

Fungible Control Structures

by Stephen C. Dewhurst

Well. It's hard to know where to begin. The first two installments of this column appeared in another journal devoted to C++ programming which has, unfortunately, ceased publication. Prior to that, a similar column of mine appeared in yet another C++ programming publication that also ceased publication.¹ (Note to publisher: There is probably no causal relationship between the appearance of my articles and a journal's demise.)

In any case, this "Common Knowledge" column is intended to be "a regular summary of basic lore that every working C++ programmer should know--but can't always."² In general, we will discuss common programming and design techniques that should be part of every C++ programmer's toolkit. Previous installments discussed the pernicious effects of dynamic casting and the role of idiom in taming C++'s complexity in base class interface design. In this installment, we'll discuss the malleable nature of control structures, and how we can use that malleability to advantage for either efficiency or flexibility.

All code is interpreted. The only differences among source, object, and intermediate code representations are the partitioning up of responsibilities between the compiler and the runtime interpreter. How these responsibilities are meted out affects the properties of the translated code with respect to its efficiency, robustness, portability, and flexibility in the face of change.

We'll look at the common technique of substituting a data structure and small interpreter for inflexible compiled code. This approach is most useful in compiled languages, like C++, where we would like to add the flexibility of an interpreted language in some few portions of a program, without sacrificing whatever advantages we gain from compiled code in the rest of the program. As we'll see, the implementation of a program's semantics is quite fluid, and can be accomplished entirely at compile time, entirely at execution time, or partially at compile time and partially at execution time.

Singletons and Whining: A Bad Combination

A portion of my childcare responsibilities includes teaching design patterns in corporate settings to groups of new hires. I like to start off with the Singleton pattern,³ because its deceptive simplicity hides a lot of design flexibility and complexity. One of the many useful properties of Singleton is that it allows for polymorphic singletons, allowing new types of singletons to be added to a framework without recompilation. The basic Singleton implementation typically implements a lazy creation of the singleton object on first request.

¹ *C++ Report* and *The C++ Journal*, respectively.

² Herb Sutter, Editor's Corner, *C++ Report* (12, 5) May 2000.

³ Gamma, et al., *Design Patterns*, Addison Wesley, 1995, page 127.

```
class S {
public:
    static S *instance(); // get/create the singleton
    virtual void opl();
protected:
    virtual ~S();
    static S *instance_; // ->the singleton
    //...
};
S *S::instance() {
    if( !instance_ )
        instance_ = new S;
    return instance_;
}
```

For polymorphic singletons, a decision is made at the point of lazy creation as to what type of singleton is required. Typically, I'll scribble something like the following, hoping we can move on to other Singleton issues.

```
S *S::instance() {
    if( !instance_ ) {
        const char *s = getenv( "STYPE" );
        if( !s || s == "STDS" )
            instance_ = new StdS;
        else if( s == "NONSTDS" )
            instance_ = new NonStdS;
        else
            throw BadSType();
    }
    return instance_;
}
```

Fat chance. "No fair! No fair! You said we could decouple the derived singleton classes from the framework! I'm telling!" Soccer dads and corporate trainers know that it does no good to lose your cool in situations like this. "Of course we wouldn't really write this code in a production situation," I'll say in an even tone. "We'd probably just substitute a list or map for the if-else and do the conditional at runtime." Blank stares.

Nothing to Lose but Our Bonds

Suppose you want to price some bonds. These things happen. As it turns out, nobody really knows what

a bond is worth, but there are a lot of ways to guess what it might be worth. One approach that won't work is the use of a hard-coded conditional. Using such a conditional to choose among the "Official," "My," and "Guess" algorithms means that all these pricing algorithms must be known in advance, and the addition of a new pricing algorithm would entail source modification and recompilation.

```
class Bond {
public:
    //...
    Money pv() const; // present value of bond
    void setModel( const char *model ) // pricing algorithm
        { model_ = model; }
private:
    //...
    string model_;
};

Money
Bond::pv() const {
    if( model_ == "Official" ) {
        //...
    }
    else if( model_ == "My" ) {
        //...
    }
    else if( model_ == "Guess" ) {
        //...
    }
}
```

Another common and misguided approach is even worse. Suppose we decide to make the pricing algorithm part of a bond's type by making `Bond` abstract and deriving `OfficialBond`, `MyBond`, and `GuessBond` concrete types. This implies that to compare the value of a bond under two different pricing algorithms, we'd have to create two largely identical but separate `Bond` objects. Worse still, every time we add a new pricing algorithm we must expand our `Bond` hierarchy. If other aspects of a bond's behavior are also encoded in the bond's type, our hierarchy will expand as a product of the number of aspect of behavior. Hmm... Given this, perhaps our first approach is not so bad after all.

Let's go back to using a conditional to implement pricing algorithm selection, but let's wait until runtime to implement it so we can avoid coupling to specific pricing algorithms. Our plan is to use the Strategy

pattern⁴ to attach different bond pricing algorithms at runtime. The `Bond` class now has a separate pricing model object that encapsulates the current pricing algorithm.⁵

```
class Bond {
    //...
    Money pv() const { return model_->pv( this ); }
    void setModel( const char *m );
private:
    //...
    Model *model_; // ->pricing strategy
};
```

The pricing model class has some simple mechanism that allows it to be identified by a character string identifier, and to be cloned.

```
class Model {
public:
    virtual ~Model();
    virtual Money pv( const Bond * ) = 0;
    const string &name() const { return name_; }
    virtual Model *clone() const = 0;
protected:
    Model( const char *name ) : name_( name ) {}
private:
    static list<Model *> models_; // available models
    string name_;
};
```

All we have to do now is replace the original conditional code with a data structure (in this case, a list) and an interpreter (in this case, iteration through the list).

```
void Bond::setModel( const char *model ) {
    for( list<Model *>::iterator i( Model::models_.begin());
        i != Model::models_.end(); ++i )
        if( (*i)->name() == model ) {
            delete model_;
            model_ = (*i)->clone();
        }
}
```

⁴ *Design Patterns*, page 315.

⁵ In order to save space in these code samples, I've omitted any consideration of access protection issues.

```
}
```

Where do the models on the list of all models come from? Wherever. To avoid hard-coding the available models, we can have each model enter itself on the list of all models as a side effect of loading the file that contains the definition of the pricing model. This can take place either at static or (with some caution) at dynamic load time.

```
namespace {  
    class Official : public Model {  
        //...  
    } officialProto;  
    struct RegisterOfficial {  
        RegisterOfficial()  
            { Model::models_.push_back( &officialProto ); }  
    } registerOfficial;  
}
```

NonStandard Iterations

As smitten as I am by the C++ Standard Template Library, there are occasions when I find a non-STL container and iterator preferable. Consider an n-ary tree of fixed size. The tree is implemented as an array of nodes arranged in preorder sequence.

```
class Tree {  
public:  
    //...  
private:  
    Node *root_;  
    int size_;  
};
```

Each node has links to (potentially) three other nodes within the array: its parent, its sibling, and its leftmost child. These links are implemented as relative indexes within the array; that is, one adds the (possibly negative) index to the address or index of the current node to follow the link. A zero index is a null link.

```
class Node {  
public:  
    //...  
private:  
    typedef short Index;  
    Index parent_;  
    Index sibling_;
```

```

    Index lchild_;
    // other members...
};

```

Obviously, we'd like to traverse the tree, so we'll implement a variety of iterators over it. A preorder iterator is simple.

```

class Preorder {
public:
    Preorder( Tree &t )
        : cur_(t.root_), end_(t.root_+t.size_ ) {}
    void next() { ++cur_; }
    bool done() const { return cur_ == end_; }
    //...
private:
    Node *cur_, *end_;
};

```

Now it's childishly simple to perform a preorder traversal of the tree nodes.

```

extern Tree &t;
for( Preorder p( t ); !p.done(); p.next() ) {
    // do preorderish things...
}

```

What about other traversals? We have a perfectly useful algorithm for postorder traversal. It's easiest to express as a recursive algorithm.

```

postorder:
    save current position
    if( lchild )
        move to lchild
        postorder
    if( sibling )
        move to sibling
        postorder
    fi
fi
restore saved position
visit // do something with the node

```

It is fairly straightforward (pedagogic doublespeak meaning, "it took me all afternoon") to transform this recursive postorder algorithm into a non-recursive one.

```
postorder:
  loop
    if( lchild )
      move to lchild
    else
      visit // leaf node
      loop
        if( sibling )
          move to sibling
          break
        else
          if( parent )
            move to parent
            visit // internal node
          else
            return // done
          fi
        fi
      repeat
    fi
  repeat
```

This is enough to ruin anyone's day, and it's certainly not the kind of R-rated code you'd want to expose to a new hire. We already know the desired syntax.

```
for( Postorder p( t ); !p.done(); p.next() ) {
    // do postorderish things...
}
```

Unfortunately, our simple non-recursive postorder traversal algorithm implements an "internal" iteration; it traverses the entire tree in one go. What we'd like is an "external" iterator that returns the next node in the sequence with each call, and picks up where it left off on subsequent calls.

Coroutines? Why Don't They Just Make it Legal?

Remember coroutines? A coroutine has the useful property of being able to return to its caller from a particular point, and then resume execution at that point the next time it is called. Aging programmers use them to spice up their war stories while rocking on the front porch, but they have other uses as well. Unfortunately, most languages, C++ included, have no direct support for them. As is usual in C++, if the language is lacking a feature that we'd like, we'll just extend the language to include the feature. This is a common practice with features like pointers, functions, and numeric types where the built-in facilities of the language may not be adequate in a particular situation.

For our postorder iterator, we'd like to implement coroutine semantics. Unfortunately, C++ functions implement (surprisingly enough) function call semantics. The standard procedure in such situations is to design a class that takes the place of the missing language feature. In this case, we don't want a standard function activation record, but one that is more like the activation record for a coroutine.

```
class Postorder {
public:
    Postorder( Tree &t )
        : cur_( t.root_ ), pc_( START ) { next(); }
    void next();
    bool done() const { return pc_ == DONE; }
    //...
private:
    bool moveToParent() {
        return cur_->parent_
            ? (cur_+=cur_->parent_, true)
            : false;
    }
    bool moveToSibling();
    bool moveToLchild();
    Node *cur_;
    enum { START, LEAF, INNER, DONE } pc_;
};
```

The content of this artificial activation record is not only the "automatic" data local to the coroutine (a pointer to the current node) but also the program counter, since successive invocations of the coroutine will resume at different points within the code.

```
void
Postorder::next() {
    switch( pc_ )
    case START:
        while( true )
            if( moveToLchild() )
                ;
            else {
                pc_ = LEAF;
                return; //visit leaf
            }
    case LEAF:
        while( true )
```

```
        if( moveToSibling() )
            break;
    else
        if( moveToParent() ) {
            pc_ = INNER;
            return; // visit inner
case INNER:
            ;
        }
        else {
            pc_ = DONE;
case DONE:
            return; // done
        }
    }
}
```

In spite of the rather ghastly use of a switch, this implementation is fairly simple if one compares it to the original non-recursive algorithm from which it was derived. It's also faster and more space-efficient than other potential approaches involving simulation of a recursive function (using a stack of activation records) or threading a list through the n-ary tree. The approach of using an interpreted data structure in preference to a built-in control structure has given us both simplicity and efficiency in this case.

Interpret This!

There's not a lot of difference between control flow implemented as program text and control flow implemented as an interpreted data structure. The differences are chiefly those of flexibility versus efficiency; roughly the same tradeoffs we confront when choosing between interpreted and compiled code. Depending on when the relevant conditional information is available, we can choose to resolve the structure of flow control at compile time (using a technique like template metaprogramming⁶), when the process is loaded (a language primitive like if-else), or at runtime (interpreting a data structure). In this article we've discussed the more traditional transformations between language control structures and interpreted data structures, as template metaprogramming does not yet fall under the rubric of "Common Knowledge."

Compilers have been playing games with such things for years. It's common for a compiler to provide the capability to insert interpreted code within an otherwise compiled module, and it's common for an interpreter to implement "just-in-time" compilation of generally interpreted code. Well, programmers can play at this game too.

end of article

⁶ "Using C++ Template Metaprograms", *C++ Report*, Vol 7 No. 4, May 1995.