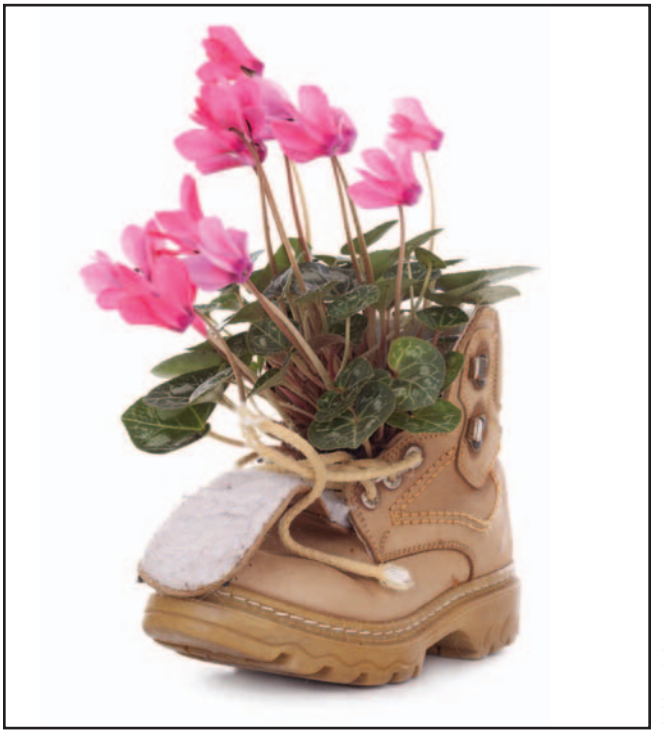
Designing Reusable Software

by Chuck Allison



ISTOCKPHOTO

For software to be reusable, it must be usable in a variety of contexts. An important attribute of reusability at the code level is *genericity*. Generic software can operate, unmodified, on a variety of object types. Dynamically typed languages provide genericity naturally. Consider the following Python function that determines the smallest of whatever it receives as a parameter:

```
def smallest(stuff):
    assert len(stuff) > 0
    theMin = stuff[0]
    for x in stuff[1:]:
        if x < theMin: theMin = x
    return theMin
```

You can call this function with any type of sequence:

```
>>> smallest([1,2,3])
1
>>> smallest("cba")
a
```

The only constraints on the parameter `stuff` are that it is an iterable entity and that its contained objects support the less-than operator. Hence, `smallest` is a generic function.

Statically typed languages usually don't have such flexibility out of the box, but that is changing. Modern languages such as Eiffel, C++, Java, C#, and D have added generic programming capability while retaining the safety and efficiency that comes with static type checking.

Consider an everyday programming task: applying a series of transformations to data. Data transformations usually involve calling a series of functions on individual data elements. While it is simple to write separate functions and apply them in sequence, it can be even easier to assemble existing functions into a reusable, composite function. In other words, we want support for *function composition*.

In the November 2007 Code Craft, I used the following Python function:

```
def compose(*funs):
    return lambda x: reduce(lambda z,f: f(z), reversed(funs), x)
```

In case you forgot, `reduce` applies its first argument, a binary function that I'll call the *applicator,* to the first list element along with the initial value. That result and the next function in the list become input arguments to a second invocation of the applicator, and so on, until the list of functions is exhausted and the accumulated result is obtained. If you prefer a procedural rendering of `compose`, see listing 1.

```
# Listing 1
def compose(*funs):
    def apply(x):
        result = x
        for f in reversed(funs):
            result = f(result)
        return result
    return apply
```

Listing 1

Both functions return a new, single-valued function representing the composition of the functions in the original list. So, if you have three like-typed functions, `f`, `g`, and `h`, you can compose them like this:

```
c = compose(f,g,h)
```

Every call to `c(x)` will compute `f(g(h(x)))`.

How can we accomplish this in C++? It won't be a one-liner, but there is a reasonable solution. But first, let's start with a non-generic version and then generalize it.

A Non-Generic Example

We'll assume the data type is `double` and will use C++'s `accumulate` algorithm, which works just like Python's `reduce`. Since we can't return functions in C++, we'll create a function object (a type implementing `operator()`), as shown in listing 2.

With a vector, `v`, of single-valued real functions, using their composition is straightforward:

```
// Listing 2: A non-generic C++ composer (for functions of doubles)
class Composer {
private:
    typedef double (*Fun)(double);
    vector<Fun> funs;
    static double apply(double sofar, double (*f)(double)) {
        return f(sofar);
    }
public:
    Composer(vector<Fun>& fs) : funs(fs) {
        reverse(funs.begin(), funs.end());
    }
    double operator()(double x) {
        return accumulate(funs.begin(), funs.end(), x, apply);
    }
};
```

Listing 2

```
// Listing 3: Generalizes the argument and sequence types
template<class T, class Iter>
class Composer {
private:
    typedef std::reverse_iterator<Iter> RevIter;
    RevIter beg, end;
    static T apply(T sofar, T (*f)(T)) {
        return f(sofar);
    }
public:
    Composer(Iter b, Iter e) : beg(RevIter(e)), end(RevIter(b)) {}
    T operator()(T x) {
        return accumulate(beg, end, x, apply);
    }
};
```

Listing 3

ISO standards for programming languages are eligible for renewal every ten years. C++ was standardized in 1998, and work began on a new version in 2003. The new version is expected to be complete before 2010, hence the "C++0x" moniker. C++0x will contain many enhancements and simplifications, including template constraints (a.k.a. "concepts"), lambda functions, and a regular expression library. Work on a concurrency model is also in progress.

```
Composer comp(v);
cout << comp(x);
```

This class has two obvious deficiencies:

1. Only functions of real numbers can be composed.
2. Users must employ `std::vector` as the data structure.

A Generic Solution

The solution to both flaws is to use templates, as shown in listing 3.

Sequences in C++ are delimited by iterators, which can be pointers or objects of classes that implement pointer operations (`operator*`, `operator->`, `operator++`, etc.). Therefore, this version of `Composer` has two template parameters—one for the data type `(T)` and one for the iterator type `(Iter)`. Since the functions need to be applied in the reverse of the order in which they appear, we use `std::reverse_iterator` to synthesize a reverse iterator type from `Iter`, which the `Composer` constructor initializes appropriately. The caller must provide

```
typedef string (*Fun)(string);
Fun funs[] = {f,g,h};
Composer<string, Fun*> comp(funs, funs+3);
cout << comp(x) << endl;
```

Listing 4

the template arguments and iterators. The sample code in listing 4 uses an array to hold pointers to the functions `f`, `g`, and `h`. The iterator type is therefore a *pointer* to such pointers-to-functions, as shown in listing 4.

Is more generalization possible? Yes, if we use a forthcoming feature of C++0x (the imminent update of standard C++), namely, `function`. The `function` class template wraps anything that is callable as a function. This way we can use function objects as well as function pointers, and even mix the two in the same sequence. This feature is available today as Boost.function (see the StickyNotes for a link). We also will use other template features of the current version of C++.

All we will require of users is to declare their sequences to hold compatible instances of `function`. The new implementa-

```
// Listing 5: Generalizes the callable type
template<class Iter>
class Composer {
private:
    // Deduce function and return/argument types
    typedef typename iterator_traits<Iter>::value_type Fun;
    typedef typename Fun::result_type T;

    // Declare reverse iterators (see constructor for use)
    typedef reverse_iterator<Iter> RevIter;
    RevIter beg, end;

    // The function called by accumulate
    static T apply(T sofar, Fun f) {
        return f(sofar);
    }
public:
    Composer(Iter b, Iter e) : beg(RevIter(e)), end(RevIter(b)) {}
    T operator()(T x) {
        return accumulate(beg, end, x, apply);  // Function Applicator
    }
};
```

Listing 5

```
// Listing 6
typedef function<double(double)> Fun;
std::list<Fun> funs;
funs.push_back(bind2nd(divides<double>(), 3.0));
funs.push_back(&f);    // A plain double (*)(double) function
funs.push_back(bind2nd(plus<double>(), 1.0));
transform(nums, nums+4, ostream_iterator<double>(cout," "),
          compose(funs.begin(),funs.end()));
```

Listing 6

tion of `Composer` also uses `iterator_traits`, a C++ feature that deduces the type to which an iterator points, as shown in listing 5.

We are assuming that the type pointed to, `Fun`, is an instance of `function`, which happens to have a `result_type` member. We use this as the argument type since we expect the functions we compose to have the same argument and return type.

To make things convenient for the user, we'll create a function template that deduces the iterator type from its arguments:

```
template<class Iter>
Composer<Iter> compose(Iter b, Iter e) {
    return Composer<Iter>(b,e);
}
```

This allows users to embed calls to `compose` in other contexts, as you can see in the sample code in listing 6, which composes two function objects along with a plain function pointer.

The first line defines the type `Fun` as `function<double(double)>`, which matches anything that is callable with a single `double` argument and returns a `double`. The first function, which we create on the fly, divides its argument by three. The `divides` function object comes with the standard C++ library, as does the `bind2nd` function-object adaptor, which converts `divides` into a single-arg function by fixing its second argument as the value 3. (See the September 2007 Code Craft article for more on standard function objects and `bind2nd`).

The call to `transform` processes an array of four doubles, `nums` (declaration not shown), applies the composition of the three given functions, and prints the result to standard output—all in one statement.

Summary

Let's review what we have accomplished—we have created a reusable class for function composition, which can come in handy in data processing, and we have gained some experience in designing generic software. The final version of `Composer` is generic in that it can compose an arbitrary number of unary, single-valued functions or function objects of compatible types, and those functions can reside in an array or in any standard sequence container. `Composer` itself has reused the standard generic types and functions: `accumulate`, `reverse_iterator`, `iterator_traits`, and `function`. **{end}**

Chuck Allison developed software for twenty years before becoming a professor of computer science at Utah Valley State College. He was senior editor of the C/C++ Users Journal *and is founding editor of* The C++ Source. *He is also the author of two C++ books and gives onsite training in C++, Python, and Design Patterns. Chuck is a technical editor for* Better Software *magazine.*

Sticky Notes

For more on the following topic go to www.StickyMinds.com/bettersoftware.

- Boost.function

Is defining for reuse on your list of priorities? Do you routinely use generics?

Follow the link on the StickyMinds.com homepage to join the conversation.