C++ Coding Standards

The Corelinux Consortium

Revision 1.6
Created on May 8, 2000
Last Revised on September 22, 2000

Abstract

This document describes the C++ Coding Standards as they are used in the corelinux++ project. It provides a set of guidelines, rationales and standards for C++ coding.

Contents

Copyright notice

1 Scope

The principle function of this document is to establish a consistent standard which will provide for easier maintenance of code. This will benefit the team and the project in that those who are new

to the code can quickly orient themselves, and thereby sooner become productive members of the team. It is intended to be a dynamic document and can be reviewed as needed. It is recommended that each programmer keep a copy on hand also.

1.1 Document Overview

The following document sections contain standards, guidelines, and rationales. Guidelines must be adhered to unless there is compelling reason to deviate. Deviation from a guideline must be discussed and approved during the code walk-through. A standard is an item to which compliance is mandatory. Deviation from a standard must be discussed, approved, and signed-off on by the team lead during the code walk-through. Rationales have been used where necessary to explain the meaning of an item, or why it was chosen.

General Principles This section contains the basic philosophy a developer should keep in mind while coding.

Comments This section deals with the placement and contents of comments in the code.

Code Layout This section has to do with the alignment of the code, white space, declarations and keywords, and where they should all be located.

Naming Conventions This section contains the structure for naming classes, functions, files, and directories.

Usage This section concerns the 'how' and 'when' certain constructs should be used (for example, loops, inheritance, error handling, etc.)

File Layout This section applies to where things should be located in header files and modules.

Coding Examples This section contains an example of a header file and module which conform to the `C++` Coding Standard.

2 General Principles

The primary goal of the coding standard is maintainability. Other important considerations that relate to the spirit of the standard includes correctness, readability, consistency, extensibility, portability, clarity, and simplicity. When in doubt, the programmer should strive for clarity first, then efficiency.

Think of the reader. Do not just write for yourself. Keep it simple. Break down complexity into simpler chunks. Clearly comment necessary complexity. Be explicit. Avoid implicit or obscure language features. Be consistent. Minimize scope, both logical and visual.

3 Comments

Guideline 1 *Be clear and concise. Say what is happening and why. Do not restate code.*

Guideline 2 *Keep code and comments visually separate.*

Standard 1 *Use top of file comments for all files.*

Standard 2 *Use header comments for all functions. The size of the comment should mirror the size and complexity of the code. Class interface functions and procedures should be commented in the header.*

Standard (`.hpp`) 1 *Use JavaDoc commenting style (`///` and `/** ... */`) . We will use an* ***HTML*** *generator tool to produce class library documentation.*

```
namespace corelinux
{
    DECLARE_CLASS( SomeClass );

    /**
    Main documentation for the class appear after namespace and any type or
    class macros and above the actual class declaration.
    Unless of course it is NOT a CoreLinux++ class.
    */

    class SomeClass
    {
    };
}
```

for methods

```
/**
Explanation of method
@param Type modifier optional summary
@return Type modifier @exception
Type [,Type]
*/
```

–or–

```
/// Destructor
```

for members

```
/// Single line primarily for data members
}
```

Standard (`.cpp`) 1 *Use only the `C++` comment style (double slashes) for single line comments.*

Guideline 3 *For block comment styles either `C++` (double slashes) or C style (`/* ... */`) can be used.*

```
// ...
```

–OR–

```
// // ...  //
```

–OR–

```
/* ...  */
```

Standard 3 *The trailer comment should only include configuration management information defined as:*

```
/*
  Common rcs information do not modify
  $Author: dulimart $
  $Revision: 1.6 $
  $Date: 2000/09/22 23:44:22 $
  $Locker:  $
*/
```

Guideline 4 *Prefer block comments over trailing comments. Use block comments regularly. Only use trailing comments for special items.*

Standard 4 *Trailing comments must all start in the same column in the function.*

Standard 5 *Trailing comments at a closing brace are indented one level from brace.*

Standard 6 *Block comments are at the same indentation level as the block they describe.*

Standard 7 *Ensure comments are correct (and stay correct).*

4 Code Layout

Guideline 5 *Write code in a series of chunks.*

Guideline 6 *Use block comments to separate the chunks.*

Standard 8 *Put one statement per line, except with in-line code in a header file.*

Guideline 7 *Functions shall have a single exit point.*

Rationale: Multiple exit points usually add to the complexity of a function. Post conditions and invariant checks must also be performed prior to each return.

Standard 9 *Indentation level is three (3) spaces.*

4.1 Braces and Parenthesis

Standard 10 *Braces shall be aligned using Ulman style where the braces are at the same scope as the statement that proceeds them and the code within the braces is indented one level. Braces are always on a line by themselves.*

```
void    doSomething( void )
{
   if( x != y )
   {
      y = x;
   }
   else
   {
      ; // do nothing
   }
}
```

–NOT–

```
void    doSomething( void )
{
   if( x != y )
   {
   y = x;    // should be indented
   }
   else
   {
      ; // do nothing
   }
}
```

– OR –

```
void    doSomething( void )
{
   if( x != y )
      {             // brace is not the same scope
      y = x;
      }
   else
      {
      ; // do nothing
      }
}
```

– OR –

```
void    doSomething( void )
```

```
{
   if( x != y ){   // brace is not on a line by itself
      y = x;
      }
   else{
      ; // do nothing
      }
}
```

Standard 11 *While, for and if statements shall always use braces, even if they are not syntactically required.*

Guideline 8 *Use parenthesis to group within a statement and to emphasize evaluation order.*

Guideline 9 *Avoid unnecessary parentheses.*

Guideline 10 *Avoid deep (more than three) levels of parenthesis nesting.*

4.2 Declarations

Standard 12 *Order declarations as storage class specifier, type specifier as described in ISO/IEC 14882 C++ standard, section 7.1.*

Standard 13 *Start each declaration on a new line.*

Standard 14 *Enumeration declarations will be declared with named constants in upper case and the identifier specified as described in Naming conventions.*

```
enum Identifier
{
   ONE,
   TWO,
   THREE
};
```

– or –

```
enum Identifier
{
   ONE = 1,
   TWO,
   THREE
};
```

Standard 15 *All variables must be initialized.*

Guideline 11 *Use vertical alignment to ease scanning of declarations.*

```
String  aStringToUse;
Int     anInt;
Real    aRealNumberToUse;
```

–instead of–

```
String  aStringToUse;
Int anInt;
Real  aRealNumberToUse;
```

Standard 16 *Do not create anonymous types (structs), except in a class declaration where a private structure is declared.*

4.3 Keyword Constructs

Guideline 12 *Most if statements should be followed by an else statement.*

Standard 17 *Empty statements shall have a semicolon with a* `// do nothing` *comment.*

```
if( x != y )
{
   y = x;
}
else
{
   ; // do nothing
}
```

Rationale: The statement clearly shows that the develop has thought about the condition.

Standard 18 *Nested else if statements shall be indented as normal statements, the if shall appear on the same line as the else, and shall have a final else statement (most likely with a* `NEVER_GET_HERE;` *statement).*

```
if( x > y )
{
   x = x - y;
}
else if( x < y )
{
   y = y - x;
}
else
{
   NEVER_GET_HERE; //  See Assertion.hpp
}
```

Standard 19 *Indent cases one level from the* `switch` *and indent the code one level beyond the case. The break statement is at the same indentation level as the code.*

Standard 20 *All* `switch` *statements shall have a* `default` *case. If all cases have been handled then the default code shall be* `NEVER_GET_HERE`*. If not then it shall be an empty statement.*

```
switch( variable )
{
   // ...

   case  1:

      break;

   // ...

   case  2:

      break;

   // ...

   default:
      NEVER_GET_HERE;   //  See Assertion.hpp
      break;
}
```

– OR –

```
switch( variable )
{
   // ...

   case  1:
      break;

   // ...

   case  2:
      break;

   // ...

   default:
      ;         // do nothing
      break;
}
```

Standard 21 *Put the* `while` *in a* `do while` *statement on the same line as the closing brace.*

```
do
{
   ++x;
} while( x < y );
```

4.4 Preprocessor

Guideline 13 *Preprocessor directives should be avoided whenever possible.*

Standard 22 `#define` *shall not be used for manifest constants. Use* `const Type name = value;` *instead.*

```
#define SOME_MAGIC_NUMBER       5       // wrong

const Int       magicNumber(5);         // ok
```

Standard 23 *For CoreLinux++ developed header files use the standard* `#include` *method:*
`#include <Common.hpp>`

Guideline 14 *A code base should define a central include header that maps to* `INCL_ClassName`*: e.g. cpp or hpp:*

```
#include   <ProjectIncludes.hpp>

#if !defined(__MAILBOXENTRY_HPP)
#include INCL_MailBoxEntry
#endif
```

and in ProjectIncludes.hpp:

```
#define INCL_MailBoxEntry <mbxe.hpp>
```

Rationale: In large projects many compilation units may refer to the same header includes. By separating the actual file name and location it makes it easier to maintain directory and or file name changes without having to change the compilation units.

Standard 24 *Multi-statement macros shall have one statement per line.*

```
#define MULTIPLE_LINE_MACRO( s )        \
        ++(s);                          \
        (s) = ((s) % 3 ? ++(s) : (s) )
```

4.5 Spaces

NOTE: Adhere to the indentation standard.

Standard 25 *Do not use spaces in object de-references.*

```
val = *pFoo;    // ok
val = * pFoo;   // wrong
```

Standard 26 *Do not space between an unary operator and its operand, but do space the other side.*

Standard 27 *Balance spacing on either side of binary operators.*

Standard 28 *Do not space before separators (semicolon, argument comma separator) but do space the other side.*

Guideline 15 *It is acceptable to put a space before the semicolon that terminates a statement, as long as it is consistent throughout the function.*

Standard 29 *Balance spacing inside parenthesis.*

Standard 30 *Do not space between function name and parenthesis, but space after open parens and before close parens with the exception of no arguments. e.g.*

```
void  doFunction( void )                // ok
void  doFunction( ObjectRef aRef )      // ok
void  doFunction(ObjectRef aRef)        // wrong
void  doFunction( ObjectRef aRef)       // wrong
void  doFunction()                      // ok
```

Standard 31 *Use blank lines before and after block comments.*

Standard 32 *Use vertical alignment to indicate association.*

Standard 33 *Use spaces, not tabs.*

Rationale: Tab sizes vary between developers. When spaces are used, the alignment is maintained no matter where the file is edited.

4.6 Wrapping

Guideline 16 *No line of code should extend beyond column 78.*

Rationale: When the audience for the source and headers of a project may reach thousands, if not more, readability and continuity become prime factors for comprehension. Additionally, in a multi-developer environment, the potential for disjoint style is personified with no restrictions on column length.

Standard 34 *When wrapping lines, indent the continuation line past the current indent column.*

```
Int     val(2);

cout << "This is an example where I wrap "
   << val
   << " lines of code" << endl;
```

Standard 35 *Wrap assignments past the equal sign.*

```
ObjectMapConstIterator  aItr;

aItr = theMapOfObjects.begin();
```

Standard 36 *Wrap conditional expressions after the operators.*

```
if( theNameOfTheGame == aGameName &&
    theTimeBeingPlayed > aLimit )
{
    // ...
}
```

Standard 37 *Wrap* `for` *statements after the semi-colons.*

```
/* Example? */
```

Standard 38 *Wrap long function signatures with one parameter per line.*

```
void    ClassMethod::setValues
   (
      ObjectCref a1,
      ObjectCref a2,
      StringCref aName,
      IntCref    aValue,
      ...
   )
{
   ;
}
```

5 Naming Conventions

Standard 39 *Spell words using correct English spelling. For the most part, avoid abbreviations.*

Rationale: The semantics of a type are much better understood by the reader when they have names like "SpeakerCabinet" instead of "spkcb".

Guideline 17 *Make names clearly unique. Avoid similar-sounding names and similarly-spelled names.*

```
Int     aCount;
String  aName;
Object  aPerson;
```

– INSTEAD OF –

```
Int     x1;
String  x2;
Object  x3;
```

Standard 40 *Make all identifiers unique within a function.*

Standard 41 *Use mixed case to distinguish name segments instead of underscores.*

```
WellFormedObject        aObject;        // ok
non_standard_form       aObject;        // wrong
```

Standard 42 *Types start with an upper case letter and use mixed case to separate name segments. (i.e.* `String`*,* `BigCar`*).*

Standard 43 *Variables and objects names that are data members of a class start with a '*`the`*' and use mixed case thereafter:*

```
class Foo
{
   public:

      //

   protected:

      String        theName;
};
```

Standard 44 *Variables and objects names that are arguments or locals start with lower case '*`a`*'. (i.e.* `String aName;` *).*

Standard 45 *Pointer, reference and const reference types are created by postfixes* `Ptr`, `Cptr`, `Ref` *or* `Cref` *as appropriate. The macros* `DECLARE_TYPE` *and* `DECLARE_CLASS` *create correctly formed pointer and references types.*

```
#define DECLARE_TYPE( mydecl, mytype )    \
   typedef mydecl         mytype;         \
   typedef mytype *       mytype ## Ptr;  \
   typedef const mytype * mytype ## Cptr; \
   typedef mytype &       mytype ## Ref;  \
   typedef const mytype & mytype ## Cref

#define DECLARE_CLASS( tag )           \
   class   tag;                        \
   typedef tag *       tag ## Ptr;     \
   typedef const tag * tag ## Cptr;    \
   typedef tag &       tag ## Ref;     \
   typedef const tag & tag ## Cref
```

Rationale: The new `C++` casting syntax is used to do casts. It more accurately reflects the semantics of a cast, that is, conversion of one type to another by invocation of a conversion constructor. By typedef'ing the pointer and reference declarations this notation is easier to read.

Guideline 18 *Only use short variable names when they have limited scope and obvious meaning. Beware of them causing confusion.*

Standard 46 *Use capital letters to begin new name segments within the name.*

Guideline 19 *Name functions with verb-noun (verb object) combinations.*

Guideline 20 *Name variables and structures with noun, adjective noun combinations.*

Standard 47 *Accessor methods start with the word '*`get`*' and should be* `const`*.*

Standard 48 *Boolean accessor functions start with '*`is`*' and return* `bool`*.*

Standard 49 *Mutator procedures start with the word '*`set`*' and don't return values.*

```
class Foo
{
public:

   //
   // Accessor
   //
```

```
   StringCref   getName( void ) const;
   bool         isNameEmpty( void ) const;

   //
   // Mutator
   //

   void         setName( StringCref aName );

protected:

   String       theName;
};
```

Standard 50 *Factory instantiation functions start with the word '*`create`*'.*

```
ThreadPtr createThread( void );
```

Standard 51 *Factory destruction functions start with the word '*`destroy`*'.*

```
void destroyThread( ThreadPtr );
```

5.1 Files and Directories

Standard 52 *Use project names in source include statements:*

```
#if !defined(__SOMETHING_HPP)
#define (__SOMETHING_HPP)

#if !defined(__COMMON_HPP)
#include  <corelinux/Common.hpp>   // Correct
#include  <Common.hpp>   // While also correct,
   // there is too much assumption on environment
#endif

...

#endif
```

Guideline 21 *Name files like variables, describing the functions they contain. Long file names are encouraged.*

Standard 53 *Use* `.cpp` *and* `.hpp` *for class definition source and header file suffixes.*

Guideline 22 *Use* `.c` *and* `.h` *extensions for 'C' style source and header files. 'C' code is discouraged.*

Guideline 23 *Any procedural code written should be compiled in* `C++`*.*

Rationale: `C++` compilers provide much stricter type checking than C compilers. The stronger type checking is well worth using, even if the code does not take advantage of the object oriented features of `C++`.

Guideline 24 *Name directories like nested structures.*

6 Usage

Guideline 25 *There are no circumstances where* `goto` *is allowed.*

Guideline 26 *Avoid deep nesting of statements, parentheses, and structures.*

Guideline 27 *All assignments shall stand alone, unless a series of variables are being assigned to the same value.*

Standard 54 *Do not use comparisons in mathematical expressions.*

```
numberOfDays = ( isLeapYear() == TRUE ) + 28; // not OK
```

Guideline 28 *All non-boolean comparison expressions should use a comparison operator. Do not use implicit* `!= 0`*.*

```
REQUIRE( aObjectPtr != NULLPTR );       // good
REQUIRE( aObjectPtr );                  // bad
```

Guideline 29 *Avoid assignment in comparisons, except where the alternative is significantly more complex.*

Standard 55 *Use explicit casting, instead of the compiler default.*

```
Dword   aUnsignedValue(0);
Real    aRealValue(3.7);

aBigValue = Dword(aRealValue);
```

Also note that the class operator overloads should be used as a preference for upcasting and downcasting:

```
class Foo
{
public:

   //
   // Accessor
```

```
   //

   operator      Dword( void ) const
   {
      return Dword( theValue );
   }

   operator      Short( void ) const
   {
      return Short( theValue );
   }

   //
   // Mutator
   //

protected:

   Real          theValue;
};
```

Guideline 30 *Default to pre-increment and pre-decrement unless the post-increment/decrement operators are logically necessary.*

Guideline 31 *Minimize negative comparisons.*

Guideline 32 *Minimize use of the comma operator.*

6.1 Conditionals

Guideline 33 *Use* `if (`*cond*`) ... else` *rather than conditional expressions* `( (`*cond*`) ? : )` *if only to clarify the intended operation.*

Guideline 34 *Use nested if only to clarify the intended order of evaluation.*

```
if( foo() == true )
{
   if( bar() == true )
   {
   }
}
```

— VERSUS —

```
if( foo() && bar() )
{

}
```

Standard 56 *In a* `switch` *statement, make all cases independent by using break at the end of each. All switch statements should have a default. If all cases have been handled, then the default should never be* `NEVER_GET_HERE`*. This is also true for if...elseif...else blocks.*

Standard 57 *Use* `if...else` *for two alternative actions. Put the major action first.*

6.2 Loop Constructs

Guideline 35 *Count for loops in ascending.*

Standard 58 *Use for loops when the loop control needs initializing or recalculating; otherwise, use while.*

Standard 59 *Use* `while( 1 )` *to implement an infinite loop. Make its usage clear with comments.*

Guideline 36 *Be careful with the logic of do loops. Use* `do...while( !(...) )` *to loop until a comparison becomes true.*

Guideline 37 *Minimize use of* `break` *in loops. Only use it for abnormal escape.*

Guideline 38 *Use* `continue` *sparingly. Clearly comment why* `continue` *is used.*

6.3 Data

Standard 60 *Use* `NULLPTR` *with pointers only. (* `#define NULLPTR 0` *is declared in* `Types.hpp` *).*

Standard 61 *Don't rely on RTTI for dynamic and static casts.*

Guideline 39 *Use RTTI where appropriate.*

Rationale: The developer may not want to compile with RTTI for whatever reason. You should consider using templates if type reasoning is required. On the other hand, RTTI is part of the `C++` 14882 (1998E) standard.

Guideline 40 *Beware of operations with constants going out of range.*

Standard 62 *Use single-quoted characters for character constants, but never single-quote more than one character (or hex for non-printing).*

Standard 63 *Use* `sizeof` *rather than a constant.*

Standard 64 *Do not compare floating point numbers for equality.*

Standard 65 *Assign to all data members in* `operator=`*.*

Standard 66 *Check for assignment to* `this` *in* `operator=`*. If assignment to* `this` *is attempted, simply return from the function.*

```
MyClassRef  MyClass::operator=( MyClassCref aRef )
{
   if( this != &aRef )
   {
      //      do the assignment
      ...
   }
   else
   {
      ;   // do nothing
   }
   return *this;
}
```

Standard 67 *Make sure* `operator=` *invokes any parents'* `operator=`*, except with virtual inheritance.*

6.4 Constructors and Destructor

Standard 68 *Always define a default and copy constructor as well as an assignment operator for a class. Make these functions private if they should not be used.*

Rationale: The compiler will create these for you if you don't. It is the side effects and tracking down of such that is avoided with explicitly defining them.

Guideline 41 *Prefer initialization to assignment in constructors.*

```
MyClass::MyClass( void )
   :
   theAlpha(0),
   theBeta(0),
   theGamma(0)
{
   ; // do nothing
}
```

— INSTEAD OF —

```
MyClass::MyClass( void )
{
   theAlpha = 0;
   theBeta   = 0;
   theGamma = 0;
}
```

Standard 69 *Make destructor virtual in all polymorphic classes, or classes where the potential is high.*

Standard 70 *A constructor should put its object in a well-defined state. At the end of the constructor, the object must satisfy its class invariant.*

Standard 71 *A constructor that fails shall throw an exception.*

6.5 Initialization

Standard 72 *Explicitly initialize static data.*

Standard 73 *Initialize all variables at the time they are declared to the appropriate value. If the value is not yet known, initialize pointers to* `NULLPTR` *and simple types to zero.*

Standard 74 *List members in a constructor initialization list in order in which they are declared in the class header.*

6.6 Declaration

Guideline 42 *All data should be defined as close as possible to where it is needed.*

Standard 75 *Use floating point numbers only where necessary.*

Standard 76 *Use* `DECLARE_TYPE` *and* `DECLARE_CLASS` *macros to create new data types and new class declarations. When using templates with more than one parameter, a two-step process is needed:*

typedef to some temporary type name,
then `DECLARE_TYPE`*.*

Standard 77 *Do not use global data. Consider putting global information in the context of a static class.*

Standard 78 *Do not use unions.*

Standard 79 *Do not use bit structures.*

6.7 Programming

Guideline 43 *Make sure interface definitions are clear and sufficient.*

Guideline 44 *Defend against system, program and user errors. Heavy use of assertions (see Assertion.hpp) and exceptions are encouraged.*

Guideline 45 *Enable the user and the maintainer to find sufficient information to understand an error. Include enough diagnostic information to give an accurate picture of why the error occurred.*

Guideline 46 *Do not use arbitrary, predefined limits(e.g. on symbol table entries, user names, file names).*

Guideline 47 *Make code more testable by reducing complexity.*

Standard 80 *When coding, use exactly the same names as in the object model.*

Standard 81 *Use the same form in corresponding calls to* `new` *and* `delete` *(i.e.* `new Foo` *uses* `delete aFooPtr` *and* `new Foo[ 100 ]` *uses* `delete []` *aFooArray.*

Guideline 48 *Know what* `C++` *silently creates and calls (e.g. default constructor, copy constructor, assignment operator, address-of operators(const and not), and the destructor for a derived class where the base class's destructor is defined.)*

Standard 82 *Ensure that objects (both simple and class) are initialized before they are used.*

Standard 83 *Eradicate all compiler warnings. Set the compiler to the highest warning level. Any unavoidable warnings must be explicitly commented in the code. Unavoidable compiler warnings should be extremely rare.*

6.8 Class and Functions

Guideline 49 *Strive for class interfaces that are complete and minimal.*

Guideline 50 *The programmer should only have to look at the .hpp file to use a class.*

Standard 84 *Do not put data members in the public interface.*

Standard 85 *Pass and return objects be reference instead of value whenever possible.*

Standard 86 *If the passed object is not going to be modified then pass it as a const reference.*

Standard 87 *The keyword class will appear in the left most column. If declaring the class in the scope of a namespace, then class will be indented appropriately.*

Standard 88 *The member access controls appear flush with the class keyword.*

Standard 89 *Access controls that have no members may be omitted.*

Standard 90 *Access controls appear in the following order*

```
public:         //      Public method declarations
protected:      //      Protected method declarations
private:        //      Private method declarations
protected:      //      Protected class data members
private:        //      Private class data members
```

comments here are for clarification.

Standard 91 *The virtual or static declarations appear in the first indentation level from the class declaration.*

Standard 92 *The return type of a class method or the storage type of a class data member appear after the first tab position beyond the space were virtual applied.*

Standard 93 *Method identifiers follow the return type and should be reasonably lined with other method declarations.*

Standard 94 *In each access control group for methods, Constructors are followed by destructor, followed by operator overloads, followed by accessors followed by mutators.*

```
class MyClass
{

public:

   /// Default constructor

   MyClass( void )

   /**
      Copy constructor
      @param Object constant reference
    */

   MyClass( ObjectCref );
```

```
   /// Destructor

   virtual         ~MyClass( void );

   /**
      Equality operator overload
      @param MyClass constant reference
      @return bool - true if equal, false otherwise
    */

   bool  operator==( MyClassCref );

   //
   // Accessors
   //

   /**
      Get the number of MyClass instantiations.
      @return Int const reference to count
    */

   static    IntCref      getInstanceCount( void );

   /**
      Return the object data member
      @return Object const reference
    */

   ObjectCref   getObject( void ) const;

   //
   // Mutators
   //

   /**
      Sets the something thing
      @param Something const reference
    */

   virtual   void         setSomething( SomethingCref );

protected:

   /// Copy never allowed

   MyClass( MyClassCref ) throw(Assertion)
   {
```

```
      NEVER_GET_HERE;
   }

   /// Assignment operator denied

   MyClassRef   operator=( MyClassCref ) throw(Assertion)
   {
      NEVER_GET_HERE;
      return *this;
   }
private:

   // No private methods

protected:

   // No public data members

private:

   /// Class instance counter

   static    Int   theInstanceCount;
};
```

Guideline 51 *Don't return a reference, or pointer, when you must return an object, and don't return an object when you mean a reference.*

Guideline 52 *Avoid overloading on a pointer and a numerical type. (i.e.* `foo( char * )` *vs.* `foo( int )` *– a call to* `foo( 0 )` *is ambiguous).*

Guideline 53 *Use static classes to partition the global namespace. (dated with* `C++` *namespace)*

Standard 95 *Do not return handles to internal data from const member functions. If a handle must be returned, make it const.*

Standard 96 *Avoid member functions that return pointers or references to members less accessible than themselves. Use* `const` *!*

Standard 97 *Never return a reference to a local object or a de-referenced pointer initialized by new within the function.*

Rationale: Obviously, when a functions ends, the local object goes out of scope, and the reference is no longer valid. One might attempt to NEW the object in the function instead, but then who would call the corresponding DELETE?

Standard 98 *Use enums for integral class constants.*

Guideline 54 *Use inlining judiciously.*

Guideline 55 *Inlines cause code bloat, slow down compile times, eat up name space, and not all compilers handle them the same way.*

The following puts much of what we have said into an example

In MyClass.hpp:

```
#if     !defined(__MYCLASS_HPP)
#define __MYCLASS_HPP

/*
  CoreLinux++
  Copyright (C) 1999, 2000 CoreLinux Consortium

   The CoreLinux++ Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Library General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   The CoreLinux++ Library library is distributed in the hope that it will
   be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Library General Public License for more details.

   You should have received a copy of the GNU Library General Public
   License along with the GNU C Library; see the file COPYING.LIB.  If not,
   write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
   Boston, MA 02111-1307, USA.
*/


#include   <Common.hpp>

#if !defined(__LASTCLASS_HPP)
#include <LastClass.hpp>
#endif

DECLARE_TYPE( CORELINUX(Dword), Value );

/**
   MyClass adds to the functionality of the libcorelinux++ library by
   making things even better for users of the base class LastClass. It
   achieves this by doing more, better, faster, and cheaper!
*/
```

```
DECLARE_CLASS( MyClass );

class MyClass : public LastClass
{

public:

   /// Default constructor

   MyClass( void );

   /**
      Initializing constructor
      @param Value const reference
    */

   MyClass( ValueCref );

   /**
      Copy constructor
      @param MyClass const reference
    */

   MyClass( MyClassCref );

   /**
      This constructor is kind of an adapter
      for objects of that last class
      that may be hanging around.
      @param LastClass const reference
    */

   MyClass( LastClassCref );

   /// Destructor

   virtual   ~MyClass( void );


   /**
      Assignments operator
      @param MyClass const reference
      @returns MyClass reference
    */

   MyClassRef operator=( MyClasCref );
```

```
   /**
      Assignments operator
      @param LastClass const reference
      @returns MyClass reference
    */

   MyClassRef operator=( LastClassCref );

   /**
      Equality operators
      @param MyClass const reference
      @returns bool true if equal, false otherwise
    */

   bool       operator==( MyClassCref );

   /**
      Returns the internal value for reading
      @returns Value const reference
    */

   virtual  ValueCref   getValue( void ) const;

   /// In-line safe castdown to Value const reference

   operator ValueCref( void ) const
   {
      return this->getValue();
   }


   /**
      Change the internal value
      @param Value const reference
    */

   virtual  void setValue( ValueCref );


   //
   // Protected and Private methods follow public
   // methods
   //

protected:   // Not required if there are none
```

```
private:     // Not required if there are none


   //
   // Protected and Private data follow
   // protected and private methods (if there are any)
   //

protected:   // Not required if there are none

private:

   /// The internal value

   Value       theValue;
};

#endif  // if !defined(__MYCLASS_HPP)

/*
   Common rcs information do not modify
   $Author: dulimart $
   $Revision: 1.6 $
   $Date: 2000/09/22 23:44:22 $
   $Locker:  $
*/
```

In MyClass.cppa:

```
/**
  CoreLinux++
  Copyright (C) 1999, 2000 CoreLinux Consortium

   The CoreLinux++ Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Library General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   The CoreLinux++ Library library is distributed in the hope that it will
   be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Library General Public License for more details.

   You should have received a copy of the GNU Library General Public
   License along with the GNU C Library; see the file COPYING.LIB.  If not,
   write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
   Boston, MA 02111-1307, USA.
*/
```

```
#include   <Common.hpp>

#if !defined(__MYCLASS_HPP)
#include <MyClass.hpp>
#endif

#if !defined(__LASTCLASS_HPP)
#include <LastClass.hpp>
#endif

// Default constructor
// Uses initializer list

MyClass::MyClass( void )
   :
   LastClass(),
   theValue(0)
{
   ; // do nothing
}

// Constructor that sets the
// internal value

MyClass::MyClass( ValueCref aValue )
   :
   LastClass(),
   theValue(aValue)
{
   ; // do nothing
}

// Constructor which sets the internal
// value to the value of the passed
// reference.

MyClass::MyClass( MyClassCref aCref )
   :
   LastClass(aCref),
   theValue(aCref.getValue())
{
   ; // do nothing
}

// Constructor supplied to advance usage
// while minimizing code changes.
```

```
MyClass::MyClass( LastClassCref aCref )
   :
   LastClass(aCref),
   theValue(0)
{
   ; // do nothing
}

// Sets the internal state of the
// object before destructing

MyClass::~MyClass( void )
{
   theValue = 0;
}

// Assignment operator from another
// MyClass

MyClassRef MyClass::operator=( MyClasCref aRef )
{
   LastClass::operator=(aRef);
   if( this != &aRef )
   {
      theValue = aRef.getValue();
   }
   else
   {
      ; // do nothing
   }

   return *this;
}

// Migration assistant assignment operator
// from last implementation

MyClassRef MyClass::operator=( LastClassCref aRef )
{
   LastClass::operator=(aRef);

   return *this;
}

// Equality test
```

```
bool MyClass::operator==( MyClassCref aRef )
{
   bool isMe(false);

   if( this == &aRef )
   {
      isMe = true;
   }
   else
   {
      ; // do nothing
   }

   return isMe;
}

// Accessor : return theValue

ValueCref   MyClass::getValue( void ) const
{
   return theValue;
}

// Mutator : set theValue to aValueRef

void  MyClass::setValue( ValueCref aValueRef )
{
   theValue = aValueRef;
}

/*
   Common rcs information do not modify
   $Author: dulimart $
   $Revision: 1.6 $
   $Date: 2000/09/22 23:44:22 $
   $Locker:  $
*/
```

6.9 Templates and Template Functions

As per sourceware.cygnus.com with the standards in this guide applied

Standard 99 *Template function indentation should follow the form*

```
template<class T> void  doSomethingFunction( args )
{
   //
}
```

-NOT-

```
template<class T> void template_function( args ) {};
```

Rationale: In class definitions, without indentation whitespace is needed both above and below the declaration to distinguish it visually from other members.

Standard 100 *Template class indentation should follow the form with what is not shown following the Section 6.8 on page 20.*

```
template<class T> class Base {

   public: // Types:

};
```

-NOT-

```
template<class T> class Base { public: // Types:
};
```

6.10 Inheritance

Guideline 56 *Make sure public inheritance models 'is-a'.*

Guideline 57 *Differentiate between inheritance of the interface and of the implementation, and prefer delegation to implementation inheritance.*

Rationale: Inheritance of the interface only is forced by making a function pure virtual– its implementation must be defined by the derived class. Inheritance of implementation occurs when the function is declared as simple virtual–derived classes may or may not override the implementation.

Standard 101 *Never redefine an inherited non-virtual function.*

Standard 102 *Never redefine an inherited default parameter value.*

Standard 103 *A virtual function cannot strengthen a pre-condition or weaken a postcondition. Use the* `BASE_INVARIANT` *in the* `INVARIANT` *of the derived class.*

Guideline 58 *Use private inheritance judiciously.*

Guideline 59 *Differentiate between inheritance and templates.*

Rationale: If the type of the object being manipulated does not affect the behavior or the class, then a template will do. However, if the type of the object DOES affect the behavior, then virtual functions should be used through inheritance.

6.11 Object-Oriented Considerations

Standard 104 *Factor our common code into an ancestor.*

Guideline 60 *Encapsulate external code within an operation or class.*

Guideline 61 *Keep member functions small, coherent and consistent.*

Guideline 62 *Separate policy member functions (e.g. error and status checkers) from implementation member functions (computational).*

Standard 105 *Do not use indirect function calls unless absolutely necessary.*

Standard 106 *Write member functions for all combinations of input conditions. Avoid using modes to distinguish between conditions. Use member function overloading instead.*

Standard 107 *Avoid case statements on object type; use member functions instead.*

Standard 108 *Keep internal class structure hidden from other classes. Do not use global or public data.*

Standard 109 *Do not traverse multiple links or member functions in a single statement. Invoke each member function via a temporary pointer or reference.*

Standard 110 *Use* `const` *wherever a function, parameter or return value will not change.*

Guideline 63 *All accessor functions should be* `const`*.*

6.12 Error Handling

Standard 111 *Use exceptions rather than returning a failure status.*

Standard 112 *Use assertions (*`REQUIRE`*,* `ENSURE`*,* `CHECK`*,* `NEVER_GET_HERE`*) liberally.*

Standard 113 `ENSURE` *or* `CHECK_INVARIANT`*, should be the last line before a return in a function.*

Standard 114 *Derived classes must maintain the base class's invariant. This is to be verified by a call to* `BASE_INVARIANT(BaseClassName)`*.*

6.13 Exception Specification

Standard 115 *Exceptions are to be specified in the signature of a class method declaration and definition.*

Standard 116 *Exception type hierarchies should be created which reflect the domain.*

Rationale: It is easier to reason with an Exception type rather than figuring out what the exception was through a text message.

```
class Bridge : public CoreLinuxObject {
   public:
   /**
      Bridge assignment operator
      @param Bridge constant reference
      @return Reference to *this
      @exception Exception
    */

   BridgeRef operator=( BridgeCref ) throw(Exception);
};

//
// can't be reasoned with compared to:
//
class BridgeException : public Exception {..};
class Bridge : public CoreLinuxObject
{
   public:

   /**
      Bridge assignment operator
      @param Bridge constant reference
      @return Reference to *this
      @exception BridgeException
    */
    BridgeRef operator=( BridgeCref ) throw(BridgeException);
};

//
// at a minimum, and
//
class BridgeException : public Exception {..};
class SetImplementationException : public BridgeException {..};
class Bridge : public CoreLinuxObject
{
   public:
```

```
    /**
       Bridge assignment operator
       @param Bridge constant reference
       @return Reference to *this
       @exception SetImplementationException
     */
    BridgeRef operator=( BridgeCref ) throw(SetImplementationException);
};
//
// being the most useful
//
```

7 File Layout

Guideline 64 *Use functional cohesion to group similarly items.*

Guideline 65 *Source files may be split up along boundaries between class administrator, accessor, mutator and provider functions.*

Standard 117 *A class shall have a single header file.*

Guideline 66 *Lay out data files to reflect the systems they serve.*

Standard 118 *Do not reserve memory in header files.*

Standard 119 *Minimize header file interdependence.*

Guideline 67 *Keep function length in code files to within one or two pages (100 lines).*

Guideline 68 *Keep modules small. Each module should be functionally cohesive and should be as small as possible. Modules should not exceed 10-15 pages in length.*

Guideline 69 *Classes with a large number of functions should break the implementation into several* `.CPP` *files. These files should be functionally cohesive.*

7.1 Header File Layout

Standard 120 *All headers must be wrapped to prevent multiple inclusion.*

```
#if !defined(__MYHEADER_HPP)
#define __MYHEADER_HPP
...
#endif /* __MYHEADER_HPP */
```

–OR–

```
#if !defined(__MYHEADER_HPP)
#define  __MYHEADER_HPP
...
#endif /* __MYHEADER_HPP */
```

Standard 121 *The wrapper must be the name of the file, prefixed with a double underscore* `__` *and followed by* `_HPP`*. (i.e.* `__COMMON_HPP`*).*

Standard 122 *public, protected and private line appear between column 1 and 3 inclusive within a class definition.*

Standard 123 *Types, returns, member names, functions must be lined up consistently in header.*

Standard 124 *Trailing comments must be aligned within the module.*

8 Linux Special Care and Handling

In some cases, you may need to handle situations where a conflict arises between the use of both your class and header files, and installed system or third party files. This may result in either failed compilation or run-time library resolution problems.

This section will keep track of where we, or users, have found problems with guidelines that resolve the situation.

Guideline 70 *Order system, or third party, includes before local header files.*

Guideline 71 *Make use of namespaces defined by the provider.*

References

Grady Booch. *Object Oriented Analysis and Design With Applications*. Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

The Corelinux Consortium. *The Corelinux C++ Coding Standards*. The Corelinux Consortium, 1.3 edition, May 2000a. http://corelinux.sourceforge.net/cppstnd.php.

The Corelinux Consortium. *The Corelinux Object Oriented Design Standards*. The Corelinux Consortium, 1.3 edition, May 2000b.

Phillip B. Crosby. *Quality Is Free*. McGraw-Hill, New York, NY., 10020, 1976.

FSF. *GNU Autoconf Manual*. FSF, 2.13 edition, 1999.

FSF. *GNU Automake Manual*. FSF, 1.4 edition, 2000a.

FSF. *GNU Libtool Manual*. FSF, 1.3.4 edition, 2000b.

D.E. Knuth. Structured programming with goto's. *ACM Computing Surveys*, Vol 6(No. 4), December 1974.

Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1988.

Inc. Taligent. *The Taligent Guide to Well-Mannered Object-Oriented Programs*. Taligent Inc., Cupertino, CA., 1994.