

Elemental Base Idioms

by Stephen C. Dewhurst

It is an old observation that the best writers sometimes disregard the rules of rhetoric. When they do so, however, the reader will usually find in the sentence some compensating merit, attained at the cost of the violation. Unless he is certain of doing as well, he will probably do best to follow the rules.¹

This often-quoted advice from the classic guide to clarity in English prose often finds its way into texts on programming style as well, most recently in Kernighan and Pike's wonderfully curmudgeonly *The Practice of Programming*.² I approve of the quote and the chastening sentiment behind it. However, I find it unsatisfying because, taken out of context, it doesn't indicate why it is generally profitable to follow the rules of rhetoric, and how those rules come about. I've always preferred White's cowpath simile to Strunk's Olympian dictum.

The living language is like a cowpath: it is the creation of the cows themselves, who, having created it, follow it or depart from it according to their whims or their needs. From daily use, the path undergoes change. A cow is under no obligation to stay in the narrow path she helped make, following the contour of the land, but she often profits by staying with it and she would be handicapped if she didn't know where it was and where it led to.³

Programming languages are not as complex as natural languages, and our goal of writing clear code is not as difficult to achieve as that of writing clear sentences. A programming language like C++, however, is sufficiently complex that the community of C++ programmers has built up, over the years, a body of standard usage and idiom. Moo. The design of the C++ language is not proscriptive in that it allows much flexibility in how it may be used, but idiomatic usage of language features allows efficient and clear communication of a design. Ignorance or willful disregard of idiom is an invitation to misunderstanding and misuse. This article addresses a small subset of C++ programming idioms concerned with the design of base class interfaces. In deference to size limitations of this journal and the readers' patience, I'll limit discussion to several low-level issues in coding base class interfaces.

Base Class Basics: Totalitarianism and Tough Love

A base class is a contract between generic code written to the base class interface and a derived class that implements the contract. This organization is how we achieve code reuse and correctness most

¹ William Strunk and E. B. White, *The Elements of Style*, 3rd edition, Macmillan, 1979. The author of this quote is actually William Strunk alone, as it appeared in the original volume before its resurrection by White in 1959.

² Brian W. Kernighan & Rob Pike, *The Practice of Programming*, Addison Wesley, 1999.

³ E. B. White, *Writings from The New Yorker*, HarperCollins, 1990, p. 143.

effectively: we write generic code in terms of the properties advertised by the base class's public interface, then leverage that code with the design of substitutable derived classes. The key to making this work is to publish a contract in the base class that is clear to both the generic code and to derived classes that customize that code.

Note that we are most concerned with sharing the base class interface with its derived classes. We are not so concerned with sharing common implementation in the base class, because we typically achieve more reuse through leveraging entire subsystems than we do through factoring out common implementations of derived classes. Such factoring is good, in general, but it should not come at the expense of the base class interface.

It can be difficult to write substitutable derived classes. A derived class is substitutable for its base classes when it can be substituted for those classes in generic code and produce acceptable results. It must therefore implement the base class interfaces in such a way that it does not "shock"⁴ any current or future generic code written to those interfaces. Unfortunately, substitutability is often a low-level property of an inheritance relationship, and depends heavily not only on the abstract semantics the base class represents, but also on the precise interface employed to implement those semantics.

You don't do derived class designers a favor by giving them too much freedom. It's far better to permit them to exercise creativity only within bounds acceptable to the generic code they customize. An important goal of a base class interface is to spell out these limitations unambiguously. To the extent possible, these requirements should be communicated in the C++ language itself, through the use of base class design idioms, rather than through the use of comments. Comments are essential, but are generally over-used and under-maintained.

The Elements of Base Class Style

In homage to Strunk and White, let's put our idioms into the bold style of their rules of usage.

Base classes have virtual destructors.

I mean, really. This is a subject that has been covered in almost every C++ programming text over the past decade and one-half. First, there is no better documentation that a class is, or is not, intended for use as a base class than the virtualness of its destructor. If the destructor isn't virtual, it's not a base class. Period.⁵

Publication of the standard has made this advice even more compelling. First, destroying a derived class through its base class interface now results in undefined behavior if the destructor is not virtual.

```
class B { public: ~B(); };
class D : public B { ~D(); };
//...
B *bp = getABofSomeSort();
delete bp; // undefined behavior!
```

⁴ Cline & Lomow, *C++ FAQs*, Addison Wesley, 1995. This book is now available in a new edition.

⁵ Readers familiar with the C++ standard library will note exceptions to most of these idioms in the standard. Remember that the standards committee consisted of an international collection of C++ experts that labored for nine years on its design. "Unless [the reader] is certain of doing as well..."

Chances are, you'll just get a call of the base class destructor for the derived class object: a bug. But the compiler may decide to do anything else it feels like (dump core? send nasty email to your boss?)

On the positive side, having a virtual destructor in a base class allows you to achieve the effect of a virtual static member function call. Virtual and static are mutually exclusive function-specifiers, but the most specialized member `operator delete` should be invoked during a deletion, particularly if there is a corresponding member `operator new`.

```
class B {
public:
    virtual ~B();
    void *operator new( size_t );
    void operator delete( void *, size_t );
};
class D : public B {
public:
    ~D();
    void *operator new( size_t );
    void operator delete( void *, size_t );
};
//...
B *bp = getABofSomeSort();
delete bp;
```

Thanks to the virtual destructor in the base class, we will invoke the member `operator delete` in "the scope of the dynamic type of the class." That is, we'll probably invoke the member `operator delete` from within the derived class destructor. Neat. Virtual statics.

Partition the member functions.

Whether a member function is non-virtual, virtual, or pure virtual communicates to derived class designers how it is should be customized.

A non-virtual function specifies an invariant over the hierarchy (or sub-hierarchy) rooted at the base class. Derived class designers can not override, and should not hide non-virtual functions. The rational for this rule is basic and straightforward: to do otherwise would defeat polymorphism.

A polymorphic object has a single implementation (class), but many types. From our knowledge of abstract data types, we know that a type is set of operations, and these operations are represented in an accessible interface. For example, a `Circle` is-a `Shape`, and should work in an unsurprising and consistent fashion with code written to either of its interfaces.

```
class Shape {
public:
    virtual ~Shape();
    virtual void draw() const = 0;
    void move( Point );
```

```

    //...
};
class Circle : public Shape {
public:
    Circle();
    ~Circle();
    void draw() const;
    void move( Point );
    //...
};

```

The designer of `Circle` has decided to hide the base class `move` function (perhaps the base class assumes that the `Point` is an upper corner, but the version for `Circle` uses the center). Now it's possible for the same `Circle` object to behave differently depending on the interface used to access it.

```

void doShape( Shape s, void (Shape::*op)(Point), Point p )
    { (s->*op)( p ); }
Circle *c = new Circle;
Point pt( x, y );
c->move( pt );
doShape( c, &Shape::move, pt ); //oops!

```

Virtual and pure virtual functions are the mechanisms used to specify type-variant implementations. In the case of pure virtual functions, of course, the derived class is required to provide an implementation if it is to be concrete. Note that with virtual functions, overriding in the derived class assures that there will be only a single implementation--and therefore a single set of behaviors--available for a particular object at runtime. Therefore the behavior of the object is not dependent on the interface used to access it. (Of course, virtual functions can be called in a non-virtual manner through use of the scope operator, but this is not a property of the design of a class's interface.)

Don't make template methods too flexible.

The template method pattern⁶ gives us level of control between that of non-virtual and virtual functions. We specify an invariant algorithm as a non-virtual member in the base class. However, this non-virtual function allows customization of certain steps of its algorithm by derived classes. Typically, the algorithm invokes protected virtual functions that may be overridden in derived classes.

Taking our base class design idioms together, it's instructive to see how much design constraint we can communicate to derived class designers through idiom alone.

```

class Base {
public:
    virtual ~Base(); // I'm a base class
    virtual bool verify() const = 0; // you must verify yourself

```

⁶ Gamma, et al., *Design Patterns*, Addison Wesley, 1995, p. 325.

```

    virtual void doit(); // you can do it your way or mine
    long id() const; // live with this function or go elsewhere
    void jump(); // when I say "jump," all you can ask is...
protected:
    virtual double doHowHigh() const; // ...how high, and...
    virtual int doHowManyTimes() const = 0; // ...how many times.
};

```

As we discussed in the previous installment of this column, many novice designers erroneously assume that a design should be maximally flexible. These designers often make the mistake of declaring the algorithm of a template method to be virtual, assuming that the additional flexibility would benefit derived class designers. Wrong. Designers of derived classes are helped most by an unambiguous contract in the base class. If generic code is expecting a particular general behavior from a template method, than that must be respected and implemented by derived classes.

Don't confuse overloading and overriding.

From a design point of view, function overloading is used to express an abstraction; a set of overloaded functions implements the same abstract operation with different arguments. Function overloading is "syntactic sugar." Mechanically, function overloading can occur only among functions that have the same name and are declared in the same scope.

From a design point of view, overriding is a modification of the behavior of a base class function in a derived class while retaining the base class interface. Mechanically, overriding can take place only when a derived class member function matches the name and signature of a base class virtual function.

```

class B {
public:
    void f( int ); // f is overloaded
    void f( double );
    void g();
    virtual void h();
};

class D : public B {
public:
    void g( int ); // B::g is not overloaded or overridden
    void h(); // B::h is overridden
};

```

Overloading and overriding have nothing to do with each other either from the design point of view or from the mechanical point of view.

Don't overload virtual functions.

What's wrong with the following base class fragment?

```

class Thing {

```

```
public:
    //...
    virtual void update( int );
    virtual void update( double );
};
```

Consider a derived class produced by a designer who has determined that only the integer version of `update` requires different behavior in the derived class.

```
class MyThing : public Thing {
public:
    //...
    void update( int );
};
```

Here we have an unhappy confluence of overloading and overriding--which have nothing to do with each other. The result is similar to that of hiding a base class non-virtual; the behavior of a `MyThing` object will vary depending on the interface used to access it.

```
MyThing *mt = new MyThing;
Thing *t = mt;
t->update( 12.3 ); // OK
mt->update( 12.3 ); // oops!
```

Don't give virtual functions default initializers.

This is really the same problem as overloading virtual functions. Like overloading, default argument initializers are basically syntactic sugar, used to change the interface of a function without adding new behavior.

```
class Thing {
    //...
    virtual void doitNtimes( int numTimes = 12 );
};
class MyThing : public Thing {
    //...
    void doitNtimes( int numTimes = 10 );
};
```

The problems that arise result from a mismatch of static and dynamic behavior of an object, and are often very difficult to track down.

```
Thing *p = new MyThing;
p->doitNtimes();
```

The assumption is that there is some importance attached to doing it, by default, ten times for a `MyThing`, versus twelve times for a `Thing`.

Base classes are abstract.

From the design point of view, base classes should generally be abstract because they represent abstract concepts from the problem domain. Just as we don't want or expect to see abstractions wandering around in our physical space (imagine, for example, what a generic employee, fruit, or I/O device might look like), we don't want objects of abstract interfaces wandering around in our program space.

In C++, we also have practical concerns related to implementation. We are primarily concerned with slicing and associated issues such as the implementation of copy operations. Slicing results when a derived class object is copied to an object of one of its public base classes.

```
class B { /*...*/ };  
class D : public B { /*...*/ };  
void f( B );  
//...  
D d;  
f( d ); // slice!
```

Here we have "sliced off" both the derived class data and behavior. Note also that the remaining data resident in the base class part of the derived class may contain values that make sense only in the context of a derived class object.

But I'm Different

Programmers (like myself) are good at giving out technical advice, but often have a hard time following it. We preach against global variables, poor variable names, "magic" numbers and the like, but often insert them into our own code. This phenomenon confounded me for many years, until I read a magazine article that described the same phenomenon in adolescents. It is apparently common for adolescents to criticize risky behavior in others but, through a "personal fantasy," come to believe that they are themselves immune from any negative effects of engaging in that same behavior. As a class, then, programmers suffer from arrested emotional development. To this group of adolescents, I can think of no better advice to offer than some more of White's perfectly turned sentences.

We knew a countryman once who spoke with wonderful vigor and charm, but ungrammatically. In him the absence of grammar made little difference, because his speech was full of juice. But when a dullard speaks in a slovenly way, his speech suffers not merely from dullness but from ignorance, and his whole life, in a sense, suffers--though he may not feel pain.⁷

Even the best programmers and designers can't always write juicy code, but we can at least be aware, always, that our code and designs exist within a context of idiom. If we are aware of coding and design idioms, we can choose to stay within their narrow path, or make an educated decision to depart from them according to our needs. However, we can most often profit by staying within them, and we would be handicapped indeed if we were ignorant of them.

⁷ White, op. cit. p. 142.