B.

File B.hh

```
#include <iostream>

#ifndef rtt_my_component_B_hh
#define rtt_my_component_B_hh

template <class T>
class B
{
    T d_data;
  public:
    B(const T &data) : d_data(data) {}
    void print_data();
};

template <class T> void B<T>::print_data()
{
    std::cout << d_data << std::endl;
}

#endif
```

Code Example 6: Automatic instantiation model for class B.

File B.hh

```

#ifndef rtt_my_component_B_hh
#define rtt_my_component_B_hh

template <class T>
class B
{
    T d_data;
  public:
    B(const T &data) : d_data(data) {}
    void print_data();
};

#include "B.i.hh"

#endif
```

File B.i.hh

```
#include <iostream>

template <class T> void B<T>::print_data()
{
    std::cout << d_data << std::endl;
}
```

Code Example 7: Automatic instantiation model for class B, using B.i.hh file.

File B.hh

```
#ifndef rtt_my_component_B_hh
#define rtt_my_component_B_hh

template <class T>
class B
{
    T d_data;
  public:
    B(const T &data) : d_data(data) {}
    void print_data();
};

#endif
```

File B.t.hh

```
#include "B.hh"
#include <iostream>

template <class T> void B<T>::print_data()
{
    std::cout << d_data << std::endl;
}
```

File B.t.cc

```
#include "B.t.hh"

template class B<int>;
template class B<double>;
```

Code Example 8: Explicit instantiation model for class B. In this case, B.t.o must be linked with uses_B.o.