A RISC class (extended abstract)

Andrew Koenig

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

The state of the art

Over the past several years, C++ class authors have spent a lot of effort on *container classes*: classes whose objects contain objects of some other class. During that time, a canonical style has evolved: a *container* actually holds objects while an object of an auxiliary class called an *iterator* marks a place in the container.

This strategy is similar to how arrays work in traditional languages: an array holds values and an auxiliary value, usually an integer or a pointer, marks a place in the array. Thus when we write a C program fragment like:

```
int a[N];
int i;

for (i = 0; i < N; i++)
        a[i] = i;
```

we are using `a` as a container and `i` as an iterator.

Because the iterator idea fits well with data structures in traditional programming languages, C++ programmers find it easy to adopt. However, iterators are not quite as simple in practice as in theory. For example, the possibility that containers might be `const` implies that two kinds of iterators are actually necessary for each kind of container: one that can only read the container and one that can read and write it. This is analogous to the two kinds of pointers that might point to an object of type `T`, namely `T*` and `const T*`.

In addition to requiring three classes for each kind of container, the traditional approach puts lots of features into the container and iterators. Aside from the usual social pressure, this is sometimes necessary to ensure proper interaction with iterators.

For example, the list class from the USL Standard Components library supports the following operations:

- Create a list from 0−4 elements.
- Copy and assign a list.
- Determine the length of a list.
- Concatenate two lists.
- Determine if two lists are equal.
- Add, access, or delete an element at either end.
- Access the *n*th element.
- Sort a list.
- Print an entire list, recursively using the output operation defined on the list elements

In addition, list iterators support the following operations:

- Create, copy, assign, and compare iterators.
- Find the list associated with a given iterator.
- Determine if the iterator is at the end of its list.

- Search for an element with a given value.
- Advance the iterator to refer to the next element in the list (in either direction).
- Access the element before or after the iterator (iterators point ''between'' elements).

The operations add up: the USL list class has 70 members and one friend function, divided as follows:

- `List`: 27 members, one friend.
- `Const_listiter`: 28 members.
- `Listiter`: 15 members.

The implementation is more than 900 lines of source code.

A radical old idea

All this functionality is very nice, but it sometimes leaves one wishing for less. If we can have lightweight processes and RISC (Reduced Instruction Set Computer) chips, why not a RISC (Reduced Instruction Set Container) class? For inspiration, I went back more than a third of a century and looked at Lisp.

Lisp has grown greatly over the years, but its original notion of a list was defined in terms of just five operations:

- `nil`: a list with no elements.
- `cons(a,b)`: a list whose first component is `a` and whose subsequent components are those of the list `b`.
- `car(s)`: the first component of `s`; `s` must be a list with at least one component.
- `cdr(s)`: all the components of `s` except the first; `s` must be a list with at least one component.
- `null(s)`: true if `s` has no components, false otherwise; `s` must be a list.

There is a *very* large body of theory and practice that shows that these operations are useful. Moreover, it's easy enough to implement in C++ if we add the restriction that all the elements of a list must be the same type. Using templates, we can then define a class `Seq<T>` that represents a list of objects of type `T`. It's called `Seq` rather than `List` to avoid confusing it with any of the other `List` classes out there.

Suppose we start by implementing just the five primitive Lisp operations. How might those look in C++? Assume that `t` has type `T` and `s` has type `Seq<T>`. Then the following C++ operations suggest themselves:

- `Seq<T>()`: a sequence with no elements.
- `cons(t,s)`: a sequence with first element `t` and subsequent elements those of `s`.
- `s.hd()`: the first element of `s`; `s` must have at least one element.
- `s.tl()`: a sequence containing all but the first element of `s`; `s` must have at least one element.
- `(const void*)s`: nonzero if `s` has no elements, otherwise zero.

The names `hd` and `tl` are there because they are easier to explain to non-Lispers than `car` and `cdr`. The conversion to `const void*` is intended to be used in `if` and `while` statements.

Well, maybe a few extras...

This class is pretty easy to implement. Moreover, it is useful as it stands. However, it is hard to resist the temptation to add a few additional operations to make it easier to use. Some of this ease of use comes from differences in typical style between Lisp and C++ programs.

For example, suppose we want to compute the number of elements in a sequence `s`. A Lisp programmer might write it this way:

```
template<class T>
int length(const Seq<T>& s)
{
      if (s)
            return (1+ (length (s.tl())));
      return 0;
}
```

but a C++ programmer would be prone to do it this way:

```
template<class T>
int length(Seq<T> s)
{
      int n = 0;

      while (s) {
            s = s.tl();
            n++;
      }
      return n;
}
```

This combination of testing for null and extracting an element happens so often that it is useful to introduce an operation called `next` to do that. Using it, the inner loop looks like this:

```
while (s.next())
      n++;
```

It also turns out to be useful to overload `next` to put the element it extracted into a variable passed by reference, so that

```
int n, sum = 0;
while (intseq.next(n))
      sum += n;
```

adds the elements of a `Seq<int>` named `intseq` and puts the result in `sum`.

Another common operation is to put a new value at the head of a list. Using the primitives, that looks like this:

```
s = cons(s, x);
```

and is again sufficiently common to warrant a built-in operation:

```
s.insert(x);
```

There are efficiency reasons for these extra built-ins, too; we will discuss those later. Of course, it is a hard judgment call where to stop adding things.

Example of use

The best way to give a feel for how this class works is with a complete example. We will begin with a function to merge two sequences, assumed to be in ascending order:

```
template<class T>
Seq<T> merge(const Seq<T>& x, const Seq<T>& y)
{
      if (!x) return y;
      if (!y) return x;

      T xh = x.hd();
      T yh = y.hd();

      if (xh < yh)
            return cons(xh, merge(x.tl(), y));
      return cons(yh, merge(x, y.tl()));
}
```

Programmers who aren't used to recursion might prefer doing it this way:

```
template<class T>
Seq<T> merge(Seq<T> x, Seq<T> y)
{
      Seq<T> r;

      while (x && y) {
            if (x.hd() < y.hd()) {
                  r.insert(x.hd());
                  x.next();
            } else {
                  r.insert(y.hd());
                  y.next();
            }
      }
      while (x) {
            r.insert(x.hd());
            x.next();
      }

      while (y) {
            r.insert(y.hd());
            y.next();
      }

      r.flip();
      return r;
}
```

Aside from the primitives and `insert` and `next`, this second implementation uses one other built-in: `s.flip()` reverses the elements of `s` in place.

Which of these two versions is easier to understand probably depends on where you were brought up.

Next we'll look at a function that splits a sequence into two piles, much the way one would deal a deck of cards. We will assume either that the piles are empty to start, or that our caller doesn't mind our piling the elements on top of the ones already there. The elements are reversed in the process, just as they are when dealing cards:

```
template<class T>
void split(Seq<T> x, Seq<T>& y, Seq<T>& z)
{
	while (x) {
		y.insert(x.hd());
		x.next();
		if (x) {
			z.insert(x.hd());
			x.next();
		}
	}
}
```

The only thing the least bit tricky here is that

```
x.next();
if (x) { // ...
```

cannot be collapsed into

```
if (x.next()) { // ...
```

because we want to see if `x` is non-empty only *after* removing the first element.

Finally, we can use these two functions to make a simple sort function:

```
template<class T>
Seq<T> sort(const Seq<T>& x)
{
	if (!x || !x.tl())
		return x;

	Seq<T> p, q;

	split(x, p, q);
	return merge(sort(p), sort(q));
}
```

Simple as it is, this merge sort has *O(log n)* performance, even in the worst case. Perhaps a measure of the utility of the `Seq` class is how rare it is to find a sort that can be explained to novices and also has good asymptotic performance.

Experience

I implemented this class using a straightforward use-counted memory allocation technique. That means that copying a `Seq<T>` object does not copy its components, so it is not particularly expensive to pass `Seq<T>` objects directly to functions instead of references. This implementation partly motivates some of the ''extra'' operations, because some optimizations are possible if the right use counts happen to be 1. For example, if the use counts of all the elements of a sequence are 1, it is possible to concatenate another sequence onto the end of it in place. This can be substantially faster than the obvious implementation:

```
template<class T> Seq<T> operator+(const Seq<T>& r, const Seq<T>& s)
{
        if (!s)
                return r;
        if (!r)
                return s;
        return cons(r.hd(), r.tl()+s);
}
```

A similar optimization is possible for the `flip` operation.

Even with the extra operations and optimizations, the class has only fourteen members and four friends. The complete implementation is less than 200 lines of source code; the manual page fits on one (double sided) sheet of paper. For the operations it supports, it runs about as fast as the USL list class.

Future directions

Right now, this class is used by the author and a few of his colleagues. That experience shows that it is indeed useful, but a few more operations would make it even more useful. One is a `length` function. Of course it is possible to find the length of a sequence, but it is sometimes important to be able to do so without looking through the whole thing. For instance, one may wish to copy a sequence into an array, in which case it is important to know how much memory to allocate before beginning the copy. The price of this ability is storing the length in every sequence. However, once that is done, updating it is inexpensive.

Another useful operation is the ability to compare entire sequences, especially in a lexical sense. That makes it easy, for example, to sort a container full of sequences.

More generally, it would be interesting to apply the RISC approach to other data structures. It works well for sequences because it is possible to get away without modifying the elements of a sequence after it has been created. This restriction is harder for more general data structures, but the functional programming community has shown that it is possible. The interesting question is whether that style can be transplanted into C++ and, if so, whether people will find it worthwhile to use. This is an open question that deserves an answer.