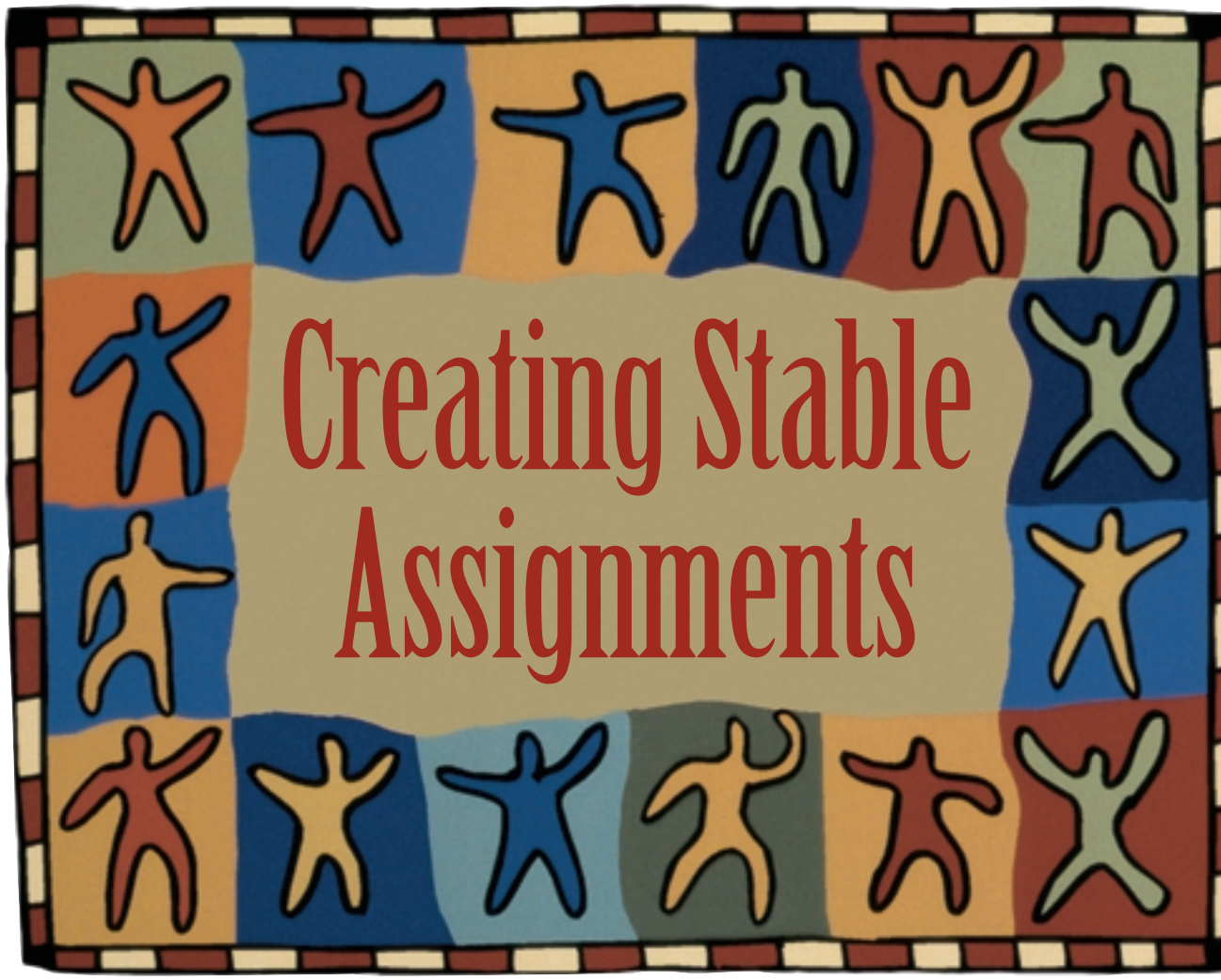

Jose Ortega/SIS

A paradigm for creating efficient, aesthetic self-referential code

Kevlin Henney

> The First Rule of Optimization: Don't do it.
> The Second Rule of Optimization (For experts only): Don't do it yet.
> —*Michael Jackson*[1]

When defining a copy assignment operator for a class, the problem of self assignment is often considered trivial or irrelevant, or is simply not considered. However, there are good reasons to consider what might happen if an object is assigned to itself, and there is also documented wisdom in this area. The standard form to handle self assignment was captured by Coplien as part of his Orthodox Canonical Class Form (OCCF)[2]:

```
type &type::operator=(const type &rhs)
{
    if(this != &rhs)
    {
        appropriate copying and release actions
    }
    return *this;
}
```

This is the basic schema you should seek to follow for all your assignment operators. A revisit to the Orthodox Canonical Class Form calls into doubt its general applicability for reasons of efficiency,[3] but before dealing with that issue specifically it is important to establish what is meant by Orthodox Canonical Class Form and what we hope to achieve with it.

ORTHODOX CANONICAL CLASS FORM

> **orthodox** *adj.* conforming with established standards, as in religion, behavior, or attitudes.[4]
>
> **canonical** *adj.* (of an expression, etc.) expressed in a standard form.[5]

The OCCF is a *recommendation*, not a *rule*:

> Programming standards must be valid for both newcomers and experts. This is often difficult to accomplish. We have solved this problem by differentiating our guidelines into rules and recommendations. Rules should almost never be broken by anyone, whereas recommendations are supposed to be followed most of the

time, unless there is a good reason not to. This division allows experts to break a recommendation, or sometimes even a rule, if they badly need to.[6]

It is not something to be followed slavishly, but has an important property: It works, it's safe and, as an idiom, it clearly communicates its purpose to readers. You can have greater confidence in something that is written following this form than in something that has not been.

Confidence and Intrinsic Quality Confidence is not something woolly that should be underestimated or ignored in the process of software development; it is essential. In the past, I have been presented with code that looked like it grew on a spaghetti tree,

If you can understand the code by its form, it will be easier for you to both have confidence in it and spot any mistakes.

greeted it with a perplexed expression, and then been told chirpily by its author "not to worry; it works."* Not.

Not only should a piece of software "work" (what this means is a whole topic in itself, but I'm sure you can come up with a number of plausible consensus definitions), but it should also *look* like it works. Commercial code is not written solely for the benefit of its author—although clearly the industry would empty out in the absence of any such gratification—and the idea that the only deliverable is a piece of executable code at the end of a waterfall development process should be greeted with the derision (as well as project failure) it deserves.

If you can understand the code by its form, it will be easier for you to both have confidence in it and spot any mistakes. If you cannot *see* that a piece of code is correct, how can you have confidence that it *is* correct? Executing it is not the answer: Dynamic bug hunting and bashing is a poor substitute for code that is internally well structured. The concept we are identifying here is that of *intrinsic quality*.[7]

Orthodoxy and Heresy There is a great deal of benefit in following a standard form for something that could otherwise give rise to obscure and unsafe behavior. One of the aims in programming is to be precise. If you are not being precise, you are being vague. If you are being vague, you don't need the help of a programming language—beer is a far better medium for this.

* I am reminded of an occasion when I was working on-site and needed some new code from someone back at the office. I said that I didn't expect it to be fully tested as he didn't have the right environment in which to do this. I was a little perplexed when the code arrived and failed to compile. Upon closer inspection, I understood why: There were basic syntax errors all over the place, and the code could never have been compiled. This was confirmed when I rang him to discover that because I had not expected full testing, he had taken this to mean that he didn't need to compile it either. In his vocab "compile" and "test" had somehow ended up as synonyms! Confidence was *not* high.

But as I said, this is a recommendation and not a doctrine or religious law. What if you feel an alternative solution is more appropriate? Will you be cast out from the gates of the C++ programming community and roasted over a code review? Should you just rebel outright, go off and establish your own orthodoxy? Nothing so dramatic is necessary: A comment will do. Just show that not having an explicit self check was considered, but deemed unnecessary as the code presented is already safe.

The important property of the canonical form is that it is based on some guarantees of behavior; a specification. Whatever code structure you settle on should satisfy this spec. To return to the idea of rules and recommendations, the OCCF is a recommendation, but the spec it is based on is a rule:

> **Rule 5.12** Copy assignment operators should be protected from doing destructive actions if an object is assigned to itself.[6]

Any departure from the OCCF must satisfy the same set of requirements. In short, it must still work.

EQUIVALENT FORMS The basic structure of the code offered by Glassborow can be summarized as:

```
type &type::operator=(const type &rhs)
{
    take a copy of rhs's resources
    release existing resources
    bind copy to self
    return *this;
}
```

Placing copy before release is required because this control flow ensures self assignment is not a problem. If &rhs is the same as this, we will waste a bit of time making a redundant copy of the current object, releasing current resources and then reassigning the copy. Perhaps the redundancy is not so aesthetically pleasing, but it is certainly safe and it will not be executed commonly enough to make it an issue. Perhaps the only thing missing is a comment (*note*: "comment," not "essay") stating that this code is safe for self assignment.

The applicability of this is for dynamically allocated representation—typically a single pointer to an object—that can be easily copied (based on the statically declared type) or cloned (based on the dynamic type—giving rise to the concept of *type-shallow* and *type-deep* copying).

What we have is the idea of behavioral (or *black box*) equivalence. Given the basic requirements outlined, this structure is substitutable for the OCCF.

Don't Optimize So, in the name of overall efficiency and correctness: no problems. Glassborow's motivation, however, is questionable:

> [T]he cost of making the check for self assignment is some kind of comparison and branch statement. Branches are bad news on pipelined architecture. If we can write code with fewer branches we should do so.[3]

This misplaced concern for code-level efficiency is the kind of thing that has been shown time and again as subordinate to optimiza-

tion through effective data structure and algorithm use.

Let's take a look at some code:

```
type &type::operator=(const type &rhs)
{
    rep_type *new_body = new rep_type(*rhs.body);
    delete body;
    body = new_body;
    return *this;
}
```

No branches? Take a look at a pseudo-assembler output:

```
    push        sizeof__rep_type
    call        __op_new                ; operator new
    move        new_body, result

    compare     new_body, null
    jmpifeq     postctor

    push        rhs + body
    push        new_body
    call        rep_type__ctorcp        ; rep_type::rep_type

postctor:
    compare     this + body, null
    jmpifeq     postdtor

    push        this + body
    call        rep_type__dtor          ; rep_type::~rep_type

postdtor:
    push        this + body
    call        __op_delete             ; operator delete

    move        this + body, new_body

    move        result, this
    return                              ; return *this
```

That's right; there are two implicit conditional branches:

- A null return from a new should not have a constructor called on it[†]; and
- a null pointer should not have a destructor called on it before being handed to delete.

In truth, this is a case of two rather than three branches, as opposed to zero or one. The difference? Look at everywhere there is a call instruction. This means we are calling four other functions, two of which we know handle heap management. Against that backdrop, the extra couple of instructions from an explicit self check look even less clock threatening than normal:

```
    compare     this, rhs
    jmpifeq     wayout                  ; if(this != &rhs)
    ...
wayout:
    move        result, this
    return                              ; return *this
```

[†] A relatively late but minor change to the Draft ISO C++ Standard clarifies that any version of operator new that has a throw spec is assumed not to return null to indicate failure. The global new is specified to throw std::bad_alloc—therefore, this null check will not be required. However, it will take a little time for implementors to catch up with this change.

The level of optimization we have achieved is what the phrase "a drop in the ocean" was intended to describe—if we used the word "optimize" anywhere near such code we would be deceiving ourselves.

Don't Optimize Yet The next claim to investigate is that of branches—specifically conditional branches—on pipelined architectures. Eliminating them based on some hope for optimization is as rational as not walking under ladders based on superstition—there are times when it is unwise to do so, e.g., when someone's already up the ladder with a can of paint, but that kind of judgment is not the same as superstition. So clearly we need to understand something about both conditional branching and pipelining before making a decision.

Some uses of conditional branching are simply the result of poor basic programming skills:

```
if(enabled)
    enabled = false;
else
    enabled = true;
```

The previous example illustrates the weak grasp the programmer has of logic. You don't need to be a genius to write and comprehend:

```
enabled = !enabled;
```

I agree with the basic tenet that we should write fewer control structures. A well-abstracted system tends to encapsulate control

flow within operations. Examples of this include polymorphism over explicit switch code, STL's combination of iterators and iterator algorithms, and the Enumeration Method pattern.[8]

But many conditional branches are a fact of life: It is difficult to eliminate them if they are intrinsic to a problem description. How many branches are there in the following code?

```
if(year % 4 == 0 && (year % 400 == 0 || year % 100 != 0))
    cout << "Leap!!!" << endl;
```

Three. One for every condition: Remember, C++'s built-in conditional operators are short circuiting.

An instruction pipeline contains instructions prefetched for execution. The many stages of an effective pipeline might include fetch instruction, decode instruction, calculate operands, fetch operands, execute instruction, and write operand result. Running these in parallel rather than in sequence is a very effective processor optimization. The only glitch appears to be that a branch in the control flow may invalidate the instructions in the pipeline: Although one branch is prefetched, what if the other is taken?

It would be surprising and unfortunate if such an elegant architecture had not been fully thought out —"bad news" indeed— but luckily, the impact of branches is anything but devastating, and pipelined chips sell and perform very well. One solution is to use a multi-stream architecture, i.e., you can hold more than one branch at a time. Branch prediction and delayed branching are more cerebral in their approach. Perhaps the simplest approach used is that the instruction stream following the branch instruction is loaded, i.e., what would have happened in the pipeline anyway.

How much of an impact does this last approach have on the code we've examined so far? None whatsoever. If you look at how the code is arranged, it is the common case that immediately follows the branch, and the uncommon one that must be branched on. If you wanted a rule concerning branches that took this into account it would be a simple one: *Place the commonly executed code nearest to the condition that tests for it.*

Interestingly, this is what many programmers tend to do already, but for readability reasons: *Given an if else, the if body should deal with the common case code, and the else body with the more exceptional occurrence. If they are equally valid, i.e., neither is exceptional, then the order is best determined by the most positive phrasing of the condition, i.e., the equivalent expression with the least contorted logic.*

It is often said that cleanly structured code tends to be more efficient than code whose guiding philosophy has been one of successive application of folklore optimizations. This case seems to vindicate that.

Relative Merits We have looked at behaviorally equivalent forms, but there is a stronger equivalence that is hinted at in the recommendation given above where I mention "equivalent expression" for a condition. For built-in types (and—one would hope—user-defined types) an example of strong logical equivalence would be that `!(a == b)` and `a != b` have the same meaning and are fully interchangeable.

Here's an interesting question: How much time is spent debating the following code snippets:

```
type &type::operator=(const type &rhs)
{
    if(this != &rhs)
    {
        appropriate copying and release actions
    }
    return *this;
}
```

Or:

```
type &type::operator=(const type &rhs)
{
    if(this == &rhs)
        return *this;
    appropriate copying and release actions
    return *this;
}
```

Which is the better implementation? These are equivalent in the sense that they have identical meaning, and one can be transformed into the other by a good compiler. If you are wondering which way such a compiler would lean, look back at some of the points we have discussed. That's right, the scruffy multiple return version is less optimal than the version that uses the structured programming form.‡

However, few compilers do that well enough so you are left with a separate set of concerns to balance. The common case is that the left and right hand sides of an assignment are not the same, so if your interest is either pipeline efficiency or layout, you would chose the first example. A direct translation of the second example into assembler tends to result in two jumps:

```
    compare     this, rhs
    jmpifne     postif              ; if(this == &rhs)
    jmp         wayout
postif:
    ...                             ; perform copying, etc.
wayout:
    move        result, this
    return                          ; return *this
```

For C++ it is important that common function exit code is shared as this can involve destructor calls which, if space is your concern, you would not wish to have duplicated at every return point. If you ask to optimize the second example for speed you will probably end up with duplicated code:

```
    compare     this, rhs
    jmpifne     postif              ; if(this == &rhs)
    move        result, this
    return                          ; return *this
postif:
    ...                             ; perform copying, etc.
wayout:
    move        result, this
    return                          ; return *this
```

‡ It has been said that in the light of modern optimizing techniques based on data rather than control flow, Niklaus Wirth wishes he had not included any jump statements (a function return statement and a loop exit) in Oberon (MODULA 2's successor) as the discontinuities introduced into the control flow are not only inelegant, but they thwart a number of optimizations.

It is interesting that we can arrive at the same conclusion from two completely different approaches; it says something about the relationship between forms at different levels. I personally side with those whose concern is the structure of the written code—my reasons for this are based on the belief that software development is an engineering profession, albeit an immature one.

> What if you feel an alternative solution is more appropriate? Will you be cast out from the gates of the C++ programming community and roasted over a code review?

There are a few people who need to be concerned with the machine level, but that figure is a lot less than the number who concern themselves with it.

STABLE INTERMEDIATE FORMS Returning to the proposed alternative code structure for assignment: Although arrived at from a faulty line of logic, it is sound. For those interested in patterns, what we have here is a language level pattern (better known as an *idiom*) that has a well-defined context, i.e., C++ copy assignment operator for an object structured using the Handle/Body idiom[2,9] (more generally, the Bridge pattern[10]) where the body is easily copied (either shallow or deep with respect to its type). The proposed configuration is something that works, meeting all the requirements for an assignment operator.

Exception Safety However, a pattern has three essential parts: context, forces, and configuration.[11] The conflicting forces that are listed for this pattern are at fault, and hence it is not a pattern. But since the context is valid and the configuration seems to have some merit, can we say something more about it? Alan Griffiths, in his role as editor, commented on Glassborow's solution:

> This has the added benefit of leaving the object in a consistent state if an exception is thrown during the clone operation. I'd rate this as more important than worrying about the different number of processor cycles required for each version.[3]

My only caveat to this—as we have shown—is that exception safety is the *only* benefit to this approach—as an issue, processor cycles are not even on the radar. In addition to the usual forces describing the requirements on a copy assignment operator, exception safety is the most important force resolved. Don't underestimate how significant exceptions can be for invalidating assumptions—and therefore code.[11] Let us examine the problem solved:

1. release existing resources
2. take a copy of rhs's resources
3. bind copy to self

What if an exception is thrown during step two? The object remains in existence, but it now has a chaotic and unstable state: Its resources have been released, but it still refers to them. What will happen upon destruction of that object? That's right, destruction of a completely different kind! Objects in an unstable state cannot be destroyed without spreading that instability to the rest of the program. However, there is no safe and consistent way to stop an object from reaching the end of its life. To put it mildly, this is a nontrivial issue.

The solution is to ensure that at every intermediate step the object has a coherent state, i.e., not only is the result of every macro change stable, but each micro change from which it is composed is also stable. This principle of Stable Intermediate Forms underlies successful software development strategies[12] as well as other disciplines of thought and movement, e.g., T'ai Chi.

1. take a copy of rhs's resources
2. release existing resources
3. bind copy to self

This sequence resolves the forces. It is also sufficiently general that it is possible to use this with the original OCCF—for instance, when writing a copy assignment for a class whose objects have a mixed style of representation.

A Pattern In summary, the many concerns facing a developer branch into myriad forces that fall somewhere between "challenging" and "daunting" in the software engineer's dictionary. Compared to other industries, software development sports a high number of people who can juggle. In this light, it is perhaps easy to see why. Close inspection of the configuration reveals a sound solution to a different problem, and a documentable pattern.

COPY BEFORE RELEASE

Problem:

- Ensuring copy assignment in C++ is exception safe.

Context:

- A class has been implemented as handle/body pair.
- The body is copyable—type shallow or deep as appropriate.

Forces:

- Any of the steps taken in performing the assignment may fail, resulting in a thrown exception. Partial completion of the steps may leave the handle in an unstable state.
- The result of assignment, successful or otherwise, must result in a stable handle.
- Self assignment must also result in a stable handle.
- After successful completion of the assignment, the handle on the left-hand side of the assignment must be behaviorally equivalent to the handle on the right-hand side.
- Assignment—successful or otherwise—must be nonlossy, i.e., no memory leaks.

Solution:

- Perform the body copy before releasing the existing body.
- Bind the body copy to the handle after releasing the existing body.

Resulting Context:

- The existing body is not deleted before the body copy has been attempted. Therefore, a failed body copy will not result in an unstable handle.
- Failed body release may still result in an unstable handle. However, throwing exceptions from destructors is a practice commonly cautioned against.
- The ordering accommodates safe self assignment at the cost of redundant copy.
- If the body copy preserves behavior equivalence, a successful assignment will preserve it for the composite handle/body object.
- The solution can be used in conjunction with the schema for copy assignment from the Orthodox Canonical Class Form.

The issue of a failed deletion is an interesting one, although not often one worth solving.[13] However, it is possible in this pattern with a minor tweak to the ordering. The solution is left, as they say, as an excercise for the reader. ◖

References

1. Bentley, J. *More Programming Pearls*, Addison–Wesley, Reading, MA, 1988.
2. Coplien, J. *Advanced C++ Programming Styles and Idioms*, Addison–Wesley, Reading, MA, 1992.
3. Glassborow, F. "The Problem of Self Assignment," *Overload* 19, Association of C and C++ Users, UK, Apr./May 1997.
4. *The Collins Concise Dictionary*, Collins, UK, 1988.
5. Borowski, E. J. and J. M. Borwein. *Dictionary of Mathematics*, Collins, UK, 1989.
6. Henricson, M. and E. Nyquist. *Industrial Strength C++: Rules and Recommendations*, Prentice Hall, Upper Saddle River, NJ, 1997.
7. Hatton, L. *Safer C*, McGraw–Hill, New York, NY, 1994.
8. Beck, K. *Smalltalk Best Practice Patterns*, Prentice Hall, Upper Saddle River, NJ, 1997.
9. Coplien, J. *Software Patterns*, SIGS Books, New York, NY, 1996.
10. Gamma, E. *et al. Design Patterns: Elements of Reusable Object-Oriented Software*, Addison–Wesley, Reading, MA, 1995.
11. Alexander, C. *The Timeless Way of Building*, Oxford University Press, UK, 1979.
12. Booch, G. *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Menlo Park, CA, 1994.
13. Sutter, H. "More Exception-Safe Generic Containers," C++ Report, 9(10):24–31, Nov./Dec. 1997.

Kevlin Henney is a Senior Technologist with QA Training in the UK. He can be contacted at kevlin@acm.org.