

Common Knowledge: A Matter of Judgment

A Matter of Judgment

Abstract

I defend myself against the forces of zealotry, and suggest that thinking is preferable to prejudice, at least while coding. Along the way we discuss using unusual switches to implement coroutines and playing with dolls.

My new book, *C++ Gotchas*¹, has had pretty positive reviews, but it contains one section that seems to send most everyone into paroxysms of rage, including the reviewers of the original manuscript. This is Gotcha #7, “Ignorance of Base Language Subtleties,” in which the book discusses various lesser-known aspects of the C-like part of C++, and demonstrates how these lesser-known facilities may be useful in certain circumstances. Many reviewers have taken such offense to my pointing out these potential uses of lower-level constructs that they feel compelled to insert comments like “...a hideous code snippet from a C++ compiler...” or “This falls under my ‘stupid code tricks’ category.” in the middle of an otherwise positive review. This installment of Common Knowledge will demonstrate why I’m right about this issue and everyone else is wrong.²

Received Wisdom vs. Experienced Judgment

Many years ago, we learned that gotos were considered harmful, that one-entry-one-exit structured programming was the true path to correct and maintainable software, and that functional decomposition with stepwise refinement was the correct way to attack a complex problem and render it into code. At the time it was very good advice, but like most good advice about software engineering and coding, it had a limited shelf life as the one-and-only way to do things. Not quite as many years ago, Kathy Stark and I wrote a C++ programming text that contained the following observation:

No single paradigm is suitable for solving all programming problems well. Programming requires engineering expertise but is not yet a science. Programming techniques need to be applied flexibly, with an eye to how well they suit the problem at hand. Blind application of the currently most popular paradigm is never a substitute for careful examination and thoughtful abstraction of a problem.³

That statement was made in reaction to the object-oriented frenzy of the time, when managers quoted from *The Structure of Scientific Revolutions*⁴ (without having read it), and ordinarily rational programmers were inclined to create hierarchies in which integers and graphical shapes shared a common base class. In effect, Kathy and I were suggesting that earlier approaches still had application to real problems, and should be used in conjunction with newer paradigms and techniques. We were suggesting that effective programming requires that a programmer actually apply experience and judgment to the problem at hand, selecting among various paradigms, idioms, and techniques for those most suitable to solving the problem and, if necessary, abandon idiom for an effective nonstandard solution. The existence of paradigm, patterns, and idiom is of inestimable value to the practice of programming, but it doesn’t absolve us completely from the necessity of thinking.

Common Knowledge: A Matter of Judgment

Good Advice Doesn't Always Scale Down

Therefore, the simple ideas behind 70's-style structured programming still apply in general. Code is generally more correct and easier to understand and maintain if control flow constructs have a single point of entry and a single point of exit. This applies to functions as well, and we generally prefer to enter at the top of a function body and exit from the bottom. In particular, the use of multiple return statements within nested control flow is nearly as bad as the use of `gotos` as far as complexity and maintenance are concerned.

But times change. Increasing use of the object-oriented paradigm transfers much control flow to the type system, through the use of dynamic binding. As a result, the average length of a function in a well-designed object-oriented program has decreased dramatically from that of a typical function produced as a result of functional decomposition. Additionally, the presence of exceptions forces C++ programmers to consider every explicit or implicit function call a potential alternative return from a function. Shorter function length renders multiple-path control flow less damaging, because if the entire function implementation is in view, even nonstandard control structures are obvious. Indeed, adhering to a piece of good, general advice can produce bad effects in specific cases.

```
Name *lookup( SymbolTable &tab, const Key &id ) {
    for( Iter i( tab ); !i.done(); i.next() )
        if( i.get()->id() == id )
            return i.get();
    return 0;
}
```

The code above violates a basic tenet of structured programming. However, application of that general advice in this specific case results in code that is less clear, potentially less efficient, and less likely to remain correct over time:

```
Name *lookup( SymbolTable &tab, const Key &id ) {
    Iter i( tab );
    Name *result = 0;
    while( !i.done() && !result ) {
        if( i.get()->id() == id )
            result = i.get();
        else
            i.next();
    }
    return result;
}
```

What's Everyone So Exercised About?

The example in *C++ Gotchas* that seems to bother people the most is an unusual—but perfectly standard—use of the `switch`-statement. Here's the offending code:

```
bool Postorder::next() {
```

Common Knowledge: A Matter of Judgment

```
switch( pc )
case START:
while( true )
    if( !lchild() ) {
        pc = LEAF;
        return true;
case LEAF:
    while( true )
        if( sibling() )
            break;
        else
            if( parent() ) {
                pc = INNER;
                return true;
case INNER:    ;
            }
            else {
                pc = DONE;
case DONE:    return false;
            }
        }
    }
}
```

The switch-statement is actually a multiway goto based on the integral value in the switch-expression, very similar to a FORTRAN computed goto and very unlike a Pascal case. After the switch-statement branches to the appropriate case, its work is over. The case labels may appear anywhere within the switch-statement, at any nesting depth. In the code above, the statement into which we're switching is a while-statement (rather than the more typical block consisting of a number of statements enclosed in curly brackets), and the case labels are distributed at various nesting depths within the while-statement. This `next` function implements an external iteration of a complex tree structure. That is, each time `next` is called, it moves the current position of the `Postorder` iterator to the following tree node in the postorder traversal sequence.⁵

In order to implement an external iteration, a `Postorder` iterator object has to keep track of the state of the iteration between calls to `next`. This is a non-trivial task, and there are a number of common ways to do this, including simulating nested function calls in data by implementing a stack of function activation records, using an internal iteration (that is, iterate through the entire tree in one function call) in order to thread the tree (that is, build a linear data structure of pointers to the nodes of the tree, arranged in postorder sequence). However, these approaches have associated costs in either complexity or runtime.

What we really want here is a coroutine. In its simplest form, a coroutine can be considered to be a kind of function that can return in the middle of its execution, and a subsequent call to the function will continue execution where it left off. That is, a

Common Knowledge: A Matter of Judgment

coroutine can detach (leave off execution) and resume (pick up where it left off). On the positive side, this approach allows us to implement the `next` algorithm in a straightforward and efficient way. On the negative side, in order to implement the coroutine semantics, we must embed our simple algorithm in a decidedly unstructured switch-statement.⁶

However, I claim that this use of the switch-statement is preferable to any other approach. It is certainly the most efficient. It is also the clearest, most maintainable, and best-encapsulated approach. Other approaches to implementation of the coroutine-like semantics for `next` require that significant structure be implemented for the `Postorder` class that will affect nearly every aspect of its implementation. Such a design decision introduces complexity not only in the use of the `Postorder` type, but also in all future maintenance of any aspect of the type. On the other hand, use of an unstructured switch-statement is restricted to the implementation of a single function, and a short function at that. The only individual who will come into contact with the switch-statement is the maintainer of the `Postorder` class, and that only when the `next` algorithm changes.

In short, I feel that criticism of the `next` function is based more on a knee-jerk reaction to the construct rather than a “careful examination and thoughtful abstraction of the problem.”

Playing With Dolls

Another school of thought might permit the unstructured switch-statement provided that it is suitably disguised. Typically, this involves reaching for a macro, or several macros:

```
#define CO_START switch(pc_.pc_){ case 0:;
#define CO_END }
#define CO_IMPLEMENT struct PC_{ long pc_; PC_():pc_(0){} } pc_;
#define CO_DETACH( e ) {pc_.pc_=__LINE__;return(e);case __LINE__:}
```

The `Postorder` class would include a `CO_IMPLEMENT` in order to declare a “program counter” data member, and the `next` function would take on a more structured appearance. Appearances can be deceiving.

```
bool Postorder::next() {
    CO_START
    while( true )
        if( !lchild() ) {
            CO_DETACH( true )
            while( true )
                if( sibling() )
                    break;
                else if( parent() )
                    CO_DETACH( true )
                else
                    CO_DETACH( false ) // !!!
        }
}
```

Common Knowledge: A Matter of Judgment

```
CO_END  
}
```

This approach reminds me of the medieval custom of practicing medicine on dolls so as to avoid seeing exposed flesh. As might be expected, that approach to medicine often resulted in faulty diagnoses. In our case, it has resulted in introduction of a few ad hoc coding rules and a subtle change in semantics. The use of these macros requires that the coder of a coroutine not follow the macros with a semicolon, and that a `CO_DETACH` not appear on the same source line as another `CO_DETACH`. (Of course!) From a practical perspective, we've traded a one-line comment that points out use of the unusual switch-statement semantics for a page or so of mechanism and usage rules that serve only to implement a single control construct in a single function.

The subtle change in semantics is indicated by the comment in the code above. The original implementation of `next` allowed it to be called without error even after it had reached the end of the iteration. This version does not, but the change in meaning is not obvious, and is likely not documented.

In effect, use of these macros hides the truth from the maintainers of `Postorder`, making the job of maintaining the code correctly that much less likely. We've made things harder for maintainers, without any compensating merit for users of `Postorder`, since users of `Postorder` never see the implementation of `next`.

Experienced Judgment Beats Prejudice Every Time

The use of an unstructured switch-statement may not appeal to everyone's aesthetic sensibilities, but in this case I claim that it is the most practical approach to solution of a bounded problem in the implementation of a single function. As I wrote in *C++ Gotchas*:

Effectively, while I do not recommend that this construct be commonly used, I recommend that it be commonly known. It should be available to the expert C++ programmer for those rare occasions when its use is required or preferable to other constructs. It's part of the C++ language for a reason.⁷

Copyright © 2003 by Stephen C. Dewhurst

¹ Dewhurst, S.C. *C++ Gotchas*, Addison-Wesley 2003.

² That statement, true though it may be, was intended as a joke.

³ Dewhurst, S.C. and Stark, K.T., *Programming in C++*, Prentice Hall, 1989, p. 3.

⁴ Kuhn, T.S., University of Chicago Press. The 3rd edition is current (1996), but the managers were quoting from the 2nd edition (1970).

⁵ The interested reader can find the source code for this example on the *C++ Gotchas* web site: http://www.semantics.org/cpp_gotchas. Go to the "Code" section and look at the file `iter.cpp` under Gotcha #7. By the way, this example also appeared in my first Common Knowledge column, in "Fungible Control Structures," *C/C++ Users Journal* 18, 12 (December 2000). I also received abusive mail about it then.

Common Knowledge: A Matter of Judgment

⁶ We don't actually have to use a switch-statement, but it's the simplest mechanism for our simple function. A real coroutine framework would have to consider the problems imposed by multiple, recursive coroutine instantiations.

⁷ *C++ Gotchas*, p. 16.