C++ Coding Standard

Last Modified: 2004-01-12

tmh@possibility.com / My Home Page

Changed by Stefan Eilers (eilers@sra.uni-hannover.de)

Using this Standard. If you want to make a local copy of this standard and use it as your own you are perfectly free to do so. That's why I made it!

Names

Make Names Fit

A name is the result of a long deep thought process about the ecology it lives in. Only a programmer who understands the system as a whole can create a name that "fits" with the system. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected.

If you find all your names could be Thing and DoIt then you should probably revisit your design.

Class Names

Name the class after what it is. If you can't think of what it is that is a clue you have not thought through the design well enough. Compound names of over three words are a clue your design may be confusing various entities in your system. Revisit your design. Try a CRC card session to see if your objects have more responsibilities than they should.
Avoid the temptation of bringing the name of the class a class derives from into the derived class's name. A class should stand on its own. It doesn't matter what it derives from.
Suffixes are sometimes helpful. For example, if your system uses agents then naming something DownloadAgent conveys real information.

Method and Function Names

Usually every method and function performs an action, so the name should make clear what it does: CheckForErrors() instead of ErrorCheck(), DumpDataToFile() instead of DataFile(). This will also make functions and data objects more distinguishable.

Classes are often nouns. By making function names verbs and following other naming conventions programs can be read more naturally.

Suffixes are sometimes useful:
   o *Max* - to mean the maximum value something can have.
   o *Cnt* - the current count of a running count variable.
   o *Key* - key value.

For example: RetryMax to mean the maximum number of retries, RetryCnt to mean the current retry count.

Prefixes are sometimes useful:
   o *Is* - to ask a question about something. Whenever someone sees *Is* they will know it's a question.
   o *Get* - get a value.
   o *Set* - set a value.

For example: IsHitRetryLimit.

Include Units in Names

If a variable represents time, weight, or some other unit then include the unit in the name so developers can more easily spot problems. For example:

```
uint32 mTimeoutMsecs;
uint32 mMyWeightLbs;
```

Better yet is to make a variable into a class so bad conversions can be caught.

No All Upper Case Abbreviations

When confronted with a situation where you could use an all upper case abbreviation instead use an initial upper case letter followed by all lower case letters. No matter what.

Justification

People seem to have very different intuitions when making names containing abbreviations. It's best to settle on one strategy so the names are absolutely predictable.

Take for example *NetworkABCKey*. Notice how the C from ABC and K from key are confused. Some people don't mind this and others just hate it so you'll find different policies in different code so you never know what to call something.

Example

```
class FluidOz            // NOT FluidOZ
class NetworkAbcKey      // NOT NetworkABCKey
```

Class Names

Use upper case letters as word separators, lower case for the rest of a word
First character in a name is upper case
No underbars ('_')

Justification

Of all the different naming strategies many people found this one the best compromise.

Example

```
class NameOneTwo
```

```
class Name
```

Class Library Names

Now that name spaces are becoming more widely implemented, name spaces should be used to prevent class name conflicts among libraries from different vendors and groups.
When not using name spaces, it's common to prevent class name clashes by prefixing class names with a unique string. Two characters is sufficient, but a longer length is fine.

Example

John Johnson's complete data structure library could use *JJ* as a prefix, so classes would be:

```
class JjLinkList
{
}
```

Method Names

Use the same rule as for class names. BUT: The first char in lower case and we don't use any namespace-prefix !

Justification

Of all the different naming strategies many people found this one the best compromise.

Example

```
class JjNameOneTwo
{
public:
   int              doIt();
   void             handleError();
}
```

Class Attribute Names

Attribute names should be prepended with the characters 'm_'.
After the 'm_' use the same rules as for class names.
'm_' always precedes other name modifiers like 'p' for pointer.

Justification

Prepending 'm_' prevents any conflict with method names. Often your methods and attribute names will be similar, especially for accessors.

Example

```
class NameOneTwo
{
public:
  int                varAbc();
  int                errorNumber();
private:
  int                m_varAbc;
  int                m_errorNumber;
  String*            m_pname;
}
```

Method Argument Names

The first character should be lower case.
All word beginnings after the first letter should be upper case as with class names.

Justification

You can always tell which variables are passed in variables.
You can use names similar to class names without conflicting with class names.

Example

```
class NameOneTwo
{
public:
  int   startYourEngines( Engine& rSomeEngine,
                          Engine& rAnotherEngine );
}
```

Variable Names on the Stack

use all lower case letters
use '_' as the word separator.

Justification

With this approach the scope of the variable is clear in the code.
Now all variables look different and are identifiable in the code.

Example

```
int NameOneTwo::HandleError(int errorNumber)
{
   int            error = OsErr();
   Time           time_of_error;
   ErrorProcessor error_processor;
}
```

Pointer Variables

> pointers should be prepended by a 'p' in most cases
> place the * close to the pointer type not the variable name

Justification

> The idea is that the difference between a pointer, object, and a
> reference to an object is important for understanding the code,
> especially in C++ where -> can be overloaded, and casting and
> copy semantics are important.
> Pointers really are a change of type so the * belongs near the type.
> One reservation with this policy relates to declaring multiple
> variables with the same type on the same line. In C++ the pointer
> modifier only applies to the closest variable, not all of them, which
> can be very confusing, especially for newbies. You want to have one
> declaration per line anyway so you can document each variable.

Example

```
String* pName= new String;

String* pName, name, address; // note, only pName is a pointer.
```

Reference Variables and Functions Returning References

> References should be prepended with 'r'.

Justification

> The difference between variable types is clarified.

It establishes the difference between a method returning a modifiable object and the same method name returning a non-modifiable object.

Example

```
class Test
{
public:
  void DoSomething( StatusInfo& rStatus );

  StatusInfo& rStatus();
  const StatusInfo&  Status() const;

private:
  StatusInfo& m_rStatus;
}
```

Global Variables

Global variables should be prepended with a 'g_'.

Justification

It's important to know the scope of a variable.

Example

```
Logger  g_Log;
Logger* g_pLog;
```

Global Constants

Global constants should be all caps with '_' separators.

Justification

It's tradition for global constants to named this way. You must be careful to not conflict with other global *#define*s and enum labels.

Example

```
const int A_GLOBAL_CONSTANT= 5;
```

Static Variables

Static variables may be prepended with 's'.

Justification

It's important to know the scope of a variable.

Example

```
class Test
{
public:
private:
   static StatusInfo m_sStatus;
}
```

Type Names

When possible for types based on native types make a typedef. Typedef names should use the same naming policy as for a class with the word *Type* appended.

Justification

Of all the different naming strategies many people found this one the best compromise.
Types are things so should use upper case letters. *Type* is appended to make it clear this is not a class.

Example

```
typedef uint16  ModuleType;
typedef uint32  SystemType;
```

#define and Macro Names

Put #defines and macros in all upper using '_' separators.

Justification

This makes it very clear that the value is not alterable and in the case of macros, makes it clear that you are using a construct that requires care.

Some subtle errors can occur when macro names and enum labels use the same name.

Example

```
#define MAX(a,b) blah
#define IS_ERR(err) blah
```

C Function Names

In a C++ project there should be very few C functions.
For C functions use the GNU convention of all lower case letters with '_' as the word delimiter.

Justification

It makes C functions very different from any C++ related names.

Example

```
int some_bloody_function()
{
}
```

Enum Names

Labels All Upper Case with '_' Word Separators

This is the standard rule for enum labels.

Example

```
enum PinStateType
{
```

```
    PIN_OFF,
    PIN_ON
  };
```

Enums as Constants without Class Scoping

Sometimes people use enums as constants. When an enum is not embedded in a class make sure you use some sort of differentiating name before the label so as to prevent name clashes.

Example

```
    enum PinStateType
    {
      PIN_OFF,
      PIN_ON
    };
```
If *PIN* was not prepended a conflict would occur as OFF and ON are probably already defined.

Enums with Class Scoping

Just name the enum items what you wish and always qualify with the class name: Aclass::PIN_OFF.

Make a Label for an Error State

It's often useful to be able to say an enum is not in any of its *valid* states. Make a label for an uninitialized or error state. Make it the first label if possible.

Example

```
 enum { STATE_ERR,  STATE_OPEN, STATE_RUNNING, STATE_DYING };
```

Required Methods for a Class

To be good citizens almost all classes should implement the following methods. If you don't have to define and implement any of the "required" methods they should still be represented in your class definition as comments.

    Default Constructor

If your class needs a constructor, make sure to provide one. You need one if during the operation of the class it creates something or does something that needs to be undone when the object dies. This includes creating memory, opening file descriptors, opening transactions etc.

If the default constructor is sufficient add a comment indicating that the compiler-generated version will be used.

If your default constructor has one or more optional arguments, add a comment indicating that it still functions as the default constructor.

Virtual Destructor

If your class is intended to be derived from by other classes then make the destructor virtual.

Copy Constructor

If your class is copyable, either define a copy constructor and assignment operator or add a comment indicating that the compiler-generated versions will be used.

If your class objects should not be copied, make the copy constructor and assignment operator private and don't define bodies for them. If you don't know whether the class objects should be copyable, then assume not unless and until the copy operations are needed.

Assignment Operator

If your class is assignable, either define a assignment operator or add a comment indicating that the compiler-generated versions will be used.

If your objects should not be assigned, make the assignment operator private and don't define bodies for them. If you don't know whether the class objects should be assignable, then assume not.

Justification

Virtual destructors ensure objects will be completely destructed regardless of inheritance depth. You don't have to use a virtual destructor when:
  - You don't expect a class to have descendants.
  - The overhead of virtualness would be too much.
  - An object must have a certain data layout and size.

A default constructor allows an object to be used in an array. The copy constructor and assignment operator ensure an object is always properly constructed.

The Law of The Big Three

A class with any of (destructor, assignment operator, copy constructor) generally needs all 3. For more information see http://www.parashift.com/c++-faq-lite/coding-standards.html#[25.9].

Example

The default class template with all required methods. An example using default values:

```
class Planet
{
public:
  // The following is the default constructor if
  // no arguments are supplied:
  //
  Planet(int radius= 5);

  // Use compiler-generated copy constructor, assignment, and destructor.
  // Planet(const Planet&);
  // Planet& operator=(const Planet&);
  // ~Planet();
};
```

Braces {} Policy

Brace Placement

Of the three major brace placement strategies two are acceptable, with the first one listed being preferable:

Place brace under and inline with keywords:

```
if (condition)          while (condition)
{                       {
      …                           …
}                       }
```

Traditional Unix policy of placing the initial brace on the same line as the keyword and the trailing brace inline on its own line with the keyword:

```
if (condition) {          while (condition) {
     ...                       ...
}                         }
```

Justification

Another religious issue of great debate solved by compromise. Either form is acceptable, many people, however, find the first form more pleasant. Why is the topic of many psychological studies.

There are more reasons than psychological for preferring the first style. If you use an editor (such as vi) that supports brace matching, the first is a much better style. Why? Let's say you have a large block of code and want to know where the block ends. You move to the first brace hit a key and the editor finds the matching brace. Example:

```
if (very_long_condition && second_very_long_condition)
{
        ...
}
else if (...)
{
        ...
}
```

To move from block to block you just need to use cursor down and your brace matching key. No need to move to the end of the line to match a brace then jerk back and forth.

When Braces are Needed

All if, while and do statements must either have braces or be on a single line.

Always Uses Braces Form

All if, while and do statements require braces even if there is only a single statement within the braces. For example:

```
if (1 == somevalue)
{
   somevalue = 2;
```

```
 }
```

Justification

It ensures that when someone adds a line of code later there are already braces and they don't forget. It provides a more consistent look. This doesn't affect execution speed. It's easy to do.

One Line Form

```
 if (1 == somevalue) somevalue = 2;
```

Justification

It provides safety when adding new lines while maintainng a compact readable form.

Add Comments to Closing Braces

Adding a comment to closing braces can help when you are reading code because you don't have to find the begin brace to know what is going on.

```
 while(1)
 {
   if (valid)
   {

   } // if valid
   else
   {
   } // not valid

 } // end forever
```

Comment blocks of Code

 It is good to place comments before code blocks which seems to be complicated and describe them in a general manner:

[...]

```
// This if clause does the following thing:
// …
// …
if ( .. )
{
         …
}
```

Consider Screen Size Limits

Some people like blocks to fit within a common screen size so scrolling is not necessary when reading code.

Indentation/Tabs/Space Policy

Indent using 8 (!) spaces for each level.
Do not use tabs, use spaces. Most editors can substitute spaces for tabs.
Tabs should be fixed at 8 spaces. Don't set tabs to a different spacing, uses spaces instead.
Indent as much as needed, but no more. There are no arbitrary rules as to the maximum indenting level. If the indenting level is more than 4 or 5 levels you may think about factoring out code.

Justification

If it is a problem to show the text without scrolling horizontally, using 8 speces for indentation, you have too many indentation levels. Changing a tab to be less than 8 spaces is a problem because that setting is usually local. When someone prints the source tabs will be 8 characters and the code will look horrible. Same for people using other editors. Which is why we use spaces...
Especially if you code several hours, you see the advantage of 8 spaces instead 4 or 2 ..
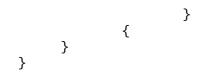When people using different tab settings the code is impossible to read or print, which is why spaces are preferable to tabs.
Nobody can ever agree on the correct number of spaces, just be consistent.
As much as people would like to limit the maximum indentation levels it never seems to work in general. We'll trust that programmers will choose wisely how deep to nest code.

Example

```
void
func()
{
        if (something bad)
        {
                if (another thing bad)
                {
                        while (more input)
                        {
```

```
                    }
              {
         }
     }
```

Parens *()* with Key Words and Functions Policy

Do not put parens next to keywords. Put a space between.
Do put parens next to function names.
Do not use parens in return statements when it's not necessary.

Justification

Keywords are not functions. By putting parens next to keywords keywords and function names are made to look alike.

Example

```
if (condition)
{
}

while (condition)
{
}

strcpy(s, s1);

return 1;
```

Class Layout

A common class layout is critical from a code comprehension point of view and for automatically generating documentation. C++ programmers, through a new set of tools, can enjoy the same level generated documentation Java programmers take for granted.

Class and Method Documentation

11.05.2004

It is recommended a program like DOXYGEN be used to document C++ classes, method, variables, functions, and macros. The documentation can be extracted and put in places in a common area for all programmers to access. This saves programmers having to read through class headers. Documentation generation should be integrated with the build system where possible.

Template

Please use the following template when creating a new class.

```
/**
 * A one line description of the class.
 *
 * #include "XX.h" <BR>
 * -llib
 *
 * A longer description.
 *
 * @see something
 */

#ifndef XX_h
#define XX_h

// SYSTEM INCLUDES
//

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//

// FORWARD REFERENCES
//


class XX
{
public:
// LIFECYCLE

   /**
    * Default constructor.
    */
   XX(void);

   /**
    * Copy constructor.
    *
    * @param from The value to copy to this object.
    */
   XX(const XX& from);
```

```
  /**
   * Destructor.
   */
  ~XX(void);

// OPERATORS

  /**
   * Assignment operator.
   *
   * @param from THe value to assign to this object.
   *
   * @return A reference to this object.
   */
  XX&               operator=(XX& from);

// OPERATIONS
// ACCESS
// INQUIRY

protected:
private:
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif  // _XX_h_
```

Required Methods Placeholders

The template has placeholders for [required methods]. You can delete them or implement them.

Ordering is: public, protected, private

Notice that the public interface is placed first in the class, protected next, and private last. The reasons are:

> programmers should care about a class's interface more than implementation
> when programmers need to use a class they need the interface not the implementation

It makes sense then to have the interface first. Placing implementation, the private section, first is a historical accident as the first examples used the private first layout. Over time emphasis has switched deemphasizing a class's interface over implementation details.

LIFECYCLE

The life cycle section is for methods that control the life cycle of an object. Typically these methods include constructors, destructors, and state machine methods.

OPERATORS

Place all operators in this section.

OPERATIONS

Place the bulk of a class's non access and inquiry method methods here. A programmer will look here for the meat of a class's interface.

ACCESS

Place attribute accessors here.

INQUIRY

These are the *Is** methods. Whenever you have a question to ask about an object it can be asked via in *Is* method. For example: IsOpen() will indicate if the object is open. A good strategy is instead of making a lot of access methods you can turn them around to be questions about the object thus reducing the exposure of internal structure. Without the IsOpen() method we might have had to do: if (STATE_OPEN == State()) which is much uglier.

What should go in public/protected/private?

Public Section

Only put an object's interface in the public section. DO NOT expose any private data items in the public section. At least encapsulate access via access methods. Ideally your method interface should make most access methods unnecessary. Do not put data in the public interface.

Protected and Private Section

What should go into the protected section versus the private section is always a matter of debate.

All Protected

Some say there should be no private section and everything not in the public section should go in the protected section. After all, we should allow all our children to change anything they wish.

All Private

Another camp says by making the public interface virtual any derived class can change behavior without mucking with internals.

Wishy Washy

Rationally decide where elements should go and put them there. Not very helpful.

And the Winner Is...

Keeping everything all private seems the easiest approach. By making the public methods virtual flexibility is preserved.

Method Layout

The approach used is to place a comment block before each method that can be extracted by a tool and be made part of the class documentation. Here we'll use DOXYGEN which supports the Javadoc format. See the DOXYGEN documentation for a list of attributes supported by the document generator.

Method Header

Every parameter should be documented. Every return code should be documented. All exceptions should be documented. Use complete sentences when describing attributes. Make sure to think about what other resources developers may need and encode them in with the @see attributes.

```
/**
 * Assignment operator.
 *
 * @param val The value to assign to this object.
 *
 * @return A reference to this object.
 */
XX&                operator=(XX& val);
```

Additional Sections

In addition to the standard attribute set, the following sections can be included in the documentation:

1. PRECONDITION
   Document what must have happened for the object to be in a state where the method can be called.
2. WARNING
   Document anything unusual users should know about this method.
3. LOCK REQUIRED
   Some methods require a semaphore be acquired before using the method. When this is the case use lock required and specify the name of the lock.
4. EXAMPLES
   Include exampes of how to use a method. A picture says a 1000 words, a good example answers a 1000 questions.

For example:

```
/**
 * Copy one string to another.
 *
 * PRECONDITION

 * REQUIRE(from != 0)
 * REQUIRE(to != 0)
 *
 * WARNING

 * The to buffer must be long enough to hold
 * the entire from buffer.
 *
 * EXAMPLES

 *

 * strcpy(somebuf, "test")
 *


 *
 * @param from The string to copy.
 * @param to The buffer to copy the string to.
 *
 * @return void
 */
void  strcpy(const char* from, char* to);
```

Common Exception Sections

If the same exceptions are being used in a number of methods, then the exceptions can be documented once in the class header and referred to from the method documentation.

Formatting Methods with Multiple Arguments

11.05.2004

We should try and make methods have as few parameters as possible. If you find yourself passing the same variables to every method then that variable should probably be part of the class. When a method does have a lot of parameters format it like this:

```
int          AnyMethod( int  arg1, int  arg2,
                        int  arg3, int  arg4);
```

Block Comments

Use comments on starting and ending a Block:

```
{
  // Block1  (meaningful comment about Block1)
  ... some code

  {
    // Block2  (meaningful comment about Block2)
    ... some code
  } // End Block2

} // End Block1
```

This may make block matching much easier to spot when you don't have an intelligent editor.

Think About What Work to do in Constructors

Should you do work that can fail in constructors? If you have a compiler that does not support exceptions (or thread safe exceptions if it matters to you) then the answer is definitely no. Go directly to [Do Work in Open](#). If your compiler supports exception then go to [Do Work in Constructor](#). There are still [reasons](#) to use an Open method even with exceptions.

Comment: Due to the fact that exceptions are very inefficient in C++, you should not use it, if possible ! (se)

Do Work in Open

Do not do any real work in an object's constructor. Inside a constructor initialize variables only and/or do only actions that can't fail.

Create an Open() method for an object which completes construction.
Open() should be called after object instantiation.

Example

```
class Device
{
public:
  Device()   { /* initialize and other stuff */ }
  int Open()  { return FAIL; }
};

Device dev;
if (FAIL == dev.Open()) exit(1);
```

Use Open Reasons

1. It is difficult to write exception safe code in constructor. It's possible
   to throw an exception and not destruct objects allocated in the
   constructor. Use of auto_ptr can help prevent this problem.
2. Some compilers do not support thread safe exceptions on all
   platforms.
3. Virtual methods are not available in base classes. If the base class is
   expecting a virtual method implemented by derived classes to be
   available during construction then initialization must follow
   construction. This is common in frameworks.
4. Larger scale state machines may dictate when initialization should
   occur. An object may contain numerous other objects that may have
   complex initialization conditions. In this case we could wait to
   construct objects but then we always have to worry about null
   pointers.
5. If deletion is needed to free resources we still may want to keep the
   state around for debugging or statistics or as a supplier of
   information for other objects.

Do Work in Constructor

With exceptions work done in the constructor can signal failure so it is fine
to perform real work in the constructor. This is the guru endorsed
approach as a matter of fact. But there are reasons to still use an open
style approach.

The constructor code must still be very careful not to leak resources in the
constructor. It's possible to throw an exception and not destruct objects
allocated in the constructor.

There is a pattern called Resource Acquisition as Initialization that says all
initialization is performed in the constructor and released in the

destructor. The idea is that this is a safer approach because it should reduce resource leaks.

Be Careful Throwing Exceptions in Destructors

An object is presumably created to do something. Some of the changes made by an object should persist after an object dies (is destructed) and some changes should not. Take an object implementing a SQL query. If a database field is updated via the SQL object then that change should persist after the SQL objects dies. To do its work the SQL object probably created a database connection and allocated a bunch of memory. When the SQL object dies we want to close the database connection and deallocate the memory, otherwise if a lot of SQL objects are created we will run out of database connections and/or memory.

The logic might look like:

```
Sql::~Sql()
{
   delete connection;
   delete buffer;
}
```

Let's say an exception is thrown while deleting the database connection. Will the buffer be deleted? No. Exceptions are basically non-local gotos with stack cleanup. The code for deleting the buffer will never be executed creating a gaping resource leak.

Special care must be taken to catch exceptions which may occur during object destruction. Special care must also be taken to fully destruct an object when it throws an exception.

Prototype Source File

```
#include "XX.h"                    // class implemented


///////////////////////////// PUBLIC ///////////////////////////////////////
```

11.05.2004

```
//=========================== LIFECYCLE
====================================

XX::XX()
{
}// XX

XX::XX(const XX&)
{
}// XX

XX::~XX()
{
}// ~XX


//=========================== OPERATORS
====================================

XX&
XX::operator=(XX&);
{
   return *this;

}// =

//=========================== OPERATIONS
====================================
//=========================== ACESS
====================================
//=========================== INQUIRY
====================================
//////////////////////////// PROTECTED  /////////////////////////////////

//////////////////////////// PRIVATE    /////////////////////////////////
```

Use of Namespaces

Namespaces are now commonly implemented by compilers. They should be used if you are sure your compiler supports them completely. I don't have a lot of experience with C++ namespaces in a project setting so this section is very thin.

Naming Policy

There are two basic strategies for naming: root that name at some naming authority, like the company name and division name; try and make names globally independent.

Don't Globally Define using

Don't place "using namespace" directive at global scope in a header file. This can cause lots of magic invisible conflicts that are hard to track. Keep using statements to implementation files.

Make Functions Reentrant

Functions should not keep static variables that prevent a function from being reentrant. Functions can declare variables static. Some C library functions in the past, for example, kept a static buffer to use a temporary work area. Problems happen when the function is called one or more times at the same time. This can happen when multiple tasks are used or say from a signal handler. Using the static buffer caused results to overlap and become corrupted.

The moral is make your functions reentrant by not using static variables in a function. Besides, every machine has 128MB of RAM now so we don't worry about buffer space any more :-)

To Use Enums or Not to Use Enums

C++ allows constant variables, which should deprecate the use of enums as constants. Unfortunately, in most compilers constants take space. Some compilers will remove constants, but not all. Constants taking space precludes them from being used in tight memory environments like embedded systems. Workstation users should use constants and ignore the rest of this discussion.

In general enums are preferred to *#define* as enums are understood by the debugger.

Be aware enums are not of a guaranteed size. So if you have a type that can take a known range of values and it is transported in a message you can't use an enum as the type. Use the correct integer size and use constants or *#define*. Casting between integers and enums is very error prone as you could cast a value not in the enum.

A C++ Workaround

C++ allows static class variables. These variables are available anywhere and only the expected amount of space is taken.

Example

```
class Variables
{
public:
   static const int   A_VARIABLE;
   static const int   B_VARIABLE;
   static const int   C_VARIABLE;
}
```

Use Header File Guards

Include files should protect against multiple inclusion through the use of macros that "guard" the files.

When Not Using Namespces

```
#ifndef filename_h
#define filename_h

#endif
```

The new line after the endif if is required by some compilers.

When Using Namespaces

If namespaces are used then to be completely safe:

```
#ifndef namespace_filename_h
#define namespace_filename_h

#endif
```

1. Replace *filename* with the name of the file being guarded. This should usually be the name of class contained in the file. Use the exact class name. Some standards say use all upper case. This is a mistake because someone could actually name a class the same as yours but using all upper letters. If the files end up be included together one file will prevent the other from being included and you will be one very confused puppy. It has happened!
2. Most standards put a leading _ and trailing _. This is no longer valid as the C++ standard reserves leading _ to compiler writers.
3. When the include file is not for a class then the file name should be used as the guard name.

4. Compilers differ on how comments are handled on preprocessor directives. Historically many compilers have not accepted comments on preprocessor directives.
5. Historically many compilers require a new line after last endif.

A Line Should Not Exceed 78 Characters

Lines should not exceed 78 characters.

Justification

Even though with big monitors we stretch windows wide our printers can only print so wide. And we still need to print code.
The wider the window the fewer windows we can have on a screen. More windows is better than wider windows.
We even view and print diff output correctly on all terminals and printers.

If Then Else Formatting

Layout

It's up to the programmer. Different bracing styles will yield slightly different looks. One common approach is:

```
if (condition)          // Comment
{
}
else if (condition)     // Comment
{
}
else                    // Comment
{
}
```

If you have *else if* statements then it is usually a good idea to always have an else block for finding unhandled cases. Maybe put a log message in the else even if there is no corrective action taken.

Condition Format

Always put the constant on the left hand side of an equality/inequality comparison. For example:

if (6 == errorNum) ...

One reason is that if you leave out one of the = signs, the compiler will find the error for you. A second reason is that it puts the value you are looking for right up front where you can find it instead of buried at the end of your expression. It takes a little time to get used to this format, but then it really gets useful.

---

switch Formatting

Falling through a case statement into the next case statement shall be permitted as long as a comment is included.
The *default* case should always be present and trigger an error if it should not be reached, yet is reached.
If you need to create variables put all the code in a block.

Example

```
switch (...)
{
  case 1:
    ...
  // FALL THROUGH

  case 2:
  {
    int v;
    ...
  }
  break;

  default:
}
```

Use of *goto,continue,break* and *?:*

Goto

Goto statements should be used sparingly, as in any well-structured code. The goto debates are boring so we won't go into them here. The main place where they can be usefully employed is to break out of several levels of switch, for, and while nesting, although the need to do such a thing may indicate that the inner constructs should be broken out into a separate function, with a success/failure return code.

```
   for (...)
   {
     while (...)
     {
       ...
       if (disaster)
         goto error;
     }
   }
   ...
 error:
   clean up the mess
```

When a goto is necessary the accompanying label should be alone on a line and to the left of the code that follows. The goto should be commented (possibly in the block header) as to its utility and purpose.

Continue and Break

Continue and break are really disguised gotos so they are covered here.

Continue and break like goto should be used sparingly as they are magic in code. With a simple spell the reader is beamed to god knows where for some usually undocumented reason.

The two main problems with continue are:

> It may bypass the test condition
> It may bypass the increment/decrement expression

Consider the following example where both problems occur:

```
 while (TRUE)
 {
   ...
   // A lot of code
   ...
   if (/* some condition */) {
     continue;
   }
   ...
   // A lot of code
   ...
```

```
    if ( i++ > STOP_VALUE) break;
 }
```

Note: "A lot of code" is necessary in order that the problem cannot be caught easily by the programmer.

From the above example, a further rule may be given: Mixing continue with break in the same loop is a sure way to disaster.

?:

The trouble is people usually try and stuff too much code in between the *?* and *:*. Here are a couple of clarity rules to follow:

> Put the condition in parens so as to set it off from other code
> If possible, the actions for the test should be simple functions.
> Put the action for the then and else statement on a separate line
> unless it can be clearly put on one line.

Example

```
    (condition) ? funct1() : func2();

    or

    (condition)
       ? long statement
       : another long statement;
```

Alignment of Declaration Blocks

> Block of declarations should be aligned.

Justification

> Clarity.
> Similarly blocks of initialization of variables should be tabulated.
> The '&' and '*' tokens should be adjacent to the type, not the name.

Example

```
    DWORD        mDword
    DWORD*       mpDword
```

```
char*          mpChar
char           mChar

mDword     = 0;
mpDword    = NULL;
mpChar     = NULL;
mChar      = 0;
```

Macros

Don't Turn C++ into Pascal

Don't change syntax via macro substitution. It makes the program unintelligible to all but the perpetrator.

Replace Macros with Inline Functions

In C++ macros are not needed for code efficiency. Use inlines.

Example

```
#define  MAX(x,y)        (((x) > (y) ? (x) : (y))      // Get the maximum
```

The macro above can be replaced for integers with the following inline function with no loss of efficiency:

```
inline int
max(int x, int y)
{
  return (x > y ? x : y);
}
```

Be Careful of Side Effects

Macros should be used with caution because of the potential for error when invoked with an expression that has side effects.

Example

```
MAX(f(x),z++);
```

Always Wrap the Expression in Parenthesis

When putting expressions in macros always wrap the expression in parenthesis to avoid potential communitive operation abiguity.

Example

```
#define ADD(x,y) x + y
```

must be written as

```
#define ADD(x,y) (x + y)
```

Make Macro Names Unique

Like global variables macros can conflict with macros from other packages.

1. Prepend macro names with package names.
2. Avoid simple and common names like MAX and MIN.

Initialize all Variables

You shall always initialize variables. Always. Every time.

Justification

More problems than you can believe are eventually traced back to a pointer or variable left uninitialized. C++ tends to encourage this by spreading initialization to each constructor. See [Init Idiom for Initializing Objects](#) .

Init Idiom for Initializing Objects

Objects with multiple constructors and/or multiple attributes should define a private Init() method to initialize all attributes. If the number of different member variables is small then this idiom may not be a big win and C++'s constructor initialization syntax can/should be used.

Justification

When using C++'s ability to initialize variables in the constructor it's difficult with multiple constructors and/or multiple attributes to make sure all attributes are initialized. When an attribute is added or changed almost invariably we'll miss changing a constructor.

It's better to define one method, Init(), that initializes all possible attributes. Init() should be called first from every constructor.

The Init() idiom cannot be used in two cases where initialization from a constructor is required:
- o constructing a member object
- o initializing a member attribute that is a reference

Example

```
class Test
{
public:
  Test()
  {
    Init();   // Call to common object initializer
  }

  Test(int val)
  {
    Init();   // Call to common object initializer
    mVal= val;
  }

private:
  int     mVal;
  String*  mpName;

  void Init()
  {
    mVal  = 0;
    mpName= 0;
  }
}
```

Since the number of member variables is small, this might be better written as:

```
class Test
{
public:
  Test(int val = 0, String* name = 0)
    : mVal(val), mpName(name) {}
private:
  int       mVal;
  String*    mpName;
```

```
};
```

One Statement Per Line

There should be only one statement per line unless the statements are very closely related.

The reasons are:

1. The code is easier to read. Use some white space too. Nothing better than to read code that is one line after another with no white space or comments.

One Variable Per Line

Related to this is always define one variable per line:

```
Not:
char **a, *x;

Do:
char** a= 0;  // add doc
char*  x= 0;  // add doc
```

The reasons are:

1. Documentation can be added for the variable on the line.
2. It's clear that the variables are initialized.
3. Declarations are clear which reduces the probablity of declaring a pointer when you meant to declare just a char.

Short Methods

Methods should limit themselves to a single page of code.

Justification

The idea is that the each method represents a technique for achieving a single objective.
Most arguments of inefficiency turn out to be false in the long run.

> True function calls are slower than not, but there needs to a thought out decision (see premature optimization).

Document Null Statements

Always document a null body for a for or while statement so that it is clear that the null body is intentional and not missing code.

```
while (*dest++ = *src++)
   ;        // VOID
```

Do Not Default If Test to Non-Zero

Do not default the test for non-zero, i.e.

```
if (FAIL != f())
```

is better than

```
if (f())
```

even though FAIL may have the value 0 which C considers to be false. An explicit test will help you out later when somebody decides that a failure return should be -1 instead of 0. Explicit comparison should be used even if the comparison value will never change; e.g., if (!(bufsize % sizeof(int))) should be written instead as if ((bufsize % sizeof(int)) == 0) to reflect the numeric (not boolean) nature of the test. A frequent trouble spot is using strcmp to test for string equality, where the result should *never ever* be defaulted. The preferred approach is to define a macro *STREQ*.

```
#define STREQ(a, b) (strcmp((a), (b)) == 0)
```

Or better yet use an inline method:

```
inline bool
StringEqual(char* a, char* b)
```

```
  {
    (strcmp(a, b) == 0) ? return true : return false;
          Or more compactly:
    return strcmp(a, b) == 0;
  }
```

Note, this is just an example, you should really use the standard library string type for doing the comparison.

The non-zero test is often defaulted for predicates and other functions or expressions which meet the following restrictions:

> Returns 0 for false, nothing else.
> Is named so that the meaning of (say) a true return is absolutely obvious. Call a predicate IsValid(), not CheckValid().

The Bull of Boolean Types

Any project using source code from many sources knows the pain of multiple conflicting boolean types. The new C++ standard defines a native boolean type. Until all compilers support bool, and existing code is changed to use it, we must still deal with the cruel world.

The form of boolean most accurately matching the new standard is:

```
  typedef int    bool;
  #define TRUE    1
  #define FALSE   0
```

 or

```
  const int TRUE  = 1;
  const int FALSE = 0;
```

Note, the standard defines the names true and false not TRUE and FALSE. The all caps versions are used to not clash if the standard versions are available.

Even with these declarations, do not check a boolean value for equality with 1 (TRUE, YES, etc.); instead test for inequality with 0 (FALSE, NO, etc.). Most functions are guaranteed to return 0 if false, but only non-zero if true. Thus,

```
  if (TRUE == func()) { ...
```

must be written

```
if (FALSE != func()) { ...
```

Comment: I don't agree with this statement ! If a function returns a bool return type, we should do:
If (true == func()) or if (false != func()) or if (func())
regarding which one is most readable for the programmer!
If the function returns an int, we have to do:
If (0 == func()) as said above, but without using this damn global TRUE.. !
In my opinion, we should not use "TRUE" or "FALSE" anyway ! C++ provides the bool type, therefore let's use it !
(se)

Usually Avoid Embedded Assignments

There is a time and a place for embedded assignment statements. In some constructs there is no better way to accomplish the results without making the code bulkier and less readable.

```
while (EOF != (c = getchar()))
{
  process the character
}
```

The ++ and -- operators count as assignment statements. So, for many purposes, do functions with side effects. Using embedded assignment statements to improve run-time performance is also possible. However, one should consider the tradeoff between increased speed and decreased maintainability that results when embedded assignments are used in artificial places. For example,

```
a = b + c;
d = a + r;
```

should not be replaced by

```
d = (a = b + c) + r;
```

even though the latter may save one cycle. In the long run the time difference between the two will decrease as the optimizer gains maturity, while the difference in ease of maintenance will increase as the human memory of what's going on in the latter piece of code begins to fade.

Reusing Your Hard Work and the Hard Work of Others

Reuse across projects is almost impossible without a common framework in place. Objects conform to the services available to them. Different projects have different service environments making object reuse difficult.

Developing a common framework takes a lot of up front design effort. When this effort is not made, for whatever reasons, there are several techniques one can use to encourage reuse:

Ask! Email a Broadcast Request to the Group

This simple technique is rarely done. For some reason programmers feel it makes them seem less capable if they ask others for help. This is silly! Do new interesting work. Don't reinvent the same stuff over and over again.

If you need a piece of code email to the group asking if someone has already done it. The results can be surprising.

In most large groups individuals have no idea what other people are doing. You may even find someone is looking for something to do and will volunteer to do the code for you. There's always a gold mine out there if people work together.

Tell! When You do Something Tell Everyone

Let other people know if you have done something reusable. Don't be shy. And don't hide your work to protect your pride. Once people get in the habit of sharing work everyone gets better.

Don't be Afraid of Small Libraries

One common enemy of reuse is people not making libraries out of their code. A reusable class may be hiding in a program directory and will never have the thrill of being shared because the programmer won't factor the class or classes into a library.

One reason for this is because people don't like making small libraries. There's something about small libraries that doesn't feel right. Get over it. The computer doesn't care how many libraries you have.

If you have code that can be reused and can't be placed in an existing library then make a new library. Libraries don't stay small for long if people are really thinking about reuse.

If you are afraid of having to update makefiles when libraries are recomposed or added then don't include libraries in your makefiles, include the idea of services. Base level makefiles define services that are each composed of a set of libraries. Higher level makefiles specify the services they want. When the libraries for a service change only the lower level makefiles will have to change.

Keep a Repository

Most companies have no idea what code they have. And most programmers still don't communicate what they have done or ask for what currently exists. The solution is to keep a repository of what's available.

In an ideal world a programmer could go to a web page, browse or search a list of packaged libraries, taking what they need. If you can set up such a system where programmers voluntarily maintain such a system, great. If you have a librarian in charge of detecting reusability, even better.

Another approach is to automatically generate a repository from the source code. This is done by using common class, method, library, and subsystem headers that can double as man pages and repository entries.

Comments on Comments

Comments Should Tell a Story

Consider your comments a story describing the system. Expect your comments to be extracted by a robot and formed into a man page. Class comments are one part of the story, method signature comments are another part of the story, method arguments another part, and method implementation yet another part. All these parts should weave together and inform someone else at another point of time just exactly what you did and why.

Document Decisions

Comments should document decisions. At every point where you had a choice of what to do place a comment describing which choice you made and why. Archeologists will find this the most useful information.

Use Headers

Use a document extraction system like DOXYGEN. Other sections in this document describe how to use ccdoc to document a class and method.

These headers are structured in such a way as they can be parsed and extracted. They are not useless like normal headers. So take time to fill them out. If you do it right once no more documentation may be necessary. See Class Layout for more information.

Comment Layout

Each part of the project has a specific comment layout.

Make Gotchas Explicit

Explicitly comment variables changed out of the normal control flow or other code likely to break during maintenance. Embedded keywords are used to point out issues and potential problems. Consider a robot will parse your comments looking for keywords, stripping them out, and making a report so people can make a special effort where needed.

Gotcha Keywords

> :TODO: topic
> Means there's more to do here, don't forget.
> :BUG: [bugid] topic
> means there's a Known bug here, explain it and optionally give a bug ID.
> :KLUDGE:
> When you've done something ugly say so and explain how you would do it differently next time if you had more time.
> :TRICKY:
> Tells somebody that the following code is very tricky so don't go changing it without thinking.
> :WARNING:
> Beware of something.
> :COMPILER:
> Sometimes you need to work around a compiler problem. Document it. The problem may go away eventually.
> :ATTRIBUTE: value
> The general form of an attribute embedded in a comment. You can make up your own attributes and they'll be extracted.

Gotcha Formatting

Make the gotcha keyword the first symbol in the comment.
Comments may consist of multiple lines, but the first line should be
a self-containing, meaningful summary.
The writer's name and the date of the remark should be part of the
comment. This information is in the source repository, but it can
take a quite a while to find out when and by whom it was added.
Often gotchas stick around longer than they should. Embedding date
information allows other programmer to make this decision.
Embedding who information lets us know who to ask.

Example

```
// :TODO: tmh 960810: possible performance problem
// We should really use a hash table here but for now we'll
// use a linear search.

// :KLUDGE: tmh 960810: possible unsafe type cast
// We need a cast here to recover the derived type. It should
// probably use a virtual method or template.
```

See Also

See Interface and Implementation Documentation for more details on how
documentation should be laid out.

Interface and Implementation Documentation

There are two main audiences for documentation:

Class Users
Class Implementors

With a little forethought we can extract both types of documentation
directly from source code.

Class Users

Class users need class interface information which when structured
correctly can be extracted directly from a header file. When filling out the
header comment blocks for a class, only include information needed by

programmers who use the class. Don't delve into algorithm implementation details unless the details are needed by a user of the class. Consider comments in a header file a man page in waiting.

Class Implementors

Class implementors require in-depth knowledge of how a class is implemented. This comment type is found in the source file(s) implementing a class. Don't worry about interface issues. Header comment blocks in a source file should cover algorithm issues and other design decisions. Comment blocks within a method's implementation should explain even more.

Directory Documentation

Every directory should have a README file that covers:

> the purpose of the directory and what it contains
> a one line comment on each file. A comment can usually be extracted from the NAME attribute of the file header.
> cover build and install directions
> direct people to related resources:
> - directories of source
> - online documentation
> - paper documentation
> - design documentation
> anything else that might help someone

Consider a new person coming in 6 months after every original person on a project has gone. That lone scared explorer should be able to piece together a picture of the whole project by traversing a source directory tree and reading README files, Makefiles, and source file headers.

Follow the Law of Demeter

The *Law of Demeter* states that you shouldn't access a contained object directly from the containing object, you should use a method of the containing object that does what you want and accesses any of its objects as needed.

Justification

The purpose of this law is to break dependencies so implementations can change without breaking code. If an object wishes to remove one of its contained objects it won't be able to do so because some other object is using it. If instead the service was through an interface the object could change its implementation anytime without ill effect.

Caveat

As for most laws the Law of Demeter should be ignored in certain cases. If you have a really high level object that contains a lot of subobjects, like a car contains thousands of parts, it can get absurd to created a method in car for every access to a subobject.

Example

```
class SunWorkstation
{
public:
  void        upVolume(int amount) { m_Sound.Up(amount); }

  SoundCard    m_Sound;

private:
  GraphicsCard  m_Graphics;
}

SunWorksation sun;

Do   : sun.upVolume(1);
Don't: sun.m_Sound.Up(1);
```

Minimize Dependencies with Abstract Base Classes

One of the most important strategies in C++ is to remove dependencies among different subsystems. Abstract base classes (ABCs) are a solid technique for dependency removal.

An ABC is an abstraction of a common form such that it can be used to build more specific forms. An ABC is a common interface that is reusable across a broad range of similar classes. By specifying a common interface as long as a class conforming to that interface is used it doesn't really matter what is the type of the derived type. This breaks code dependencies. New classes, conforming to the interface, can be

substituted in at will without breaking code. In C++ interfaces are specified by using base classes with virtual methods.

The above is a bit rambling because it's a hard idea to convey. So let's use an example: We are doing a GUI where things jump around on the screen. One approach is to do something like:

```
class Frog
{
public:
   void jump();
}
class Bean
{
public:
   void jump();
}
```

The GUI folks could instantiate each object and call the Jump method of each object. The Jump method of each object contains the implementation of jumping behavior for that type of object. Obviously frogs and beans jump differently even though both can jump.

Unfortunately the owner of Bean didn't like the word Jump so they changed the method name to Leap. This broke the code in the GUI and one whole week was lost.

Then someone wanted to see a horse jump so a Horse class was added:

```
class Horse
{
public:
   void jump();
}
```

The GUI people had to change their code again to add Horse.

Then someone updated Horse so that its Jump behavior was slightly different. Unfortunately this caused a total recompile of the GUI code and they were pissed.

Someone got the bright idea of trying to remove all the above dependencies using abstract base classes. They made one base class that specified an interface for jumping things:

```
class Jumpable
{
public:
```

```
    virtual void jump() = 0;
}
```

Jumpable is a base class because other classes need to derive from it so they can get Jumpable's interface. It's an abstract base class because one or more of its methods has the = *0* notation which means the method is a *pure virtual method*. Pure virtual methods must be implemented by derived classes. The compiler checks.

Not all methods in an ABC must be pure virtual, some may have an implementation. This is especially true when creating a base class encapsulating a process common to a lot of objects. For example, devices that must be opened, diagnostics run, booted, executed, and then closed on a certain event may create an ABC called Device that has a method called LifeCycle which calls all other methods in turn thus running through all phases of a device's life. Each device phase would have a pure virtual method in the base class requiring implementation by more specific devices. This way the process of using a device is made common but the specifics of a device are hidden behind a common interface.

Back to Jumpable. All the classes were changed to derive from Jumpable:

```
class Frog : public Jumpable
{
public:
    virtual void jump() { ... }
}

etc ...
```

We see an immediate benefit: we know all classes derived from Jumpable must have a Jump method. No one can go changing the name to Leap without the compiler complaining. One dependency broken.

Another benefit is that we can pass Jumpable objects to the GUI, not specific objects like Horse or Frog:

```
class Gui
{
public:
    void makeJump(Jumpable*);
}

Gui gui;
Frog* pFrog= new Frog;

gui.makeJump(pFrog);
```

Notice Gui doesn't even know it's making a frog jump, it just has a jumpable thing, that's all it cares about. When Gui calls the Jump method it will get the implementation for Frog's Jump method. Another dependency down. Gui doesn't have to know what kind of objects are jumping.

We also removed the recompile dependency. Because Gui doesn't contain any Frog objects it will not be recompiled when Frog changes.

Downside

Wow! Great stuff! Yes but there are a few downsides:

Overhead for Virtual Methods

Virtual methods have a space and time penalty. It's not huge, but should be considered in design.

Make Everything an ABC!

Sometimes people overdo it, making everything an ABC. The rule is make an ABC when you need one not when you might need one. It takes effort to design a good ABC, throwing in a virtual method doesn't an ABC make. Pick and choose your spots. When some process or some interface can be reused and people will actually make use of the reuse then make an ABC and don't look back.

Use a Design Notation and Process

Programmers need to have a common language for talking about coding, designs, and the software process in general. This is critical to project success.

Any project brings together people of widely varying skills, knowledge, and experience. Even if everyone on a project is a genius you will still fail because people will endlessly talk past each other because there is no common language and processes binding the project together. All you'll get is massive fights, burnout, and little progress. If you send your group to training they may not come back seasoned experts but at least your group will all be on the same page; a team.

There are many popular methodologies out there. The point is to do some research, pick a method, train your people on it, and use it. Take a look at the top of this page for links to various methodologies.

You may find the CRC (class responsibility cards) approach to teasing out a design useful. Many others have. It is an informal approach encouraging team cooperation and focusing on objects doing things rather than objects having attributes. There's even a whole book on it: Using CRC Cards by Nancy M. Wilkinson.

Using Use Cases

A *use case* is a generic description of an entire transaction involving several objects. A use case can also describe the behaviour of a set of objects, such as an organization. A use case model thus presents a collection of use cases and is typically used to specify the behavior of a whole application system together with one or more external actors that interact with the system.

An individual use case may have a name (although it is typically not a simple name). Its meaning is often written as an informal text description of the external actors and the sequences of events between objects that make up the transaction. Use cases can include other use cases as part of their behaviour.

Requirements Capture

Use cases attempt to capture the requirements for a system in an understandable form. The idea is by running through a set of use case we can verify that the system is doing what it should be doing.

Have as many use cases as needed to describe what a system needs to accomplish.

The Process

> Start by understanding the system you are trying to build.
> Create a set of use cases describing how the system is to be used by all its different audiences.
> Create a class and object model for the system.
> Run through all the use cases to make sure your model can handle all the cases. Update your model and create new use cases as necessary.

Unified Modeling Language

11.05.2004

The Unified Modeling Language is too large to present here. Fortunately you can see it at [Rational's](#) web site. Since you do need a modeling language UML is a safe choice. It combines features from several methods into one unified language. Remember all languages and methods are open to local customization. If their language is too complex then use the parts you and your project feel they need and junk the rest.

OPEN Method

[OPEN](#) stands for Object-oriented Process, Environment and Notation and is a worthy if not superior competitor to UML. It is another group effort composed of basically all the people not in the UML group :-) Their web site has a good comparison of OPEN and UML.

My guess is UML will win out for marketing reasons. But it is good to have some competition going.

Liskov's Substitution Principle (LSP)

This principle states:

> All classes derived from a base class should be interchangeable
> when used as a base class.

The idea is users of a class should be able to count on similar behavior from all classes that derive from a base class. No special code should be necessary to qualify an object before using it. If you think about it violating LSP is also violating the [Open/Closed](#) principle because the code would have to be modified every time a derived class was added. It's also related to dependency management using [abstract base classes](#).

For example, if the [Jump method](#) of a Frog object implementing the Jumpable interface actually makes a call and orders pizza we can say its implementation is not in the spirit of Jump and probably all other objects implementing Jump. Before calling a Jump method a programmer would now have to check for the Frog type so it wouldn't screw up the system. We don't want this in programs. We want to use base classes and feel comfortable we will get consistent behaviour.

LSP is a very restrictive idea. It constrains implementors quite a bit. In general people support LSP and have LSP as a goal.

Open/Closed Principle

The Open/Closed principle states a class must be open and closed where:

> open means a class has the ability to be extended.
> closed means a class is closed for modifications other than extension. The idea is once a class has been approved for use having gone through code reviews, unit tests, and other qualifying procedures, you don't want to change the class very much, just extend it.

The Open/Closed principle is a pitch for stability. A system is extended by adding new code not by changing already working code. Programmers often don't feel comfortable changing old code because it works! This principle just gives you an academic sounding justification for your fears :-)

In practice the Open/Closed principle simply means making good use of our old friends abstraction and polymorphism. Abstraction to factor out common processes and ideas. Inheritance to create an interface that must be adhered to by derived classes. In C++ we are talking about using [abstract base classes](). A lot.

Design by Contract

The idea of design by contract is strongly related to [LSP](). A contract is a formal statement of what to expect from another party. In this case the contract is between pieces of code. An object and/or method states that it does X and you are supposed to believe it. For example, when you ask an object for its volume that's what you should get. And because volume is a verifiable attribute of a thing you could run a series of checks to verify volume is correct, that is, it satisfies its contract.

The contract is enforced in languages like Eiffel by pre and post condition statements that are actually part of the language. In other languages a bit of faith is needed.

Design by contract when coupled with language based verification mechanisms is a very powerful idea. It makes programming more like assembling spec'd parts.

11.05.2004

Don't Over Use Operators

C++ allows the overloading of all kinds of weird operators. Unless you are building a class directly related to math there are very few operators you should override. Only override an operator when the semantics will be clear to users.

Justification

Very few people will have the same intuition as you about what a particular operator will do.

Naming Class Files

Class Definition in One File

Each class definition should be in its own file where each file is named directly after the class's name:

    ClassName.h

Implementation in One File

In general each class should be implemented in one source file:

    ClassName.cc   // or whatever the extension is: cpp, c++

But When it Gets Really Big...

If the source file gets too large or you want to avoid compiling templates all the time then add additional files named according to the following rule:

    ClassName_section.C

section is some name that identifies why the code is chunked together. The class name and section name are separated by '_'.

Miscellaneous

This section contains some miscellaneous do's and don'ts.

Don't use floating-point variables where discrete values are needed. Using a float for a loop counter is a great way to shoot yourself in the foot. Always test floating-point numbers as <= or >=, never use an exact comparison (== or !=).

Compilers have bugs. Common trouble spots include structure assignment and bit fields. You cannot generally predict which bugs a compiler has. You could write a program that avoids all constructs that are known broken on all compilers. You won't be able to write anything useful, you might still encounter bugs, and the compiler might get fixed in the meanwhile. Thus, you should write ``around'' compiler bugs only when you are forced to use a particular buggy compiler.

Do not rely on automatic beautifiers. The main person who benefits from good program style is the programmer him/herself, and especially in the early design of handwritten algorithms or pseudo-code. Automatic beautifiers can only be applied to complete, syntactically correct programs and hence are not available when the need for attention to white space and indentation is greatest. Programmers can do a better job of making clear the complete visual layout of a function or file, with the normal attention to detail of a careful programmer (in other words, some of the visual layout is dictated by intent rather than syntax and beautifiers cannot read minds). Sloppy programmers should learn to be careful programmers instead of relying on a beautifier to make their code readable. Finally, since beautifiers are non-trivial programs that must parse the source, a sophisticated beautifier is not worth the benefits gained by such a program. Beautifiers are best for gross formatting of machine-generated code.

Accidental omission of the second ``='' of the logical compare is a problem. The following is confusing and prone to error.

```
if (abool= bbool) { ... }
```

Does the programmer really mean assignment here? Often yes, but usually no. The solution is to just not do it, an inverse Nike philosophy. Instead use explicit tests and avoid assignment with an implicit test. The recommended form is to do the assignment before doing the test:

```
abool= bbool;
if (abool) { ... }
```

Modern compilers will put variables in registers automatically. Use the register sparingly to indicate the variables that you think are most critical. In extreme cases, mark the 2-4 most critical values as register and mark the rest as REGISTER. The latter can be #defined to register on those machines with many registers.

Be Const Correct

C++ provides the *const* key word to allow passing as parameters objects that cannot change to indicate when a method doesn't modify its object. Using const in all the right places is called "const correctness." It's hard at first, but using const really tightens up your coding style. Const correctness grows on you.

For more information see Const Correctness in the C++ FAQ.

Use Streams

Programmers transitioning from C to C++ find stream IO strange preferring the familiarity of good old stdio. Printf and gang seem to be more convenient and are well understood.

Type Safety

Stdio is not type safe, which is one of the reasons you are using C++, right? Stream IO is type safe. That's one good reason to use streams.

Standard Interface

When you want to dump an object to a stream there is a standard way of doing it: with the << operator. This is not true of objects and stdio.

Interchangeablity of Streams

One of the more advanced reasons for using streams is that once an object can dump itself to a stream it can dump itself to any stream. One stream may go to the screen, but another stream may be a serial port or network connection. Good stuff.

Streams Got Better

11.05.2004

Stream IO is not perfect. It is however a lot better than it used to be. Streams are now standardized, acceptably efficient, more reliable, and now there's lots of documentation on how to use streams.

Check Thread Safety

Some stream implementations are not yet thread safe. Make sure that yours is.

But Not Perfect

For an embedded target tight on memory streams do not make sense. Streams inline a lot of code so you might find the image larger than you wish. Experiment a little. Streams might work on your target.

Use #if Not #ifdef

Use #if MACRO not #ifdef MACRO. Someone might write code like:

```
#ifdef DEBUG
     temporary_debugger_break();
#endif
```

Someone else might compile the code with turned-of debug info like:

```
cc -c lurker.cpp -DDEBUG=0
```

Alway use #if, if you have to use the preprocessor. This works fine, and does the right thing, even if DEBUG is not defined at all (!)

```
#if DEBUG
     temporary_debugger_break();
#endif
```

If you really need to test whether a symbol is defined or not, test it with the defined() construct, which allows you to add more things later to the conditional without editing text that's already in the program:

```
#if !defined(USER_NAME)
 #define USER_NAME "john smith"
#endif
```

Commenting Out Large Code Blocks

Sometimes large blocks of code need to be commented out for testing.

Using #if 0

The easiest way to do this is with an #if 0 block:

```
void
example()
{
  great looking code

  #if 0
  lots of code
  #endif

  more code
}
```

You can't use /**/ style comments because comments can't contain comments and surely a large block of your code will contain a comment, won't it?

Don't use #ifdef as someone can unknowingly trigger ifdefs from the compiler command line.

Use Descriptive Macro Names Instead of 0

The problem with #if 0is that even day later you or anyone else has know idea why this code is commented out. Is it because a feature has been dropped? Is it because it was buggy? It didn't compile? Can it be added back? It's a mystery.

Use Descriptive Macro Names Instead of #if 0

```
#if NOT_YET_IMPLEMENTED

#if OBSOLETE

#if TEMP_DISABLED
```

Add a Comment to Document Why

Add a short comment explaining why it is not implemented, obsolete or temporarily disabled.

Register/Dispatch Idiom

Another strategy for reducing dependencies in a system is the Register/Dispatch Idiom (RDI). RDI treats large grained occurrences in a system as events. Events are identified by some unique identifier. Objects in the system register with a dispatch system for events or classes of events it is interested in. Objects that are event sources send events into the dispatch system so the dispatch system can route events to consumers.

RDI separates producers and consumers on a distributed scale. Event producers and consumers don't have to know about each other at all. Consumers can drop out of the event stream by deregistering for events. New consumers can register for events at anytime. Event producers can drop out with no ill effect to event consumers, the consumer just won't get any more events. It is a good idea for producers to have an "I'm going down event" so consumers can react intelligently.

Logically the dispatch system is a central entity. The implementation however can be quite different. For a highly distributed system a truly centralized event dispatcher would be a performance bottleneck and a single point of failure. Think of event dispatchers as being a lot of different processes cast about on various machines for redundancy purposes. Event processors communicate amongst each other to distribute knowledge about event consumers and producers. Much like a routing protocol distributes routing information to its peers.

RDI works equally well in the small, in processes and single workstations. Parts of the system can register as event consumers and event producers making for a very flexible system. Complex decisions in a system are expressed as event registrations and deregistrations. No further level of cooperation required.

More expressive event filters can also be used. The above proposal filters events on some unique ID. Often you want events filtered on more complex criteria, much like a database query. For this to work the system has to understand all data formats. This is easy if you use a common format like attribute value pairs. Otherwise each filter needs code understanding packet formats. Compiling in filter code to each dispatcher is one approach. Creating a downloadable generic stack based filter

language has been used with success on other projects, being both simple and efficient.

Different Accessor Styles

Why Accessors?

Access methods provide access to the physical or logical attributes of an object. Accessing an object's attributes directly as we do for C structures is greatly discouraged in C++. We disallow direct access to attributes to break dependencies, the reason we do most things. Directly accessing an attribute exposes implementation details about the object.

To see why ask yourself:

> What if the object decided to provide the attribute in a way other than physical containment?
> What if it had to do a database lookup for the attribute?
> What if a different object now contained the attribute?

If any of the above changed code would break. An object makes a contract with the user to provide access to a particular attribute; it should not promise how it gets those attributes. Accessing a physical attribute makes such a promise.

Accessors Considered Somewhat Harmful

At least in the public interface having accessors many times is an admission of failure, a failure to make an object's interface complete. At the protected or private level accessors are fine as these are the implementation levels of a class.

Implementing Accessors

There are three major idioms for creating accessors.

Get/Set

```
class X
{
public:
  int   getAge() const    { return m_Age; }
  void  setAge(int age)   { m_Age= age; }
private:
  int m_Age;
```

```
    }
```

The problem with Get/Set is twofold:

> It's ugly. Get and Set are strewn throughout the code cluttering it up.
> It doesn't treat attributes as objects in their own right. An object will have an assignment operator. Why shouldn't age be an object and have its own assignment operator?

One benefit, that it shares with the *One Method Name*, is when used with messages the set method can transparently transform from native machine representations to network byte order.

One Method Name

```
    class X
    {
    public:
      int   age() const    { return m_Age; }
      void   age(int age)    { m_Age= age; }
    private:
      int m_Age;
    }
```

Similar to Get/Set but cleaner. Use this approach when not using the *Attributes as Objects* approach.

Attributes as Objects

```
    class X
    {
    public:
      int           Age() const    { return m_Age; }
      int&          rAge()          { return m_Age; }

      const String&        Name() const   { return m_Name; }
      String&              rName()        { return m_Name; }
    private:
      int          m_Age;
      String       m_Name;
    }
```

The above two attribute examples shows the strength and weakness of the Attributes as Objects approach.

When using an int type, which is not a real object, the int is set directly because *rAge()* returns a reference. The object can do no checking of the

value or do any representation reformatting. For many simple attributes, however, these are not horrible restrictions. A way around this problem is to use a class wrapper around base types like int.

When an object is returned as reference its = operator is invoked to complete the assignment. For example:

```
X x;
x.rName()= "test";
```

This approach is also more consistent with the object philosophy: the object should do it. An object's = operator can do all the checks for the assignment and it's done once in one place, in the object, where it belongs. It's also clean from a name perspective.

When possible use this approach to attribute access.

Layering

Layering is the primary technique for reducing complexity in a system. A system should be divided into layers. Layers should communicate between adjacent layers using well defined interfaces. When a layer uses a non-adjacent layer then a layering violation has occurred.

A layering violation simply means we have dependency between layers that is not controlled by a well defined interface. When one of the layers changes code could break. We don't want code to break so we want layers to work only with other adjacent layers.

Sometimes we need to jump layers for performance reasons. This is fine, but we should know we are doing it and document appropriately.

Delegation

Delegation is the idea of a method using another object's method to do the real work. In some sense the top layer method is *a front* for the other method. Delegation is a form of dependency breaking. The top layer method never has to change while it's implementation can change at will.

Delegation is an alternative to using inheritance for implementation purposes. One can use inheritance to define an interface and delegation to implement the interface.

Some people feel delegation is a more robust form of OO than using implementation inheritance. Delegation encourages the formation of abstract class interfaces and HASA relationships. Both of which encourage reuse and dependency breaking.

Example

```
class TestTaker
{
public:
   void writeDownAnswer()   { m_paidTestTaker.writeDownAnswer(); }
private:
   PaidTestTaker  m_paidTestTaker;
}
```

In this example a test taker delegates actually answering the question to a paid test taker. Not ethical but a definite example of delegation!

Code Reviews

If you can make a formal code review work then my hat is off to you. Code reviews can be very useful. Unfortunately they often degrade into nit picking sessions and endless arguments about silly things. They also tend to take a lot of people's time for a questionable payback.

My god he's questioning code reviews, he's not an engineer!

Not really, it's the form of code reviews and how they fit into normally late chaotic projects is what is being questioned.

First, code reviews are way too late to do much of anything useful. What needs reviewing are requirements and design. This is where you will get more bang for the buck.

Get all relevant people in a room. Lock them in. Go over the class design and requirements until the former is good and the latter is being met. Having all the relevant people in the room makes this process a deep fruitful one as questions can be immediately answered and issues immediately explored. Usually only a couple of such meetings are necessary.

If the above process is done well coding will take care of itself. If you find problems in the code review the best you can usually do is a rewrite after someone has sunk a ton of time and effort into making the code "work."

You will still want to do a code review, just do it offline. Have a couple people you trust read the code in question and simply make comments to the programmer. Then the programmer and reviewers can discuss issues and work them out. Email and quick pointed discussions work well. This approach meets the goals and doesn't take the time of 6 people to do it.

Create a Source Code Control System Early and Not Often

A common build system and source code control system should be put in place as early as possible in a project's lifecycle, preferably before anyone starts coding. Source code control is the structural glue binding a project together. If programmers can't easily use each other's products then you'll never be able to make a good reproducible build and people will piss away a lot of time. It's also hell converting rogue build environments to a standard system. But it seems the right of passage for every project to build their own custom environment that never quite works right.

Some issues to keep in mind:

Appoint a *buildmaster* who is in charge of the build environment and making builds. If you have a tools group so much the better as build environments generally generate a lot of tools.
Programmers should generally leave tagging and other advanced operations to the buildmaster.
Shared source environments like CVS usually work best in largish projects.
Makefiles should be completely templated so developers need no makefile expertise.
You will need multiple builds of the same code meaning programmers will want to have the option of building against optimized code versions, debugger code versions, code with the -DDEBUG macro defined and versions without, etc. One way to do this is have everyone check out all code and do a total rebuild. The other better way is for the buildmaster to build all necessary versions of code so programmers can check out only the code which they are modifying. The idea is that each combination of compile flags and macros defines a build.
Dependency checking should work!
Maintain a cycle of builds. Create a nightly build so problems can be detected early. Create a valid build every week or for every feature set that is guaranteed to be good. Every release to QA or to customers should be tagged and available in the tree. Official builds should be official because they contain a certain set of features that assumably meet a schedule, not because it's a certain date.

If you use CVS use a *reference tree* approach. With this approach a master build tree is kept of various builds. Programmers checkout source against the build they are working on. They only checkout what they need because the make system uses the build for anything not found locally. Using the *-I and -L* flags makes this system easy to setup. Search locally for any files and libraries then search in the reference build. This approach saves on disk space and build time.

Get a lot of disk space. With disk space as cheap it is there is no reason not to keep plenty of builds around and have enough space for programmers to compile.

Make simple things simple. It should be dead simple and well documented on how to:

- check out modules to build
- how to change files
- how to setup makefiles
- how to add new modules into the system
- how to delete modules and files
- how to check in changes
- what are the available libraries and include files
- how to get the build environment including all compilers and other tools

Make a web page or document or whatever. New programmers shouldn't have to go around begging for build secrets from the old timers.

Use the idea of services. If using a feature requires linking to 3 libraries think about making a *service* specifiable in the makefile programmers can use instead of having to know the exact libraries needed. With this approach the libraries making up a service can be changed in a based makefile without requiring changes in all makefiles.

On checkins log comments should be useful. These comments should be collected every night and sent to interested parties.

Sources

If you have the money many projects have found [Clear Case](link) a good system. Perfectly workable systems have been build on top of GNU make and CVS. CVS is a freeware build environment built on top of RCS. Its main difference from RCS is that is supports a shared file model to building software.

11.05.2004

Create a Bug Tracking System Early and Not Often

The earlier people get used to using a bug tracking system the better. If you are 3/4 through a project and then install a bug tracking system it won't be used. You need to install a bug tracking system early so people will use it.

Programmers generally resist bug tracking, yet when used correctly it can really help a project:

> Problems aren't dropped on the floor.
> Problems are automatically routed to responsible individuals.
> The lifecycle of a problem is tracked so people can argue back and forth with good information.
> Managers can make the big schedule and staffing decisions based on the number of and types of bugs in the system.
> Configuration management has a hope of matching patches back to the problems they fix.
> QA and technical support have a communication medium with developers.

Not sexy things, just good solid project improvements.

FYI, it's not a good idea to reward people by the number of bugs they fix :-)

Source code control should be linked to the bug tracking system. During the part of a project where source is frozen before a release only checkins accompanied by a valid bug ID should be accepted. And when code is changed to fix a bug the bug ID should be included in the checkin comments.

Sources

Several projects have found DDTS a workable system. There is also a GNU bug tracking system available. Roll your own is a popular option but using an existing system seems more cost efficient.

Honor Responsibilities

Responsibility for software modules is scoped. Modules are either the responsibility of a particular person or are common. Honor this division of

responsibility. Don't go changing things that aren't your responsibility to change. Only mistakes and hard feelings will result.

Face it, if you don't own a piece of code you can't possibly be in a position to change it. There's too much context. Assumptions seemingly reasonable to you may be totally wrong. If you need a change simply ask the responsible person to change it. Or ask them if it is OK to make such-n-such a change. If they say OK then go ahead, otherwise holster your editor.

Every rule has exceptions. If it's 3 in the morning and you need to make a change to make a deliverable then you have to do it. If someone is on vacation and no one has been assigned their module then you have to do it. If you make changes in other people's code try and use the same style they have adopted.

Programmers need to mark with comments code that is particularly sensitive to change. If code in one area requires changes to code in an another area then say so. If changing data formats will cause conflicts with persistent stores or remote message sending then say so. If you are trying to minimize memory usage or achieve some other end then say so. Not everyone is as brilliant as you.

The worst sin is to flit through the system changing bits of code to match your coding style. If someone isn't coding to the standards then ask them or ask your manager to ask them to code to the standards. Use common courtesy.

Code with common responsibility should be treated with care. Resist making radical changes as the conflicts will be hard to resolve. Put comments in the file on how the file should be extended so everyone will follow the same rules. Try and use a common structure in all common files so people don't have to guess on where to find things and how to make changes. Checkin changes as soon as possible so conflicts don't build up.

As an aside, module responsibilities must also be assigned for bug tracking purposes.

Process Automation

It's a sad fact of human nature that if you don't measure it or check for it: it won't happen. The implication is you must automate as much of the development process as possible and provide direct feedback to developers on specific issues that they can fix.

Process automation also frees up developers to do real work because they don't have to babysit builds and other project time sinks.

Automated Builds and Error Assignment

Create an automated build system that can create nightly builds, parse the build errors, assign the errors to developers, and email developers their particular errors so they can fix them.

This is the best way to maintain a clean build. Make sure the list of all errors for a build is available for everyone to see so everyone can see everyone elses errors. The goal is replace a blaim culture with a culture that tries to get things right and fixes them when they are wrong. Immediate feedback makes this possible.

Automated Code Checking

As part of the automated build process you can check for coding standard violations and for other problems. If you don't check for it people will naturally do their own thing. Code reviews aren't good enough to keep the code correct. With a tool like [Abraxis Code Check](#) you can check the code for a lot of potential problems.

This feature like the automated error assignment makes problems immediately visible and immediately correctable, all without a lot of blame and shame.

Documentation Extraction

Related to this principle is the need to automatically extract documentation from the source code and make it available on line for everyone to use. If you don't do this documentation will be seen as generally useless and developers won't put as much effort into it. Making the documentation visible encourages people to do a better job.

Connect Source Code Control System and Bug Tracking System

1. When a check-in of source code fixes a bug then have the check-in automatically tell the bug tracking system that the bug was fixed.
2. When a bug fix is built in a build change the state of the bug to BUILT.
3. Have a submit tool where people ask judges of they can submit a bug fix on the branch.

11.05.2004

There are lots of things you can do depending on how complicated your environment. The more complicated the environment the more you should think about connecting all systems together.

Tools Agreement

The reality of different tool preferences is something to deal with explicitly and openly. Tools include IDEs, languages, editors, make program, source code control, bug system, debuggers, test framework, etc. Some tool decisions by their nature must be project wide, other decisions can be customized per developer.

A split might also be done by who is performing the build. For example, an IDE should be able to used in local builds, but the make program would be used for nightly and release builds.

Certain things are easy/trivial/useful with one tool, but hard/complicated/stupid with another tool. Unstated tool assumptions can be the source of a lot of confusion. "Get a better editor" is not always a workable response, though sometimes that's all there is to it!

Non-Blocking Scheduling

Schedules are lies. Schedules suck. Yes, yes, yes. But we still need them.

The most effective scheduling rule i've used is to schedule so as to unblock others. The idea is to complete the portions of a feature that will unblock those dependent on you. This way development moves along smoothly because more lines of development can be active at a time. For example, instead of implementing the entire database, implement the simple interface and stub it out. People can work for a very long time this way using that portion of the feature that caused others not to block. Plus it's a form of rapid prototyping because you get immediate feedback on these parts. Don't worry about the quality of implementation because it doesn't matter yet.

Using Personas

Personas are a powerful design tool, especially when combined with responsibility driven design.

11.05.2004

http://www.boxesandarrows.com/archives/002330.php. Cooper's personas are:

> simply pretend users of the system you're building. You describe them, in a surprising amount of detail, and then design your system for them.

I have a standard set of personas that i consider when creating a design/architecture that don't seem to be common. When you write code their are a lot of personas looking over your shoulder:

1. other programmers using the code
2. maintenance
3. extension
4. documentation group
5. training group
6. code review
7. test and validation
8. manufacturing
9. field support
10. first and second line technical support
11. live debugging
12. post crash debugging
13. build system (documentation generation and automatic testing)
14. unit testing
15. system testing
16. source code control
17. code readers
18. legal

You are much more careful and more thorough when you really thing about all the personas, all the different people and all their different roles and purposes.

C++ File Extensions

In short: Use the *.h* extension for header files and *.cc* for source files.

For some reason an odd split occurred in early C++ compilers around what C++ source files should be called. C header files always use the *.h* and C source files always use the *.c* extension. What should we use for C++?

The short answer is as long as everyone on your project agrees it doesn't really matter. The build environment should be able to invoke the right compiler for any extension. Historically speaking here have been the options:

      Header Files: .h, .hh, .hpp
      Source Files: .C, .cpp, .cc

Header File Extension Discussion

Using *.hh* extension is not widely popular but makes a certain kind of sense. C header files use *.h* file extension and C++ based header files use *.hh* file extension. The problem is if we consider a header file an [interface](#) to a service then we can have a C interface to a service and C++ interface to the service in the same file. Using preprocessor directives this is possible and common. The recommendation is to stick with using the *.h* extension.

Source File Extension Discussion

The problem with the *.C* extension is that it is indistinguishable from the *.c* extensions in operating systems that aren't case sensitive. Yes, this is a UNIX vs. windows issue. Since it is a simple step aiding portability we won't use the *.C* extension. The *.cpp* extension is a little wordy. So the *.cc* extension wins by default.

No Data Definitions in Header Files

Do not put data definitions in header files. for example:

```
/*
 * aheader.h
 */
int x = 0;
```

1. It's bad magic to have space consuming code silently inserted through the innocent use of header files.
2. It's not common practice to define variables in the header file so it will not occur to devellopers to look for this when there are problems.
3. Consider defining the variable once in a .cpp file and use an extern statement to reference it.
4. Consider using a singleton for access to the data.

Mixing C and C++

In order to be backward compatible with dumb linkers C++'s link time type safety is implemented by encoding type information in link symbols, a process called *name mangling*. This creates a problem when linking to C code as C function names are not mangled. When calling a C function from C++ the function name will be mangled unless you turn it off. Name mangling is turned off with the *extern "C"* syntax. If you want to create a C function in C++ you must wrap it with the above syntax. If you want to call a C function in a C library from C++ you must wrap in the above syntax. Here are some examples:

Calling C Functions from C++

```
extern "C" int strncpy(...);
extern "C" int my_great_function();
extern "C"
{
  int strncpy(...);
  int my_great_function();
};
```

Creating a C Function in C++

```
extern "C" void
a_c_function_in_cplusplus(int a)
{
}
```

__*cplusplus* Preprocessor Directive

If you have code that must compile in a C and C++ environment then you must use the __*cplusplus* preprocessor directive. For example:

```
#ifdef __cplusplus

extern "C" some_function();

#else

extern some_function();

#endif
```

No Magic Numbers

A magic number is a bare naked number used in source code. It's magic because no-one has a clue what it means including the author inside 3 months. For example:

```
if     (22 == foo) { start_thermo_nuclear_war(); }
else if (19 == foo) { refund_lotso_money(); }
else if (16 == foo) { infinite_loop(); }
else              { cry_cause_im_lost(); }
```

In the above example what do 22 and 19 mean? If there was a number change or the numbers were just plain wrong how would you know?

Instead of magic numbers use a real name that means something. You can use *#define* or constants or enums as names. Which one is a design choice. For example:

```
#define   PRESIDENT_WENT_CRAZY (22)
const int WE_GOOFED= 19;
enum
{
  THEY_DIDNT_PAY= 16
};

if     (PRESIDENT_WENT_CRAZY == foo) { start_thermo_nuclear_war(); }
else if (WE_GOOFED          == foo) { refund_lotso_money(); }
else if (THEY_DIDNT_PAY     == foo) { infinite_loop(); }
else                               { happy_days_i_know_why_im_here(); }
```

Now isn't that better? The const and enum options are preferable because when debugging the debugger has enough information to display both the value and the label. The #define option just shows up as a number in the debugger which is very inconvenient. The const option has the downside of allocating memory. Only you know if this matters for your application.

Error Return Check Policy

Check every system call for an error return, unless you know you wish to ignore errors. For example, *printf* returns an error code but rarely would you check for its return code. In which case you can cast the return to (void) if you really care.

Include the system error text for every system error message.
Check every call to malloc or realloc unless you know your versions
of these calls do the right thing. You might want to have your own
wrapper for these calls, including new, so you can do the right thing
always and developers don't have to make memory checks
everywhere.

Promise of OO

OO has been hyped to the extent you'd figure it would solve world hunger
and usher in a new era of world peace. Not! OO is an approach, a
philosophy, it's not a recipe which blindly followed yields quality.

Robert Martin put OO in perspective:

OO, when properly employed, does enhance the reusability of
software. But it does so at the cost of complexity and design time.
Reusable code is more complex and takes longer to design and
implement. Furthermore, it often takes two or more tries to create
something that is even marginally reusable.
OO, when properly employed, does enhance the software's
resilience to change. But it does so at the cost of complexity and
design time. This trade off is almost always a win, but it is hard to
swallow sometimes.
OO does not necessarily make anything easier to understand. There
is no magical mapping between the software concepts and every
human's map of the real world. Every person is different. What one
person percieves to be a simple and elegant design, another will
perceive as convoluted and opaque.
If a team has been able, by applying point 1 above, to create a
repository of reusable items, then development times can begin to
shrink significantly due to reuse.
If a team has been able, by applying point 2 above, to create
software that is resilient to change, then maintenance of that
software will be much simpler and much less error prone.

You can't use OO and C++ on Embedded Systems

Oh yes you can. I've used C++ on several embedded systems as have
many others. And if you can't why not? Please don't give in to vague
feelings and prejudice. An attitude best shown with a short exchange:

11.05.2004

```
Rube: Our packet driver is slow. We're only getting 100 packets per second.
Me  : Good thing you didn't do it in C++ huh?
Rube: Oh yah, it would have been really slow then!
Me  : (smiled secretly to myself)
```

My initial response was prompted by a general unacceptance of C++ in the project and blaming C++ for all problems. Of course all the parts written in C and assembly had no problems :-) Embedded systems shops tend to be hardware driven companies and tend not to know much about software development, thus any new fangled concepts like OO and C++ are ridiculed without verbally accessible reasons. Counter arguments like code that is fast and small and reusable don't make a dent. Examples like improving the speed of a driver by inlining certain methods and not hacking the code to death gently roll into the bit bucket.

Techniques

Of course C++ can be a disaster for an embedded system when used incorrectly, which of course is true of any tool. Here's some ideas to use C++ safely in an embedded system:

Get Some Training!

If people don't know C++ and OO then they will likely fail and blame their tools. A good craftsperson doesn't blame their tools. Get training. Hire at least one experienced person as guide/mentor.

Be Careful Using Streams

The streams library is large and slow. You are better off making a "fake" streams library by overloading the << operator. If you have a lot of memory then use streams, they are convenient and useful.

Be Careful Using Templates

Code using templates can suffer from extreme code bloat. This is pretty much a function of your compiler as templates can be efficiently used when done correctly. Test your compiler for it how handles templates. If it doesn't make a copy per file for each template then you are in business. Templates have good time efficiency so they would be nice to use.

You can fix the template code bloat problem by using explicit instantiation. Actually, even if the compiler generates one copy per source file. This, however, is often too much programmer work to expect on a large project, so be careful. Many linkers are smart enough to strip away all but one of the copies.

Another issue to consider is template complexity. Templates can be complex for those new to C++. Bugs in templates are very hard to find and may overwhelm the patience of users.

Exceptions Beware

Embedded applications are usually interrupt driven and multi-threaded. Test that exceptions are thread safe. Many compilers support exceptions, but not thread safe exceptions. And you probably don't want to call code in an interrupt that throws exceptions.

Use Polymorphic Interfaces to Make Frameworks

When you think through your design and come up with good abstractions you will be shocked at how little code and how little time it takes to implement new features.

Make an OS Encapsulation Library

Don't use your embedded OSs features directly. Create a layer that encapsulates OS functions and use those encapsulations. Most feature like tasks, interrupts, semaphores, message queues, messages, etc. are common to all systems. With good encapsulations it's quite possible to have the same code compile for Solaris, VxWorks, Windows, and other systems. It just takes a little thought.

ROM Beware

A lot of systems create a ROM and download code later over the network that is linked against the ROM. Something to remember is linkers will try and include only code that is used. So your ROM may not contain code that loaded code expects to be there. You need to include all functions in your ROM.

Multiple Interface Levels

Most embedded systems have a command line interface which usually requires C linkage, then they may have an SNMP interface, and they may have some sort of other friendly interface. Design this up front to be common across all code. It will make your life much easier. C functions require access to global pointers so they can use objects. The singleton pattern makes this easier.

Come up with common naming conventions. A decent one is:

- o Make up a module abbreviation that can be prefixed to all calls. For example: log for the logging module.

- o Encode an action after the prefix. For example: logHelp which prints help for the logging module.
- o Require a certain set of functions for each sub system: For example:
  - ▪ moduleHelp - prints help for the module
  - ▪ modulePrint - prints the current state of the module
  - ▪ moduleStart - start a module
  - ▪ moduleStop - stop a module
  - ▪ moduleSetDebug - set the debug level for a module. It's very nice to set debug levels on a module by module basis.

Debug and Error System First

Make your debug and error system first so everyone writing code will use it. It's very hard to retrofit code with debug output and intelligent use of error codes. If you have some way to write system assert errors to NVRAM, disk, or some other form of persistent storage so you can recover it on the next reboot.

Think About Memory

Think how you'll share memory buffers between ISR code and task level code. Think how fast your default memory allocator is, it is probably slow. Think if your processor supports purify! Think how you'll track memory corruption and leakage.

Think About System Integrity

You need to design up front how you are going to handle watchdog functions and test that the system is still running and not corrupted.

Remember to Use Volatile

When using memory mapped I/O make sure that you declare the input port variables as volatile, (some compilers do this automatically), since the value can change without notice, and the optimizer could eliminate what looks like a redundant access to that variable. Not using volatile leads to some very obscure bugs. If you suspect problems in this area take a look at the generated code to make sure read-only assumptions are being made.

Sometimes the keyword volatile is ifdef'd out for portability reasons. Check that what you think is volatile is really declared as volatile.

Thin vs. Fat Class Interfaces

11.05.2004

How many methods should an object have? The right answer of course is just the right amount, we'll call this the Goldilocks level. But what is the Goldilocks level? It doesn't exist. You need to make the right judgment for your situation, which is really what programmers are for :-)

The two extremes are thin classes versus thick classes. Thin classes are minimalist classes. Thin classes have as few methods as possible. The expectation is users will derive their own class from the thin class adding any needed methods.

While thin classes may seem "clean" they really aren't. You can't do much with a thin class. Its main purpose is setting up a type. Since thin classes have so little functionality many programmers in a project will create derived classes with everyone adding basically the same methods. This leads to code duplication and maintenance problems which is part of the reason we use objects in the first place. The obvious solution is to push methods up to the base class. Push enough methods up to the base class and you get thick classes.

Thick classes have a lot of methods. If you can think of it a thick class will have it. Why is this a problem? It may not be. If the methods are directly related to the class then there's no real problem with the class containing them. The problem is people get lazy and start adding methods to a class that are related to the class in some willow wispy way, but would be better factored out into another class. Judgment comes into play again.

Thick classes have other problems. As classes get larger they may become harder to understand. They also become harder to debug as interactions become less predictable. And when a method is changed that you don't use or care about your code will still have to be recompiled, possibly retested, and rereleased.

Portability

Use Typedefs for Types

It's a good idea to typedef int8, int16, int32, int64, float32, float64, uint8, uint16, uint32, uint64, etc., instead of assuming it'll be done with int, long, float, and short.

Minimize Inlines

Minimize inlining in declarations or inlining in general. As soon as you put your C++ code in a shared library which you want to maintain

compatibility with in the future, inlined code is a major pain in the butt. It's not worth it, for most cases.

Compiler Dependent Exceptions

Using exceptions across the shared library boundary could cause some problems if the shared library and the client module are compiled by different compiler vendors.

Compiler Dependent RTTI

Different compilers are not guaranteed to name types the same.

Alignment of Class Members

There seems to be disagreement on how to align class data members. Be aware that different platforms have different alignment rules and it can be an issue. Alignment may also be an issue when using shared memory and shared libraries.

The real thing to remember when it comes to alignment is to put the biggest data members first, and smaller members later, and to pad with char[] so that the same structure would be used no matter whether the compiler was in "naturally aligned" or "packed" mode.

For the Mac there's no blanket "always on four byte boundaries" rule -- rather, the rule is "alignment is natural, but never bigger than 4 bytes, unless the member is a double and first in the struct in which case it is 8". And that rule was inherited from PowerOpen/AIX.