The Russian Peasant Algorithm

Matt Austern

The most striking attribute of the C++ Standard Library is that so much of it is a generic library. Most obviously, every nontrivial library component is a template. Even more unusually, parts of the library clauses in the C++ standard describe things that aren't really C++ code: the character traits requirements in chapter 21, for example, and the input iterator requirements in chapter 24. These tables seem slightly intangible. They describe a pure interface; they don't describe any particular class, nor do they mandate any sort of inheritance relationship.

These requirement tables are the most visible manifestation of *generic programming*, a technique for modularization, interoperability, and reuse of algorithms and data structures. Not all parts of the C++ library use this methodology equally. The most fully generic part of the C++ library is the part dealing with containers, iterators, and algorithms that operate on linear sequences—the Standard Template Library. Just from looking at the C++ library, you might get the impression that generic programming only applies to that one narrow problem area. That impression is wrong. The STL was the first important example of generic programming, but the underlying ideas are general.

The Russian Peasant Algorithm

As is so often the case, the idea begins with algorithms. Generic programming has been defined as "Lifting of a concrete algorithm to as general a level as possible without losing efficiency; *i.e.*, the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm."

To arrive at the abstract we can begin with the concrete. Consider the problem of raising a number to a power. The fundamental definition of exponentiation, for positive integral exponents, is

$x^n = x \times \cdots \times x$ (n times).

We could turn that definition directly into code, but that would be wasteful: an algorithm based directly on that definition would use nine multiplications to compute x^{10}. Instead, since

$x^{10} = ((x^2)^2)^2 \times x^2$,

we can multiply x by itself to obtain x^2, multiply x^2 by itself to obtain x^4 and then x^4 by itself to obtain x^8, and finally multiply x^8 by x^2 to obtain the result.

This simple observation is the basis of the "Russian peasant algorithm," one of the earliest algorithms to have been discovered. (Donald Knuth, in *The Art of Computer Programming*, reports that a variation of the algorithm is known to have been used four thousand years ago.) In pseucocode, using the notation $\lfloor u \rfloor$ to represent the greatest integer less than or equal to u, the algorithm is:

Require: $n \geq 1$
Ensure: $P = x^n$
while n is even **do**
  $x \leftarrow x \times x$

$n \leftarrow \lfloor n/2 \rfloor$
end while
$P \leftarrow x$
$n \leftarrow \lfloor n/2 \rfloor$
while $n > 0$ **do**
$x \leftarrow x \times x$
if n is odd **then**
$P \leftarrow P \times x$
end if
$n \leftarrow \lfloor n/2 \rfloor$
end while

In C, this becomes the function `exponentiate1`. As promised, `exponentiate1` computes x^4 in two multiplications, and x^{10} in four multiplications.

```
double exponentiate1(double x, int n)
{
  double P;

  while ((n & 1) == 0) {
    x = x * x;
    n >>= 1;
  }
  P = x;
  n >>= 1;
  while (n > 0) {
    x = x * x;
    if ((n & 1) != 0)
      P = P * x;
    n >>= 1;
  }

  return P;
}
```

A generic algorithm

Is `exponentiate1` really an implementation of the Russian peasant algorithm? There's an important differences between C and pseudocode: C is a programming language where every variable must be declared to have some specific data type, but in the pseudocode we began calculating with *x* and *n* before ever saying just what they were.

The implementation and the pseudocode are both flawed. The implementation is flawed, because it is too specific. It can compute x^n if *x* is a `double`, but not if *x* is an `int` or a `long` or a `float`. A function that is so limited is not suitable for reuse, or for incorporation into a library.

The flaw in the pseudocode is that we never said what *x* and *n* were. The algorithm doesn't require that *x* is an double-precision floating point number, but it does have to make some

kind of assumptions about x and we never said what those assumptions were. (Does the algorithm make any sense if x isn't a number? Can you exponentiate a machine address, or a string, or a file descriptor?)

The assumptions about x and n are implicit in the operations that the Russian peasant algorithm performs, just as it is implicit in those operations that the exponent must satisfy the precondition $n \geq 1$. But, while that precondition is stated explicitly in the very first line of the algorithm, the assumptions about x are not. One further step is necessary before the algorithm has been fully described (a step that is usually omitted in books about algorithms): it must be analyzed so that the assumptions about x and n can be made precise.

The requirements on n are obvious: n must belong to some numeric type, and must support the ordinary numeric operations. We perform the following operations on n:

- **Assignment**. We can assign a new value to n, and pass n as an input variable.
- **Comparison**. We can test whether n is greater or less than another value..
- **Integer division**. We can halve n, discarding the fractional part. Halving must satisfy all of the ordinary rules of integer division—in particular, that $\lfloor n/2 \rfloor$ is non-negative and less than n.
- **Parity test**. We can test whether n is even or odd.

In other words, n is an integer.

The requirements on x are more interesting. We perform the following operations:

- **Assignment**. We can assign a new value to x. We can also pass x as an input variable, create a new variable P initialized to x, and return x as a result.
- **Multiplication**. We can form a product from two values, and the product can be assigned back to a variable of the same type.

The first of these requirements, assignment (which appears here in four different forms), is unremarkable: it seems to be ubiquitous. The second, multiplication, deserves more attention. What sorts of things can be multiplied, and what exactly do we mean by multiplication?

A different way to ask this question is to ask why the Russian peasant algorithm works in the first place. The reason we can compute x^4 using two multiplications, instead of three, is that we can square x^2; we don't have to multiply by x again and again. We need the property that
$x \times (x \times (x \times x)) = (x \times x) \times (x \times x)$.
But this is just the associative law:
$a \times (b \times c) = (a \times b) \times c$.
The Russian peasant algorithm works because multiplication is associative.

No further conditions are needed. The Russian peasant algorithm never assumes that multiplication is commutative: you can use it, for example, for square matrices, and matrix multiplication is not commutative. Neither does the Russian peasant algorithm require multiplication to have an inverse operation. Obviously not: we can use it for integers, and an integer doesn't have a multiplicative inverse. Finally, nothing in the Russian peasant algorithm requires the existence of an identity element: the operation "$x \leftarrow 1$" appears nowhere.

The Russian peasant algorithm can be used for any type that has an associative binary operation; it doesn't matter whether or not that associative binary operation is something we would ordinarily call multiplication. It is easy to see, for example, that while string concatenation is not commutative (`"abc"` appended to `"def"` is not the same as `"def"` appended to `"abc"`), it is associative. The Russian peasant algorithm is a perfectly reasonable way to construct a string from *n* copies of another string.

Mathematicians have a name for a set with an associative binary operation: it is called a *semigroup*. Finally, then, we have answered the question about *x*'s and *n*'s types: *n* must be an integer, and *x* must be an element of a semigroup.

(Technically, this is just slightly too strong. Multiplication of real numbers is associative, but a `double` is not a real number. It is a floating-point number, and floating-point arithmetic on computers does not quite obey the usual arithmetic axioms. Floating-point multiplication is only approximately associative; you can use the Russian peasant algorithm for floating-point numbers if that approximation is good enough for your purposes.)

Generic algorithms in C?

The standard C library includes two different functions that are at least partly generic: `qsort`, which sorts an array of values, and `bsearch`, which finds a value in an array that has already been sorted. Can they serve as a model for a generic version of `exponentiate1`?

The function qsort has the prototype

```
qsort(void* p, size_t N, size_t sz,
      int (*cmp)(const void*, const void*));.
```

The first argument, `p`, is a pointer to an array of `N` elements. The elements in the array may be of any type, which is why `p` is declared to be of type `void*` rather than (say) of type `int*`. The last two arguments provide some information about the elements contained in the array `p`: `sz` is the size of an element and `cmp` is a pointer to a function that compares two elements.

These specific details don't apply to `exponentiate1`, but the general approach does: refer to arguments through `void` pointers, and perform operations on them through user-supplied callback functions. In `exponentiate2` we use this idea for a our second attempt at an implementation of the Russian peasant algorithm.

```
void exponentiate2(void* x, int n, void* result,
                   void (*multiply)(void* a, void* b,
                                    void* product),
                   void (*assign)(void* from, void* to))
{
  while ((n & 1) == 0) {
    multiply(x, x, x);
    n >>= 1;
  }
```

```
    assign(x, result);
    n >>= 1;

    while (n > 0) {
      multiply(x, x, x);
      if ((n & 1) != 0)
        multiply(result, x, result);
      n >>= 1;
    }
  }
```

What's wrong with `exponentiate2`? Well, for one thing it's much harder to understand than `exponentiate1`. You probably had to stare at it for a while to figure out what the arguments `multiply` and `assign` were. It's also much harder to use; instead of just writing

`exponentiate1(x, 10)`,

you first have to define two helper functions

```
  void multiply_dbl(void* a, void* b, void* product) {
    *(double*) product = *(double*) a * *(double*) b;
  }

  void assign_dbl(void* from, void* to) {
    *(double*) to = *(double*) from;
  }
```

and then call exponentiate2 by writing

```
  exponentiate2(&x, 10, &result, multiply_dbl, assign_dbl);.
```

This interface is cumbersome and error-prone: it has several traps that are very easy to fall into. (I fell into one of them the first time I tested `exponentiate2`). Additionally, since it decorates every multiplication with a function call, a cast, and an indirection, `exponentiate2` is dreadfully slow.

Generic algorithms in C++

There is no good solution in C: there is no way to implement the Russian peasant algorithm in C that is as flexible as `exponentiate2` and also as simple and efficient as `exponentiate1`.

The Russian peasant algorithm is a way to exponentiate an element of any semigroup, and the fact that *x* is a semigroup element is a crucial part of that algorithm. Writing versions of the Russian peasant algorithm again and again for different data types isn't just wasteful; it also hides an essential abstraction. That aspect of the algorithm, too, cannot be expressed in C.

There is a solution in C++: templates. We can implement the Russian peasant algorithm as `exponentiate3`, a function template.

```
  template <class SemigroupElement, class Integer>
  SemigroupElement exponentiate3(SemigroupElement x, Integer n)
```

```
{
  while ((n & 1) == 0) {
    x = x * x;
    n >>= 1;
  }
  SemigroupElement P = x;
  n >>= 1;
  while (n > 0) {
    x = x * x;
    if ((n & 1) != 0)
      P = P * x;
    n >>= 1;
  }

  return P;
}
```

Templates were not always part of C++; they were added in 1991 precisely to solve this sort of problem. C++ supports *class templates* (parameterized types) and *function templates* (parameterized functions). The function `exponentiate3` is an example of the latter. You can use `exponentiate3` in exactly the same way as you use `exponentiate1`; you can, for example, invoke it by writing

```
exponentiate3(2.71828, 10).
```

The name `SemigroupElement` is a formal generic *template parameter*. Within the definition of `exponentiate3` it is treated as the name of a type; we use it as the type of a function argument, a temporary variable, and the return value. Just as the variable *x* doesn't refer to one specific value but to whatever value was passed as an argument to `exponentiate3`, so `SemigroupElement` refers to whatever type that argument has. The three expressions of type `SemigroupElement` are all the same type, whatever that type might be.

A function template like `exponentiate3` isn't a function in the sense that `exponentiate1` is a function; it's a schema that describes an infinite family of functions parameterized by the formal template parameters `SemigroupElement` and `Integer`. Each function in that family is called a *specialization*, or an *instance* of exponentiate3. In the expression

```
exponentiate3(2.71828, 10)
```

we are invoking the instance of `exponentiate3` for which `SemigroupElement` is `double` and `Integer` is `int`. The template argument `double` corresponds to the template parameter `SemigroupElement`.

All of this works because of specific C++ language features:

- **Templates**. We have written `exponentiate3` as a parameterized family of functions all of which represent the same algorithm, but that use different types.
- **Overloading**. Whatever `SemigroupElement` is, we can use the `*` operator to multiply two values of that type. The `*` operator means something different depending on whether its operands are `int,` or `double`, or `Complex`. The same name refers to

different things. In this case it's the name of an operator, but we could just as easily have written `exponentiate3` in terms of functions with names like `square and halve`. Writing generic algorithms requires the ability to overload functions; operator overloading is merely a special case of that general facility.

- **User-defined types**. Operator overloading is just one example of an important principle: the ability to treat built-in types and user-defined types on an equal footing. In `exponentiate3` we declare a local variable `P` of type `SemigroupElement`. We didn't have to do anything special; we didn't have to allocate a new object from the free store, for example, or call a special initialization function before using the variable, or call a special cleanup function afterwards. C++ allows true local variables of user-defined types, and it performs automatic initialization and cleanup through constructors and destructors, so using a variable of type `SemigroupElement` is just like using a variable of type `int`.

Constraints on template parameters

It's clear that `exponentiate3` is more general than `exponentiate1`. Two questions about it remain, though.

- The Russian peasant algorithm describes a procedure for exponentiating an element of any semigroup. To what extent does `exponentiate3` reflect that generality?
- What happens if you try to call `exponentiate3` with a clearly inappropriate type—that is, what happens if you try to instantiate it with a template argument that couldn't possibly be an element of a semigroup? What happens, for example, if `exponentiate3`'s first argument is a pointer?

Both of these questions straddle the boundary between an abstract algorithm (the Russian peasant algorithm) and its implementation in a specific language (`exponentiate3`). Not surprisingly, they are related.

The second question is easier: if `exponentiate3`'s first argument is a pointer, you will get an error message complaining about invalid arguments to `operator*`. The implementation of `exponentiate3` uses `operator*` for multiplication, and there is no binary `operator*` for pointers. The expressions `x * x` and `P * x` are meaningless when `x` and `P` are pointers.

This is a simple example, but it teaches us two important lessons. First, there are constraints on the template parameters `SemigroupElement` and `Integer`: the function template `exponentiate3` can only be instantiated with template arguments for which all of the expressions in `exponentiate3` make sense. Second, these constraints are purely syntactic. It's not sufficient for `X` to be in some sense a semigroup element; not only must `X` have an associative binary operation, but that binary operation must be called `operator*`.

(Strictly speaking, the constraints are actually on the template *arguments* used to instantiate `exponentiate3`: the issue is whether a specific type `X` can be substituted for the formal template parameter `SemigroupElement`. To insist on that distinction, though, would be pedantry. Writing the constraints in terms of the template parameters is clear and

unambiguous, and it lets us avoid inventing new names for the template arguments.)

The constraints on `exponentiate3`'s template arguments, then, aren't quite the same as the list of requirements in the Russian peasant algorithm itself. That list was constructed from a formal algorithmic description. We need a different way to express the constraints that apply to the actual implementation of `exponentiate3`, and those constraints must be based on the ways in which the template parameters `SemigroupElement` and `Integer` are used in expressions.

The template parameter `Integer` must be an integer type; no more analysis is necessary. The template parameter `SemigroupElement` must support the expressions that are used in `exponentiate3`'s implementation. That is, it must satisfy the following requirements:

- `SemigroupElement` has a copy constructor: it must be possible to copy values of type `SemigroupElement`. (This is necessary because `exponentiate3`'s argument of type `SemigroupElement` is passed by value.) If `x` is a value of type `SemigroupElement`, then the expression
  ```
  SemigroupElement(x)
  ```
  and the statement
  ```
  SemigroupElement x1(x);
  ```
  must be valid.
- The multiplication operation is defined for values of type `SemigroupElement`. If `x1` and `x2` are values of type `SemigroupElement`, then the expression
  ```
  x1 * x2
  ```
  must be valid and its return value must be of type `SemigroupElement`.
- `SemigroupElement` has an assignment operator. If `x` is a variable of type `SemigroupElement`, and `x1` is a value of type `SemigroupElement`, then the expression
  ```
  x1 = x
  ```
  must be valid.

These are requirements on expressions, rather than on specific functions. We rely on the fact that `SemigroupElement` supports multiplication, but we don't rely on an exact signature. Multiplication might have any of the signatures

```
SemigroupElement SemigroupElement::operator*(SemigroupElement)
SemigroupElement operator*(SemigroupElement, SemigroupElement)
SemigroupElement operator*(const SemigroupElement&,
                           const SemigroupElement&)
```

or any of a number of other possibilities. If `SemigroupElement` is a fundamental type like `int` or `double` then multiplication is a built-in operation, not a function at all. That's unimportant. So long as the expression

```
x1 * x2
```

is valid, it doesn't matter whether multiplication is a global function, a member function, or something else.

These syntactic constraints are necessary conditions, but they aren't sufficient; they're enough

to ensure that the program compiles, but not enough to ensure that it computes the right answer. We can't expect `exponentiate3` to work properly if "assignment" doesn't assign and "multiplication" doesn't multiply. Each of the expressions on that list must have the appropriate semantics:

- The copy created by `SemigroupElement`'s copy constructor is the same as the original. If `x` is a value of type `SemigroupElement`, then `SemigroupElement(x)` is the same as `x`. Similarly, assignment results in a copy of the original. If `x` and `x1` are variables of type `SemigroupElement`, then, after executing `x1 = x`, we may assume that `x` and `x1` are equal.
- Multiplication on `SemigroupElement` is associative. If `x`, `y`, and `z` are values of type `SemigroupElement`, then `x * (y * z)` is the same as `(x * y) * z`.

Summary

Let's look back at what we've done. First, we wrote a generic implementation of the Russian peasant algorithm; this implementation operates on the arbitrary type `SemigroupElement`. Second, we analyzed that implementation to discover what requirements that type had to satisfy in order for the algorithm to make sense. Third, we codified those requirements as a list of syntactic and semantic expressions involving `SemigroupElement`. This is much like the requirements tables in the C++ standard.

This list of constraints shows that we were actually programming in terms of a specific interface: an interface consisting of the operations that `SemigroupElement` must support. Any concrete type will undoubtedly support many more operations than the ones in this minimal list: you can do many more things with a `double`, for example, than just copy it or multiply it. As far as exponentiation is concerned, however, those extra operations are irrelevant.

By listing these constraints we abstracted out a precise and limited set of properties. Our interface describes the commonality between `int`, `double`, `SquareMatrix`, `Quaternion`, and many other types—types that are completely unrelated by inheritance.

This is similar to what Bertrand Meyer calls "programming by contract": `exponentiate3` imposes a specific set of requirements, and it guarantees that it will perform a certain task if and only if those requirements are satisfied. The difference is that, in its traditional sense, programming by contract deals with preconditions and postconditions of a function: it refers to the function's run-time behavior. We have arrived a different sort of contract: one that lists the conditions under which it is possible to combine different software components together.