C++ Coding Standard

Last Modified: 2009-03-22

Original by Todd Hoff
Adapted by AK

aleksey.krutko@gmail.com / http://www.---.com/

Introduction

Standardization is Important

It helps if the standard annoys everyone in some way so everyone feels they are on the same playing field. The proposal here has evolved over many projects, many companies, and literally a total of many weeks spent arguing. It is no particular person's style and is certainly open to local amendments.

Good Points

When a project tries to adhere to common standards a few good things happen:

- Programmers can go into any code and figure out what's going on.
- New people can get up to speed quickly.
- People new to C++ are spared the need to develop a personal style and defend it to the death.
- People new to C++ are spared making the same mistakes over and over again.
- People make fewer mistakes in consistent environments.
- Programmers have a common enemy :-)

Bad Points

Now the bad:

- The standard is usually stupid because it was made by someone who doesn't understand C++.
- The standard is usually stupid because it's not what I do.
- Standards reduce creativity.
- Standards are unnecessary as long as people are consistent.
- Standards enforce too much structure.
- People ignore standards anyway.
- Standards can be used as a reason for NIH (not invented here) because the new/borrowed code won't follow the standard.

Discussion

The experience of many projects leads to the conclusion that using coding standards makes the project go smoother. Are standards necessary for success? Of course not. But they help, and we need all the help we can get! Be honest, most arguments against a particular standard come from the ego. Few decisions in a reasonable standard really can be said to be technically deficient, just matters of taste. So be flexible, control the ego a bit, and remember any project is fundamentally a team effort.

Standards Enforcement

First, any serious concerns about the standard should be brought up and worked out within the group. Maybe the standard is not quite appropriate for your situation. It may have overlooked important issues or maybe someone in power vehemently disagrees with certain issues :-) In any case, once finalized hopefully people will play the adult and understand that this standard is reasonable, and has been found reasonable by many other programmers, and therefore is worthy of being followed even with personal reservations.

Failing willing cooperation it can be made a requirement that this standard must be followed to pass a code inspection. Failing that the only solution is a massive tickling party on the offending party.

Resources- Take a Look!

General

Book Recommendations

Names

Make Names Fit

Names are the heart of programming. In the past people believed knowing someone's true name gave them magical power over that person. If you can think up the true name for something, you give yourself and the people coming after power over the code. Don't laugh! 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.

General Rule For Names

Use upper case letters as word separators, lower case for the rest of a word.

By default this rule holds for every kind of names: Functions, Classes, Variables etc. Any exceptions will be adduced additionally.

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.

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

- For example: IsHitRetryLimit.

Class Names

- Use ‘C’ prefix for classes and ‘I’ for classes which are interfaces.
- Use General Rule for names.
- 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 CNameOneTwo
class CName
class IStorable
```

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.

Method Names

- Use General Rule for names.

Justification

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

Example

```
class CNameOneTwo
{
public:
    int DoIt();
    void HandleError();
};
```

Variable Names

In general, use following shema for variable names:

ScopePrefix + "_" + MeaningPrefix + General Rule

Table 1.

ScopePrefix	Description
m_	Class member
s_	Static variable
g_	Global variable

Table 2.

ScopePrefix	Meaning
a	Array
b	Boolean variable
c	Integer counter
f	Floating point variable
i	Integer variable
n	Integer number
p	Pointer
str	String

Example:

```
BOOL    m_bAllowDoingSomething;
UINT32  s_iSomeStaticInteger;
CFrog*  pLocalPointerVariable;
```

Class Attribute Names

- Attribute names should be prepended with the prefix '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 CNameOneTwo
{
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 CNameOneTwo
{
public:
    int StartYourEngines(
        Engine& SomeEngine,
        Engine& AnotherEngine);
};
```

Variable Names on the Stack

- Use General Rule

Justification

Example

```
int
NameOneTwo::HandleError( int ErrorNumber )
{
    int iError = OsErr();
    Time TimeOfError;
    CErrorProcessor ErrorProcessor;
    Time* pOutOfTime = 0;
}
```

The standard pointer notation is not entirely satisfactory because it doesn't look quite right, but it is consistent.

How do you handle statics? There's never a reason to have a static local to a function so there's no reason to invent a syntax for it. But like for most absolute rules, there is an exception, that is when making singletons. Use a "s_" prefix in this case. Take a look at Singleton Pattern for more details.

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& mrStatus;
};
```

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 name 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 CTest
{
public:
private:
    static StatusInfo ms_Status;
};
```

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;
```

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

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};
```

#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()
{
}
```

C++ File Extensions

In short: Use the *.h* extension for header files and *.cpp* 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 our choice.

Documentation

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 Extractable Headers

Use a document extraction system like Doxygen when documenting your code.

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.

As part of your nightly build system have a step the generates the documentation from the source. Then index the source using a tool like Lucene. Have a front end to the search so developers can do full text searches on nightly builds and for release builds. This is a wonderfully useful feature.

The next step in automation is to front the repository with a web server documentation can directly refer to a source file with a URL.

Comment All Questions a Programmer May Have When Looking at Your Code

At every point in your code think about what questions a programmer may have about the code. It's crucial you answer all those questions somehow, someway. If you don't, as the code writer, answer those questions, who will?

If you think your code is so clear and wonderful that nobody will have any questions then you are lying to yourself. You have a lot of tools at your disposal to answer questions:

1. A brain to think up the questions you should be answering. Why? Who? When? How? What?
2. Variable names.
3. Class names.
4. Class decomposition.
5. Method decomposition.
6. File names.
7. Documentation at all levels: package, class, method, attribute, inline.

The better you are at orchestrating all these elements together the clearer your code will be to everyone else.

Make Your Code Discoverable by Browsing

Programmers should be able to navigate your code by looking at markers in the code, namely the names and the directory structure. Nothing is more frustrating to than to have to look at pile of code and have no idea what it's organizing principles are.

Have a logicanl directory structure. Have directories called doc, lib, src, bin, test, pkg, install, etc and whatever, so others at least have some idea where stuff is. People use the weirdest names and lump everything together so that it it can be detangles. Clear thought is evidenced from the beginning by a directory structure.

Don't put more than one class in a file. Otherwise, how will others know its there when they browse your code? Should they really need to use search to find every last thing? Can't they just poke around the code and find it? They can if you organize your code.

Name your files after your classes. Why you name a file different than the class? How others are possibly supposed to know what's in the file otherwise?

Write Comments as You Code

You won't every go back later and document your code. You just won't. Don't lie to yourself, the world, and your mother by saying that you will.

So when you do something document it right then and there. When you create a class- document it. When you create a method- document it. And so on. That way when you finish coding you will also be finished documenting.

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: AK 960810: possible performance problem
// We should really use a hash table here but for now we'll
// use a linear search.
// :KLUDGE: AK 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 Implementers

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 Implementers

Class implementers 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.

Include Statement Documentation

Include statements should be documented, telling the user why a particular file was included. If the file includes a class used by the class then it's useful to specify a class relationship:

- ISA - this class inherits from the class in the include file.
- HASA - this class contains, that is has as a member attribute, the class in the include file. This class owns the memory and is responsible for deleting it.
- USES - this class uses something from the include file.
- HASA-USES - this class keeps a pointer or reference to the class in the include file, but this class does not own the memory.

Example

```
#ifndef XX_h
#define XX_h
// SYSTEM INCLUDES
//
#include // standard IO interface
#include // HASA string interface
#include // USES auto_ptr
```

Notice how just by reading the include directives the code is starting to tell you a story of why and how it was built.

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.

Complexity Management

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.

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 CFrog
{
public:
    void Jump();
};

class CBean
{
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 CHorse class was added:

```
class CHorse
{
public:
    void Jump();
};
```

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

Then someone updated CHorse so that its CJump 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 IJumpable
{
public:
    virtual void Jump() = 0;
};
```

IJumpable is a base class because other classes need to derive from it so they can get IJumpable'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 IJumpable. All the classes were changed to derive from IJumpable:

```
class CFrog : public IJumpable
{
public:
    virtual void Jump() { ... }
};

etc ...
```

We see an immediate benefit: we know all classes derived from IJumpable **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 CHorse or CFrog:

```
class CGui
{
public:
    void MakeJump( Jumpable* );
};

CGui Gui;
CFrog* pFrog = new CFrog;
Gui.MakeJump( pFrog );
```

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

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

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 behavior.

LSP is a very restrictive idea. It constrains implementers 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.

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.

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 CTestTaker
{
public:
    void WriteDownAnswer() { m_PaidTestTaker.WriteDownAnswer(); }
private:
    CPaidTestTaker 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!

Follow the Law of Demeter

The *Law of Demeter* states (Wikipedia): *An object A can request a service (call a method) of an object instance B, but object A cannot "reach through" object B to access yet another object to request its services. Doing so would mean that object A implicitly requires greater knowledge of object B's internal structure. Instead, B's class should be modified if necessary so that object A can simply make the request directly of object B, and then let object B propagate the request to any relevant subcomponents. If the law is followed, only object B knows its internal structure.*

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 CSunWorkstation
{
public:
    void UpVolume( int Amount ) { m_Sound.Up( Amount ); }
    CSoundCard m_Sound;
private:
    CGraphicsCard m_Graphics;
};

CSunWorksation Sun;

Sun.UpVolume(1);     // OK
Sun.mSound.Up(1);    // Don't do that!
```

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.

Using Design by Contract

1. **DO NOT PUT "REAL" CODE IN DBC CALLS.**. Dbc calls should only test conditions. No code that can't be compiled out should be included in Dbc calls.
2. Every method should define its pre and post conditions.

3. Every class should define its invariants.
4. Callers are responsible for checking preconditions. An object may not and is not required to test for assertion violations.
5. Method pre-conditions should be documented in a method's interface documentation.
6. Pre-conditions can be weakened by derived classes.
7. Post-conditions can be strengthened by derived classes.
8. Every Class Should:
    1. Develop its class invariants.
    2. Code its invariants and call them in its operations.
    3. Document its invariants in the class documentation.

   Yes, this takes a lot of work, but high availability is the system's __primary goal__, meeting this goal requires a lot of work and effort by each programmer.
9. Every Method Should:
    1. Develop a list of exceptions.
    2. Develop operation pre-conditions.
    3. Develop operation post-conditions.
    4. Code the exceptions, pre, and post conditions.
    5. Document the exceptions, pre, and post conditions.

   Yes, this takes a lot of work, but high availability is the system's __primary goal__, meeting this goal requires a lot of work and effort by each programmer.

Classes

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 '_'.

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

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.

```
// Description of the file.
//
// #include "XX.h" <BR>
// -llib
//
// @see something
//
// Created: JS-090101
// Updated: JD-100101
//
#ifndef XX_H
#define XX_H

// SYSTEM INCLUDES

// PROJECT INCLUDES

// LOCAL INCLUDES

// FORWARD REFERENCES

// TYPEDEFS

// Description of a class.
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=( const 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.

Prototype Source File

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

XX::XX()
{
}

XX::XX( const XX& )
{
}

XX::~XX()
{
}

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

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

//============================= OPERATIONS
==============================
//============================= ACESS
===================================
//============================= INQUIRY
=================================
/////////////////////////////// PROTECTED
///////////////////////////////
/////////////////////////////// PRIVATE
/////////////////////////////////
```

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.

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();
};
```

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.
//
// @exception   LibaryException The explanation for the exception.
// @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 != null )
// REQUIRE( to != null )
//
// 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

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);
```

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.

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:
    - constructing a member object
    - initializing a member attribute that is a reference

Example

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

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

private:
    int m_Val;
    String* m_pName;

    void Init()
    {
        m_Val = 0;
        m_pName = 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 )
        : m_Val( val ), m_pName( name ) {}
private:
    int m_Val;
    String* m_pName;
};
```

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 .

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.

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.

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 endorced 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.

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 );
```

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.

Thin vs. Thick Class Interfaces

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 re-released.

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).

In a Source file Indicate if a Method is Static or Virtual

In a source file you can't tell a method is static or virtual because this information is in the header file. Knowing this information in a source file is useful and can be communicated using comments:

```
// virtual
void
Class::method()
{
}

// static
void
Class::method()
{
}
```

Formatting

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 maintaining 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
```

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 3, 4, or 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

- Tabs aren't used because 8 space indentation severely limits the number of indentation levels one can have. The argument that if this is a problem you have too many indentation levels has some force, but real code can often be three or more levels deep. 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...

- 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. In general people have found 3 or 4 spaces per indentation level workable.
- 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;
```

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 ( 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 ) ? Func1() : Func2();
```

or

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

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 probability of declaring a variable when you actually mean to declare a pointer.

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           m_Dword
DWORD*          m_pDword
char*           m_pChar
char            m_Char

m_Dword         = 0;
m_pDword        = NULL;
m_pChar         = NULL;
m_Char          = 0;
```

Include *static* and *virtual* Key Words in Source File

If a method is made *static* or *virtual* in the header file then also include this information as a short comment in the source file. For example:

```
In .h file:
```

```
class CPopcornPopper
{
public:
    virtual void Pop( void );
    static CPopcornPopper* Singleton();
};

// In .cpp file:

// virtual
void
CPopcornPopper::Pop( void )
{
}// Pop

// static
CPopcornPopper*
CPopcornPopper::Singleton()
{
}// Singleton
```

A source file doesn't contain as much information as the header file because the virtual and static keywords are dropped from method implementations. This is very useful information to have when reading code, so we want to include it in the source file.

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
```

Exceptions

Create One Exception for Each Library

Creating very elaborate exception hierarchies is a waste of time. Nobody ends of caring and all the effort goes to waste. Instead, create one exception per library or namespace and have an exception reason within that exception to indicate the type of the exception.

For example, for your OS encapsulation library, make an exception called OsencapException.

By using just one exception you make it easy for people using your code to catch your exception. If you have thousand exceptions it is impossible for anyone to handle those. And as exceptions are added you will break existing code that thinks they are handling all your exceptions.

If you think you need to create derived exceptions the derive them from your libraries' base exception class. That allows your library users to still catch the base class exception if they wish.

Most exceptions are things the code can't do anything about anyway, so creating lots of exceptions to express every little thing that can go wrong with a separate class and is time confusing and confusing for the user. It's not necessary.

Make sure to document that an exception is being thrown in you method comment header.

Create a macro for throwing the exception so that you can transparently include __FILE__ and __LINE__ in the exception. This information is useful when debugging. You can also automatically take a time stamp and get the thread ID.

The exception should take a one or two string arguments so developers can include all the information they need in the exception without having to create a derived exception to add a few more pieces of data.

Include a stack trace of where the exception happened so you can log the stack trace.

Throwing an exception should take only one line of code. Don't uglify your code with tons of exception handling. Make a macro that looks like:

```
THROW_NAMESPACE_EX_IF ( cond, msg1, msg2, reason );
```

The condition causes the exception to be thrown when true. Msg1 and msg2 are local context. Reason is the specific error code for the exception, if you think you need it.

Don't worry about running out of memory errors. Go ahead and allocate memory in your exception code. If you are running out of memory that should be taken care of in the bad_alloc exception handler.

Selecting Between Exceptions And Asserts

If there is reason to believe that an important test could fail in the released product, it should not be tested with an assert, use an exception instead.

Two useful questions to ask yourself are:
1. Should this error ever happen?
2. Is the error so unexpected or potentially damaging that the system should failover if it is found?

When at the boundary of user input we should expect invalid parameters and thus should use an exception instead of DBC.

Exception Definition

There are as many definitions of exception as there are programmers:

The Clean Code Definition

Write a function as if everything worked.
Any error handling that prevents the code from looking
like this is an exception.

The Design by Contract Definition

When an operation is invoked with it's pre-condition
satisfied, yet it cannot return with its post-condition
satisfied.

There are those who say that exceptions should NOT be used to catch things like range errors. Here's why:

Range errors fall into a class of problem that are called **Programmer Errors**. Programmer errors are things that should NOT occur in a Bug Free Program. Ideally, range type problems are one of those things that should be caught during development. It is for these problems that we have **Design By Contract**.

You program defensively by putting lots of asserts in to make sure that your program is functioning as expected. Then when you've committed a bug, like not checking user input properly, allowing an invalid index into an array to be derived, the assert macro catches it, and the program does a graceful crash.

Exceptions, on the other hand, should be used to catch problems that would arise even in a Bug Free Program, i.e. Exceptional Circumstances. The most perfect program can still be afflicted by things like shortage of memory and other resources, communications errors, and file problems. When one of these things occurs, an exception should be thrown, and caught at a point where the program can either deal with the problem, or close gracefully.

The real theory behind exceptions is to force the programmer to anticipate things the programmer has no control over. Exceptions support the following kind of scenarios:

1. Logic error in the middle of a database transaction. An exception would allow the program a chance to leave the database in a consistent state. A trashed database can be very expensive.

2. Logic error in the middle of a program that is using a resource ike a modem and the program runs unattended. Exception handling would give the program a chance to hang up the modem connection, rather than sit there with an connection until it's discovered the next day, running up the phone bill.

3. Logic error in the middle of something like a word processor. Consider the user that has been working for a couple of hours with unsaved edits, and your assert message pops up, and the last two hours of work are pretty much lost - causing the user some misery. If you used exception handling, you could at least give the user a chance to salvage the unsaved document in a different file as part of the cleanup.

Design-by-Contract (DBC)

A design technique developed by Bertrand Meyer for producing "bug free" systems.

Design by Contract (DBC) views the relationship between a class and its clients as formal agreement, expressing each party's rights and obligation. Rights and obligations are determined by a class' specification. Correctness can only be determined in reference to an object's specification. Specifications, in DBC, are expressed through **assertions**.

Design by Contract is a development approach where a specification is associated with every software element. These specifications (or contracts) govern the interaction of the element with the rest of the world. A contract takes form as a set of preconditions, postconditions, and invariants that are run as the system executes. The contract is published in the comment block of each method.

Assertion

A boolean statement that should never be false and, therefore, will only be false because of a bug. Typically assertions are checked only during debug and are not checked during production execution, although they can be left in when performance and memory size are not issues.

Design by Contract uses three kinds of assertions: post-conditions, pre-conditions, and invariants. Preconditions and post-conditions apply to operations. Invariants apply to the class.

Pre-condition

A statement of how we expect the **world to be before we execute an operation**. We might design a pre-condition of the "square" operation as *this >= 0*. Such a pre-condition says that it is an error to invoke "square" on a negative number.

Post-condition

A statement of what the **world should look like after execution of an operation**. For example, if we define the operation "square" on a number, the post-condition would take the form *result = this *this*. The post-condition is a useful way of saying what we do without saying how we do it- in other words, of separating interface from implementation.

Invariant

An invariant is an assertion about a class. For instance, an Account class may have an invariant that says that *balance == sum(entries.amount())*. The invariant is **always** true for all instances of the class. "Always" means whenever the object is available for an operation to be invoked. During an operation invariants may not be satisfied, but invariants must hold when an operation has finished.

Is pop'ing an empty stack an exception?

When using a stack data structure is calling pop on an empty stack an exception? This is the standard use case for the endless discussions on what is and what isn't an exception. The upshot is there is not a "right" answer. In practice, as long as the library designer completely documents their choice the library user shouldn't make a mistake.

Let's apply our definition of exception.

The Clean Code View

If you write your application assuming everything works then is pop returning null something you wouldn't expect? It seems reasonable that if you ask for something it may not be there.

Before calling pop the client can call isEmpty to check if there's anything in the list. It's still possible in a multi-threaded environment for the stack to return null even if isEmpty returns true.

So don't use an exception. Pop returns data so it should return null when empty.

The Design by Contract View

Should the library designer have as a pre-condition that pop shouldn't be called when the stack is empty? From an implementer point of view this could seem reasonable. After all, why should an operation be called when there's nothing there? So, pop could assert.

But, library designers should always see their clients point-of-view. It's a bit extreme to cause a system crash because pop has an empty stack. There's nothing fundamentally or systemically wrong in this scenario that would justify an assert.

It appears the pre-conditions are satisfied for pop. Would any post-conditions not be satisfied? (this defines an exception in DBC). The stack should still be empty after the operation so no post-conditions would be violated. Thus, by definition an exception should not be thrown.

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.

Using RAII can help prevent many of not most of these type of errors.

Working With Libraries Using All Sorts of Different Policies

On many projects we have to work with C libraries that use error return codes, old C++ libraries that use error return codes, and newer C++ libraries that use exceptions. How should all these different approaches work together?

Ideally exceptions should be used where possible because that's the where all new code is going. But there are also new libraries being forced to use error return codes because the old libraries use them and application maintainers don't want you to use exceptions.

In C++ any method can throw an exception and you can't tell. In Java you know when you are not catching an exception. Not in C++. So what happens is that if a new library comes in and throws an exception, a critical program may start failing because an exception is not caught. This may make everyone use error codes instead of exceptions.

Don't give into this old timer bias. Error return codes are virtually useless because they are completely ignorable. Instead, an application should just wrap their code in a big try catch block that catches all exceptions. That way an application won't die with prejudice and C++ library writers can make use of exceptions in their design.

Templates

Template meta programming is too elite. Keeping templates simple has worked best. Just use templates for what they are good at, making a class work over different types. Not everything needs to be a template or needs to be perfectly generic. Because templates are so complex and so difficult to debug, it's best to use them only when necessary.

Namespaces

Create Unique Name Space Names

A namespace isn't of much use if it's not globally unique.

On way to make name spaces unique is to root them at some naming

There are two basic strategies for naming: root that name at some naming authority, like the company name and division name. The Java convention is to use the inverse of the company’s domain name for package names.

Personally that always seemed overkill to me, but does work in the large.

For internal packages just pick a short simple descriptive namespace that will likely prevent conflict within a project.

For externally visible code you'll probably need to the full domain style namespace just to be safe.

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.

For example, let's say we want to use boost's fabulous gregorian date library. In our class we want to return dates and use dates in methods. So our code ends up looking like:

```
//include all types plus i/o
#include "boost/date_time/gregorian/gregorian.hpp"

class CCalendar
{
public:
    boost::gregorian::date GetEventDate( void ) const;
    void SetEventDate( boost::gregorian::date DateOfTheEvent );
};
```

Clearly these are long names and are noisy and are pain to use. So you are tempted to put at the top of you class file:

```
using namespace boost::gregorian; // make namespace global for this
                                  // file
```

This would mean you could just use "date" instead of "boost::gregorian::date". That's nice. But you can't do that. If you do you are making the decision for everyone who uses your class as well. They may have a conflict, "date" is a very common name afterall.

So, don't use using in you header file, but you can use it in your source file. Because it's your source file you can make the decision to use using to shorten up the names. This strategy preserves most of the convenience while being a good citizen.

Create Shortcut Names

You can use **using** to create aliases or shortcuts, that is names that are more convenient to access than long namespace names.

For example:

```
namespace Alias = A::Very::Long::Namespace;
class CYourClass: public Alias::CTheirClass
{ ... }
```

Clearly this makes code both harder to read and easier to read. The code is harder to read because you as a programmer have to compile the alias directive in your head and know what it means whenever it is used in the code.

The code is easier to read because it is less complex. Complex namespaces are distracting and confusing. Even with the additional cognitive load, using namespace alias make code a lot easier to read.

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.

If you don't use const correctness from the start it can be nightmare to add it in later because it causes a chain reaction of needing const everywhere. It's better to start being const correct from the start or you probably won't be.

You can always cast always constness when necessary, but it's better not to.

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

Placement of the Const Qualifier

When you combine const with pointers it can become quite confusing as to what const means. For example, is "const int * = NULL" an unchangeable pointer or is the int that it points to unchangeable. It was suggested an alternate standard syntax that makes sense, so here it is:

```
int const = constant integer
int const * = pointer to constant integer
int const * const = constant pointer to constant integer
int const * const * = pointer to constant pointer to constant integ
```

This looks disturbingly different than the more common "const int _GLOBAL_CONSTANT= 5" syntax we are used to seeing. But, according to the standards, the qualifier modifies everything to its left with the special exception where the qualifier precedes the first type and modifies just that type. Since there is one generalized form that works for all occasions, it is more consistent to always put the qualifier to the right of whatever is qualified.

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

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.

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.

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;
};
```

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 ambiguity.

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.

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 )
{
    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() ) { ...
```

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.

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 0** is that even day later you or anyone else has no 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.

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
```

Always 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
```

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" SomeFunction();
#else
extern SomeFunction();
#endif
```

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

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 developers 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.

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 2GB of RAM now so we don't worry about buffer space any more :-)

Use the Resource Acquisition is Initialization (RAII) Idiom

RAII is a pattern useful in making sure allocated resources get freed. What kind of resources? Memory usually. Could be file descriptors or locks too. Anything that is allocated at one point and needs to be deleted at another point in your code.

C++ can help make this painless because of a feature, that your destructor is always called when an object leaves scope. This feature has caused the creation of the Guard idiom. A Guard can be wrapped around any resource so has to free the resource when the guard goes out of scope.

If you think you are always so careful to pair new and deletes that you don't need to use one of those pattern thingies then be sure someone else will be sitting beside you someday, searching for resource leaks in your code.

The acquisition phase happens at construction. You are supposed to do all the initialization in the constructor so you can do all the cleaning up in the destructor. And if you can guarantee through some different means that the destructor is always called then you can be pretty sure you won't leak.

Some initialization sequences can't be done in the constructor, particularly those using message driven state machines, but pretty much everything else can. And for those things use RAII. The justification is that it will save your butt from the many memory links that haunt large software products written in C++.

The key is to make sure the destructor is always called. The easiest way is to use a stack variable. That way you know the destructor will be called when leaving scope.

For pointers you can use one of the many smart pointer variants like auto_ptr. These will make sure your pointers are deleted when memory ownership is transferred or the scope is exited. You can even use them as member variables so you don't have to write code in the destructor.

If you can't use exceptions then this idiom isn't for you. As you are doing a lot of work in the
constructor you can have errors and you'll want throw an exception.

See http://www.hackcraft.net/raii/ for a better explanation of RAII.

Remove Trailing Whitespace

Avoid trailing whitespace in source code as it causes source control systems that show diffs to show unnecessary diffs that overwhelm the poor programmers ability to care.

Portability

Use Typedefs for Types

Since our applications supposed to become cross-platform, all general types should be redefined. We use upper case, '_' as word separator, and common prefix of our platform.

Example:

```
INT8
INT16
INT32

UINT8
UINT16
UINT32

BYTE
WORD
DWORD
QWORD

VOID_PTR
```

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.

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.

Popular Myths

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 perceives 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.