Light and Non-Controversial Coding Styles.

Leo Michelotti

February 14, 1997

The "hints" which follow are offered as simple rules for coding style. They are employed by many good C++ programmers, either because they are easy ways of avoiding obvious difficulties or because they make software more user-friendly.

Hint 1: Header files must prohibit multiple reading. It is essential that any header file, say MyClass.h, contain lines to prevent its being read more than once during a compilation. This is accomplished with two lines at the top of the file and one at the bottom.

```
#ifndef MYCLASS_H
#define MYCLASS_H

< ... body of header file goes here ... >

#endif // MYCLASS_H
```

The token defined should be an all caps version of the file name, written as shown.

Hint 2: Specifying argument names in header files. C++ allows that a function can be declared in a header file only by giving its signature.

```
Quaternion sqrt( const Quaternion& );
```

It also allows an argument name to be included, in which case it is ignored.

```
Quaternion sqrt( const Quaternion& x );
```

Regrettably, some design tools, such as Rational Rose, are sufficiently user-unfriendly as to require this form, even to the point of demanding that the token be the same in the function declaration and its implementation.

Of course, nothing prevents you from adding comments to the argument list.

```
Matrix pow( const Matrix& x,   // Base matrix
                const Matrix& y ); // Exponent matrix.
                                   // The Matrix returned is x**y.
```

This should be done anyway to reduce the need to consult software manuals.

Hint 3: Public members first. One of the darker features of C++ is that it permits sprinkling the keywords `public`, `private`, and `protected` throughout the class declaration, invoking each one any number of times. Some professional header files take advantage of this, much to the confusion of their readers. The preferred order for declaring members of a class is to put all public members first, followed by all protected, and all private members. (A need may arise to violate this convention, but such occasions should be rare.)

Hint 4: Inline functions. There are three ways of specifying that a class member function is inline. The first is to place the body of the function directly after its declaration in the header file. This is never done unless the function is very short: one or two lines at most. A better method is to place inline functions at the *bottom* of the header file, after all class declarations are complete, using the keyword "inline." The third method is to place the bodies of inline functions within a separate file identified with the suffix ".icc". This last method is the only one officially recommended by the committee.

Hint 5: Capitalize class names, but not instances or members. This is a style that is both popular with and violated by professionals. Names of classes should begin with a capital letter; variables should begin with a small letter.

```
SymplecticIntegrator g(5);
Hamiltonian h;
double t1 = 0.0, t2 = 1.0;
...
double answer = h.eval( g, t1, t2 );
```

Of course, the C++ keywords `double`, `int`, `char`, and so forth have not been capitalized, even though they syntactically look like instantiators of objects. Ignore that, and move on.

The lower-case convention for variables applies to member data and functions of a class as well, as in "`h.eval`" above. (However, see Hint 9, written below, concerning private member data.) A violation of this rule, sometimes appearing in professional C++ code, is to capitalize public member functions of a class.

```
double answer = h.Eval( g, t1, t2 );
myWindow.Paint();
```

However this practice is not recommended here.

Whether global functions should be capitalized depends on the context. Clearly,

```
Matrix x, y;
y = sin(x);
```

but *not*

```
Matrix x, y;
y = Sin(x);
```

is the correct style, because it agrees with the traditional use of "`sin`" applied to `double` variables (see Hint 8). On the other hand, if one has a global function of one's own

```
Complex Y( int l, int m, double theta, double phi );
double Zlorfik( Complex z );
...
u = Zlorfik( Y( 3, -1, M_PI/4.0, M_PI/2.0 ) );
```

there is nothing wrong, and there even may be some advantage, to capitalizing its name. Similarly, one might argue that "`Hamiltonian H;`" would have been a better choice than "`Hamiltonian h;`" because of the context: Hamiltonians are normally symbolized by a capital latter. In the final analysis, use your own judgement, discretion, discernment, wisdom, humility, and foresight in making such monumental decisions for special cases.

Hint 6: Use descriptive (English) identifiers. Most names of data and functions should provide clues as to what data are represented or what the function does. Seeing "`x = ourWorld.mt()`" in a program is not as informative as seeing "`todaysTemp = ourWorld.meanTemperature()`." There is a fine point on this in the context of international collaborative work: in what language is this meaning conveyed? We suggest that this should always be English.

Hint 7: Separating words in names. Descriptive names are frequently formed by catenating (English) words. If a variable represents the "average population density" of a region, then the name "`averagepopulationdensity`" is much more descriptive than something like "`apd`." However, it is also long enough to hinder reading. The constituent words in lengthy, descriptive names should be easily separated visually. One method employs capital letters, as in "`averagePopulationDensity`"; another inserts underscores between the words, "`average_population_density`." Both techniques are reasonable, and each can claim its zealous defenders. It does not matter greatly which one you use, but they should not be mixed within the same file.

There *are* usages for which the underscore is more legitimate. One such is the separation of well know acronym prefixes from words. For example, a "`class QCD_Particle`" may be quite different from a "`class NASA_Particle`." Here the prefix probably indicates a software package containing the class. This would be accomplished better with namespaces, but some compilers do not yet recognize this C++ feature.

Another frequent usage is to replace a period with an underscore in something like a file name.

```
ofstream theAnswers_dat( "theAnswers.dat" );
```

The name of the output file stream object mirrors the name of the file itself.

Hint 8: Satisfy obvious expectations for familiar symbols. One of the important advantages of object-oriented programming is providing a familiar "look-and-feel." Whether you get into a Chevrolet or a Mercedes-Benz, you expect the key, wheel, accelerator, and brake to perform certain well defined and familiar tasks. The familiarity of the environment means that you don't need detailed instructions to drive different model cars. To use an example closer to the point, anyone who runs a program under Microsoft Windows expects the **File** menu to do certain things. A programmer who placed editing functions within the **File** menu item would be guilty of violating "look-and-feel" compliance. The same hold true for any class library in which familiar operators are overloaded or familiar names are used for functions. If it makes sense to have an `X::operator+( const X& )`, then this operator should satisfy the expected commutative and associative properties. To machine precision, "`( x + y ) + z`" should produce the same result as "`y + ( x + z )`." As another example, an `X::operator+=( const X& )` should be defined only if `X::operator+( const X& )` exists and "`x += y`" means "`x = x + y`". One should not, for example, use "`+=`" to define the operation of pushing entities onto a stack. (That this is wrong would be indicated immediately by the fact that one should not be able to "add" to different kinds of objects together.) On the other hand, if one does have a `Stack` object, a member function named `Stack::Push( const X& x )` should put its argument on the top, not the bottom, of the stack. This is what is expected, and the expectation should not be violated. Similarly, one should not come up with new, unnecessary names or interfaces for functions that possess old, familiar analogs: "`x*x*x`" may be evaluated as "`pow( x, 3 )`," but it should not be evaluated as "`pow( 3, x )`,", "`power( x, 3 )`," or "`x.Exponent( 3 )`."

Unfortunately, very influential, professional computer scientists violate this rule frequently. Any physicist who has done three-dimensional graphics programming knows that, contrary to the expectation of all natural scientists, the computer scientists devised a world of left-handed coordinate systems in which to establish their conventions. Similarly, the keywords "map" and "vector" in the Standard Template Library refer to objects that possess very few of the properties familiar to any physicist who uses these words. Computer scientists seem uninterested in checking the sensibilities of natural scientists before devising their incantations.

Hint 9: First character in private member data name is '_'. A popular style for defining the names of class data members is to precede the definition with an underscore. Less frequently this convention is extended to private member functions also, as in

```
class X {
  private:
    double _u;
    double _foo( double );
  public:
    void bar();
};
```

The reason for doing this is to identify easily member data in the body of a function.

```
void X::bar()
{
  double y;
  < ... 10,389 lines of code later ...     >
  _u = 3.14*_foo( _u );
  x = _u*_u;
  < ... 8,114 more lines of code ... >
}
```

(Actually, anyone writing one function of such length should think seriously about finding another vocation.) For understandability, it is useful to recognize instantaneously that the tokens buried in the middle of this code refer to members of the class X.

There are two negative aspects to this much used convention. First, it is ugly, but that can be overlooked. Second, and more seriously, the C++ language reserves certain compiler dependent keywords which begin with an underscore. If the programmer attempts to redefine these particular tokens, errors will occur. Some people get around this by attaching the underscore to the end of the name rather than the beginning.

```
void X::bar()
{
  double y;
  < ... 10,389 lines of code later ...     >
  u_ = 3.14*foo_( u_ );
  x = u_*u_;
  < ... 8,114 more lines of code ... >
}
```

It is also possible to prefix a name with a descriptive character, such as '`s_`' for static variables, '`sf_`' for static functions, or '`d_`' for private member data. While possibly useful, this style tends to become ugly and cumbersome very quickly.

Hint 10: Object counting. A poor man's form of memory leak checking is to count objects. A static data member, `objectCount`, is included in the class declaration and initialized to zero in its implementation file. It is incremented by each constructor and decremented by the destructor. A class's `objectCount` can be monitored at any pertinent point in a large program and possible memory leaks exposed. The utility of this simple little trick should not be underestimated. With a slight modification – *viz.*, not decrementing the count upon object destruction – it also provides a mechanism for associating a serial identifier to every instance of a class.

Hint 11: No global data outside a class. Global data should never be defined outside a class. The `static` mechanism allows the sharing of data between instances of one class. For example, in order to keep track explicitly of the number of instances of a class, one might define a static member, "objectCount," in the header file.

```
class X {
  private:
    static int objectCount;
    ...
  public:
    int howMany() { return objectCount; }
    ...
};
```

It would be stored and initialized globally somewhere in an implementation file,

```
int X::objectCount = 0;
```

and manipulated by the constructors and destructor.

```
X::X() {
  ...
  objectCount++;
}

X::~X() {
  ...
  objectCount--;
}
```

If "truly" global variables are needed — that is, variables outside the context of a class — chances are you are not thinking of your program correctly. However, let us assume that they are needed, for some reason. In that case, one can define a class which contains these variables and nothing else.

```
class UglyGlobals {
  static double r;
  static double theta;
  static double phi;
};
```

This is, if you will, the last gasp of the `COMMON` block mechanism.

Hint 12: Hidden use of the copy constructor. One might naively expect that a line

```
Complex z = 3.0;
```

would be equivalent to the two lines

```
Complex z;
z = 3.0;
```

but, in fact, they are *not* equivalent. The first form, being an instantiation with initial value, actually calls the constuctor with argument and is equivalent to

```
Complex z( 3.0 );
```

There is no invocation of an **operator=** in this form. The second form is equivalent to

```
Complex z();
operator=( z, Complex( 3.0 ) );
```

Apart from the extra operations involved, this distinction is of little consequence unless one is dealing with a handler idiom, such as envelope-letter, written in such a way that the copy constructor does not actually perform a copy. For example, consider the following.

```
int foo( const Zlorfik& x )
```

```
{
  Zlorfik z = x;
  < ... function manipulates z ... >
}
```

While it looks superficially as though `x` has been copied to `z`, it may not be. The instantiation actually invokes the copy constructor.

```
int foo( const Zlorfik& x )
{
  Zlorfik z( x );
  < ... function manipulates z ... >
}
```

If the constructor `Zlorfik::Zlorfik( const Zlorfik& )` is written so as to perform a shallow copy rather than a deep copy, then `z` and `x` will refer to the same data. That is, this will not copy the *data* of `x` into `z` but only its handle. Thus, when `z`'s data are manipulated, and if care was not taken (by the programmer of `class Zlorfik`) to create a new data image for `z` before manipulation, `x`'s data will be manipulated as well. This is obviously not what was intended.

To avoid this possible error, split the instantiation from the assignment.

```
int foo( const Zlorfik& x )
{
  Zlorfik z;
  z = x;
  < ... function manipulates z ... >
}
```

In this way, *if* **Zlorfik::operator=(const Zlorfik&)** *has been written so as to perform a deep copy*, the data have been disconnected from the argument before being manipulated. However, perhaps it performs a shallow copy as well. In such cases, a **Zlorfik::deepCopy** routine should be a class member, so that one can write

```
int foo( const Zlorfik& x )
{
  Zlorfik z;
  z.deepCopy( x );
  < ... function manipulates z ... >
}
```

and be totally, unambiguously safe. In any case, everyone should be aware of the differences between deep and shallow copying and the hidden problems they might produce.

Hint 13: Avoid ambiguity: overload those operators. Early C++ textbooks praised type conversion as a way to avoid proliferation of operator overloading. If one has defined the conversion `X::X( const Y& )` and an operator, `X operator+( const X&, const X& )`, then statements like

```
Y v;
X u, w;
...
u = v + w;
```

will work as expected: `v` is converted to an `X` type object, which is then plugged into the first argument of `operator+`. However, suppose that the conversion, `Y::Y( const X& )`, and an operator, `Y operator+( const Y&, const Y& )`, have also been defined. Or suppose that more conversions, say, `Z::Z( const Y& )` and `Z::Z( const X& )`, and an operator, `Z operator+( const Z&, const Z& )`, have been defined. In either case, the compiler will become confused; it can no longer uniquely interpret the phrase "`v + w`."

There are two ways around this: (1) If the problem is truly that an unwanted conversion is taking place, the keyword `explicit` can be used when declaring constructors in order to suppress automatic type conversion. Of course, this means that the automatic conversion will *never* occur, which may not be a desired effect. (2) Operators can be overloaded explicitly (not keyword) to account for all possible combinations of arguments that make sense. It is tempting to avoid doing this if not immediately necessary. The problem is that the "not immediately necessary" of today has a tendency to become the "crucial" of tomorrow.

Hint 14: Name uniformity: is it "complex" or "Complex"? Unfortunately, there is not uniformity among header files provided by vendors regarding the identifier associated with complex numbers. Some use "complex," others "Complex." As a stopgap measure, one can work around this by inserting a compiler directive.

```
#ifdef __GNUG__
  typedef complex<double> Complex;
#else
  typedef complex Complex;
#endif // __GNUG__
```

In this way a single token "Complex" is used throughout, but it is redefined upon compilation to have the appropriate meaning depending on which compiler is being used. This trick can also be used to satisfy your group's naming conventions when they are violated by third party software that you want to use.

Hint 15: Enclose conditionals in a block. C++ syntax allows the predicate of a "for," "if," "while," or "do" condition to be a single statement. Thus, lines like

```
for( i = 0; i < N; i++ ) DoSomething(i);
```

are valid and, in fact, used frequently. However, such a line of code is like an accident waiting to happen. Even worse would be to break the line and indent it, as in:

```
for( i = 0; i < N; i++ )
  DoSomething(i);
```

While this works correctly, the danger exists that, if the code is modified later, a careless (or tired, distracted, overworked, depressed) programmer will forget to add necessary braces.

```
for( i = 0; i < N; i++ )
  DoSomething(i);
  DoSomethingElse(i);
```

This, of course, would not work correctly if what is intended is,

```
for( i = 0; i < N; i++ ) {
  DoSomething(i);
  DoSomethingElse(i);
}
```

Depending on the nature of the calculation, such errors can produce incorrect results for quite some time before being caught. In order to avoid them, it is safest either (a) to always included braces in conditionals, even when not necessary, or (b) at the minimum, write the statement on one line, so that at least there is some visual clue that reminds the programmer of what must be done.

Hint 16: Using conditionals to set variables. It is tempting to say, "Just don't do this." However, it is sometimes very convenient to write statements like

```
while( i = f(j) ) {
  j = g(i);
}
```

The intent here is to perform the statement "`j = g(i)`" until `f(j)` returns the value zero. Even more frequently, one uses this construction to traverse a data container.

```
XListIterator w( aList );
X* xp;
while( xp = w++ ) xp->doSomething();
```

Unfortunately, some compilers[1] have difficulty with using the argument of a conditional to set a variable. While all compilers should accept this syntax, it is possible to work around compilers that do not by using an extra pair of parentheses.

```
while((  i = f(j)  )) {
  j = g(i);
}
```

Besides keeping all compilers happy, this provides a *very* useful clue to any human reader[2] that the argument is intended to perform an operation. It confirms that the statement is indeed written correctly, that the intent of the author was *not*

```
while( i == f(j) ) {
  j = g(i);
}
```

for then the double parentheses would not be there. If the parentheses are separated sufficiently from the argument, recognition of this visual clue survives even complicated expressions.

```
while((  i = exp( 2.0*( f(j) + g(j) )  )) {
  j = g(i);
}
```

Finally, the form least likely to be confused for something unintended is obtained by forcing an extra comparison.

```
while( 0 != (  i = f(j)  ) ) {
  j = g(i);
}
```

Placing the constant on the left provides an early signal for the conditional's comparison part. While unambiguous, and acceptable to all compilers, this may be less transparent to the human reader, enough so as to nullify the original intent of making the assignment within a conditional statement.

Hint 17: Account for machine precision. Do not write conditionals in a way that will fail because of finite computer precision. For example, a statement like

[1]At least one.

[2]And a reminder to the programmer as well.

```
double x, y;
...
if( x == y ) DoSomething();
```

will not work properly when x and y differ at the level of machine accuracy. A more appropriate test might be,

```
if( fabs( x - y ) < epsilon*fabs(x) ) DoSomething();
```

which `epsilon` is, basically, machine precision.

The down side of this is loss of efficiency. It takes longer to perform the tests when more operations are involved.

Hint 18: Return an object from a function. Another way of putting this is, "Never return a pointer from a function." For example, never do:

```
X* acos( const X& xArg )
{
  X* theAnswer = new X;
  ...
  return theAnswer;
}
```

This makes the calling routine responsible for deleting the object. The possibility of a memory leak programming error is almost irresistible. *If at all possible, objects created using new should be deleted within the same function.* There may be rare cases in which this rule is too restrictive, but in general, violating it is very dangerous. The correct technique is to return an object.

```
X acos( const X& xArg )
{
  X theAnswer;
  ...
  return theAnswer;
}
```

This will invoke the X copy constructor to create a copy of the X object, `theAnswer`, on the return stack. When the original object goes out of scope, the copy remains (at least) as long as it is needed and will automatically be deleted when the calling function is finished.

There is an important exception to this rule. If the function acts as an object factory, producing a class instance as its result, it is crassly inefficient to require that the object produced be copied twice more before being used. Factories can be used, for example, to create objects from data in an ASCII file. A factory typically does return a pointer to the instance it creates, and it is expected that the calling routine is indeed responsible for deleting it at the appropriate time.

Of course, a rank beginner's mistake is to return a pointer to a variable declared in the body of the function.

```
X* acos( const X& xArg )
{
  X theAnswer;
  ...
  return &theAnswer;
}
```

This produces the oldest error in the book: `theAnswer` goes out of scope; what is returned points to data which could easily be overwritten before the caller makes use of it. (Anyone making this mistake also should pursue a new profession.)

Hint 19: Trace through new and delete. When an object is created on the heap with a call to *new,* a comment should be inserted on the same line indicating where the corresponding *delete* is to be found. This may be in a different module; in principle, it may even be in a completely different file, although that would not be recommended. For example, an object created as part of a constructor's operation may not be deleted until the destructor is invoked. A pointer global to a class could be set by many different functions implemented across multiple files. Nonetheless, the (human) reader should understand when and how an object is destroyed.

Conversely, it may be a good idea to inject comments by *delete* statements indicating where the corresponding *new* statements are to be found.

Hint 20: Make lines small enough not to wrap. It is annoying and confusing to have statements extend across the boundaries of an editing window. Lines of a program should be readable by any ASCII editor of reasonable width, say 80 characters.