An Object-Oriented Language Engine

Once C programmers trying to learn C++ have mastered the basics of the language, and are working to fit themselves more snugly into the object oriented design mindset, a good pointer to pick up, is that whenever one is tempted to use a switch statement to manage a certain behavior of a finite set of variant records, one instead should consider encapsulating the records as objects and using run-time polymorphyism to manage the varying behaviors. Learning to spot situations such as this, where polymorphyism can improve readability and maintainability is one of the most important steps towards using C++ effectively.

The traditional implementation of interpreted languages engines in C features several areas where switch statements are used: in the tokenizer, parser, and in the run-time engine. The parser takes the user code and renders a terse run-time code which can be rapidly followed by the run-time engine. Call this code machine code, because although it is not exactly machine language, it is designed to be read more easily by the machine than by humans. This code usually involves storing the operations and operands of the program into a uniform format, most often either prefix or postfix form. Each step of the machine code is thus a variant record, with one field detailing the primary operation for the step from the set of operations that the run-time engine supports. The switch statement which drives such run-time engines simply reads the operation field and executes the appropriate instructions.

For example, the following sequence of statements in our imaginary user language

```
a := 2 + x
b := funct(a)
display(b)
```

might translate to the following machine code sequence:

```
-- Push the value "2" on the stack
-- Push the value of "x" on the stack
-- Pop two values, perform the add operation on them, and push the result
-- Pop one value and and assign it to "a"
-- Push the value of a on the stack
-- Pop one value, pass it to the routine "funct", and push the result
-- Pop one value and and assign it to "b"
-- Push the value of b on the stack
-- Pop one value, and display it
```

This description may seem convoluted to human eyes, but since each of the statements (call them steps) can be encoded as a single operation, followed by a fixed number of parameters for that operation, the actual encoding is uniform and brief, allowing for very quick interpretation.

The traditional C/C++ implementation of the engine for such code is would appear as shown in Listing 1. The big *switch* statement driving the engine should alert us to the potential for using, instead, the run-tyme polymorphic capabilities of C++ objects. The idea is to think of each *case* statement's code as a different method of performing a single function which is common to all the variants. In the above case, we can say that all of the case statements are

"executing the step as appropriate to the given operation." This indicates that the *switch* could be recast into a virtual function, *execute*, defined for a virtual base class *Operation*, and its descendants, each of which corresponds to an operation code for the machine code, and overrides its *execute* method accordingly, as in Listing 2.

We now have a class hierarchy embodying our model of a set of operation objects which can automatically manage their own execution through virtual functions, but how do we actually put this mechanism in place? How is the machine code program embodied, and how does the run-time engine iterate over the steps that make up a program? To enable the virtual function mechanism, the actions obejects should be accessed through pointers to the base class *Operation*. The machine code program consists of a series of steps, each describing a single operation, to be executed in order. This could be implemented as an array of pointers to *Operation*:

```
Operation *program_steps[];
```

During the translation from user code to machine code, this array is allocated and populated with dynamically created objects of class *Literal_number*, *Variable_value*, etc. and the run-time engine iterates over *program_steps*, invoking the *execute* method through each pointer, and thus performing the appropriate action for the actual object pointed to. One could encapsulate a machine code program, including the methods used to generate it from the user code, and to execute it, into a Machine_code class:

The *program_steps* array needs to be allocated large enough to store an *Operation* pointer for each step in the target machine code. Since this size is unknown prior to translation, the generated steps are better stored in a linked list, or with a dynamic array container class, but the above array will serve for illustration.

The *run* method for *Machine_code* would thus be something like the following:

```
void Machine_code::run()
{
        for (loop = 0; loop < num_program_steps; loop++)
                program_steps[loop]->execute();
}
```

Which is obviously much simpler and comprehensible than the monolithic *switch* block of the imperative implementation. The functionality for the various possible operations has been encapsulated into separate classes, rather than lumped together into a single functional unit. The execute methods for each class would be very similar to the code in the corresponding *case* blocks, as one can see in listing 4.

Note the necessary ancillary objects *expr_stack* and *symbol_table*. These are implemented using seperate container classes (a stack class for *expr_stack* and a, perhaps, a splay tree class for *symbol_table*) and are best declared as *protected* references to the actual objects in the *Operation* class declaration. These references can then be bound to the actual objects during the construction of the *Operation* base object portion of any of the descendant classes.

A further benefit of the object-oriented approach is that the body of machine code is no longer an amorphous block of memory (the *param_block* buffer of listing 1) which at run-time is treated a stream of data of different types and sizes which need to be picked through: an error-prone process. With the object-oriented approach, the selector of the correct procedure for the operation is predicated on the object type and thus transparently managed by the C++ virtual function run-time mechanism, and the parameters used by the engine in executing each step (such as the number of arguments passed to a function call operation) are smartly encapsulated in the private storage of each operation object.

The above model of a run-time interpreter is the basis of what could be a very extensible and maintainable language engine. The other necessary components, the lexical analyzer and the

parser, are also traditionally implemented with big switch constructs, and are also quite amenable to object design. The lexer would be based on an object-oriented finite-state machine (see sidebar), and there are several methods for implementing a C++ parser, one of which is detailed the *C User Journal* article, "Designing an OOP Compiler" (May 1994). The breakdown of solutions from huge monolithic structures such as the traditional switch into the small packaging of well-designed objects will greatly reduce coupling, and increase cohesion, leading to much simpler maintenance. These are the much-trumpeted goals of object-oriented design, which have been notoriously hard to attain in some application areas. Luckily, language interpretation is one area where objects lead to clear design improvements.

```
int curr_value;
char *param_block;
char *function_name;

//param_block points to the part of the condensed code buffer
following the action code

operation_code = *((int)param_block);
param_block+=sizeof(int);

switch (operation_code)
{
case Literal_number:
     expr_stack.push(atoi((char *)param_block);
     break;
case Variable_value:
     curr_value = symbol_table.get_variable_value((char
*)param_block);
     expr_stack.push(curr_value);
     break;
case Add_operation:
     curr_value = expr_stack.pop() + expr_stack.pop();
     expr_stack.push(curr_value);
     break;
case Function_call:
     num_passed_arguments = (int)param_block;
     param_block += sizeof(int);
     symbol_name = (char *)param_block;
     //the execute_function method pops the number of arguments
specified by num_passed_arguments
     curr_value = symbol_table.execute_function(symbol_name,
num_arguments);
     expr_stack.push(curr_value);
     break;
case Assignment:
     curr_value = expr_stack.pop();
     symbol_name = (char *)param_block;
     symbol_table.set_variable_value(symbol_name, curr_value);
     break;
case Display_value:
     curr_value = expr_stack.pop();
     cout << curr_value << endl;
     break:
default:
     //fatal error!
}
```

Listing 1: Engine for executing machine code: the C way.

```
class Operation
{
public:
      virtual void execute() = 0;
}

class Literal_number : public Operation
{
private:
      int value;
public:
      virtual void execute();
}

class Variable_value : public Operation
{
private:
      String name;
public:
      virtual void execute();
}

class Add_operation : public Operation
{
public:
      virtual void execute();
}

class Function_call : public Operation
{
private:
      String name;
      int num_passed_arguments;
public:
      virtual void execute();
}

class Assignment : public Operation
{
private:
      String target_variable_name;
public:
      virtual void execute();
}

class Display_value : public Operation
{
public:
      virtual void execute();
}
```

Listing 2: The class hierarchy of machine code operations.

```
class Machine_code
{
private:
      Operation **program_steps;
      int num_program_steps;
public:
      Machine_code();
      ~Machine_code();
      void run();
}
```

Listing 3: Class declaration for the machine code object.

```
void Literal_number::execute()
{
     expr_stack.push(atoi((char *)param_block);
}

void Variable_value::execute()
{
     curr_value = symbol_table.get_variable_value((char
*)param_block);
     expr_stack.push(curr_value);
}

void Add_operation::execute()
{
     curr_value = expr_stack.pop() + expr_stack.pop();
     expr_stack.push(curr_value);
}

void Function_call::execute()
{
     num_passed_arguments = (int)param_block;
     param_block += sizeof(int);
     symbol_name = (char *)param_block;
     //the execute_function method pops the number of arguments
specified by num_passed_arguments
     curr_value = symbol_table.execute_function(symbol_name,
num_arguments);
     expr_stack.push(curr_value);
}

void Asignment::execute()
{
     curr_value = expr_stack.pop();
     symbol_name = (char *)param_block;
     symbol_table.set_variable_value(symbol_name, curr_value);
}

void Display_value::execute()
{
     curr_value = expr_stack.pop();
     cout << curr_value << endl;
}
```

Listing 4: Execute methods for the action classes.

Sidebar: Object-Oriented Finite State Machines

One situation where many programmers would concede the necessity of using the *switch* statement in C++ code is the implementation of finite-state machines (FSMs). There are many uses of FSMs, but the most common involve parsing, and especially the lexical analysis phase of a parser, an important stage in the implementation of interpreted languages. The common implementation involves a loop which contains a *switch* statement which selects the actions for the current state, and ends when the final state is detected. The problem with this approach, as with all uses of *switch*, is that, it involves a monolithic language construct, in which all the statements lie in the same scope. Not only can this lead to long and unwieldly source modules, but the extremely close coupling between code in different *case* blocks makes modification and extension time-consuming and error-prone.

Luckily, object-oriented techniques can be applied to the implementation of FSM, bringing with them the same improvements as with other uses of *switch*. There are several ways that this can be done, and there have been discussions on *comp.lang.c++* as to the details. One possibility involves declaring an abstract object, *State* with a pure virtual function *transit* which embodies state transition. Each of the states in the FSM are declared as public child classes of *State*, and overload the *transit* method to evaluate the input and make the appropriate transition for the state, creating and returning a pointer to a new *State* object. The FSM operation is then implemented in a loop where the initial state is created, and its *transit* method called, returning an object embodying the succeeding state, whose *transit* method is called, and so on, until the final state is detected. A code skeleton follows.

```
class State
{
public:
        virtual State *transit() = 0;
};
```

```
        class Initial_state : public State{};     //defines own transit method according to FSM
rules
        class State0 : public State{};
        class State1 : public State{};
        //...
        class Final_state : public State{};       //transit method returns NULL

        State *curr_state = new Initial_state;
        State *new_state;
        while (curr_state)
        {
                new_state = curr_state->transit();
                delete curr_state;
                curr_state = new_state;
        }
```