Make Reuse a Reality with STL Algorithms

by Chuck Allison

GETTY IMAGES

If you're like me, you like good code—like to read it and, especially, like to write it. So, what is good code? Code that has no redundancy? Code that has small functions? If time and management allow, we all should refactor our code so it has the nice properties experts tell us about.

But for me, the best code of all is code that I don't have to write.

Reuse has been a software Holy Grail for some time, and it finally seems to be paying off. Python, for example, has hundreds of packages in its standard library, and Java comes with thousands of useful classes. Using libraries should be automatic for programmers.

In this article I discuss what I believe is the most underused library in the C++ world—one that Java developers could take advantage of, too.

Generic Algorithms

Most of us are quick to use prepackaged containers such as vectors, lists, and maps in everyday programming. Where I see considerably less reuse is with algorithms. Algorithms—only the very essence of computer science! Go figure.

Quick quiz: How many common, general-purpose algorithms, like *quicksort* or *binary search*, can you think of? How many should a generic algorithms library provide?

Let me pose a similar question: How many distinct standard template library (STL) algorithms are there in the standard C++ library? If you're a C++ developer, you've probably used `sort`, `find`, `count`, and maybe even `for_each` and `transform`. How many are there altogether?

More than seventy!

The power of the STL algorithms derives from several factors:

1. They are templates, so they accommodate any type of argument with static type checking.
2. They work with built-in arrays as well as with sequence containers such as `vector`, `list`, `deque`, or any container of your own making that fills certain reasonable requirements.
3. They work with built-in types without any boxing/unboxing overhead.
4. Many of the algorithms accept functions or function-like objects as parameters, allowing you to customize their behavior.
5. Many useful function objects are available out of the box and can be combined with adaptors, also provided, so that you may discard many of the loops or functions that you might be writing today, saving you time and debugging headaches.

The result is an incredibly robust toolset. What follows are some examples that I hope you'll find interesting and useful.

To obtain the smallest element in an array of `N` elements, just call:

```
min_element(myarray, myarray+N)
```

All input sequences are delimited by a "pointer" to the beginning (`myarray`) and one past the end (`myarray+N`). Pointer is quoted in the previous sentence because you can also repeat the request for sequences other than arrays by using their iterators:

```
min_element(myvector.begin(), myvector.end())
```

The `begin()` and `end()` functions in the standard C++ containers return iterator objects that behave like pointers.

How many zeroes are there in a sequence? Try:

```
count(myvector.begin(), myvector.end(), 0)
```

How many positive elements are there? You could pass a greater-than-zero function to the `count_if` algorithm, as shown

```
bool gt_0(int x) {
    return x > 0;
}
count_if(myvector.begin(), myvector.end(), gt_0)
```

Listing 1

in listing 1.

All `count_if` needs is something that can be called as a single-argument function for every element in the given sequence.

Function Objects

You might find it easier, though, to use the built-in `greater` function object. A function object is just a class that implements the function call operator (`operator()`). The `greater` function object is a binary function, of course, so it needs to be adapted to work as a unary function with `count_if`. Here's how to make that happen:

```
count_if(myvector.begin(), myvector.end(),
         bind2nd(greater<int>(), 0))
```

The `bind2nd` adaptor function takes something that is callable as a binary function and wraps it in an object that is callable with one argument. Whenever the latter is called, it forwards the incoming argument as the first argument to the original binary function, and the original second argument it saved becomes the second argument. There also is a `bind1st` adaptor.

Do you need to fill a vector with a constant value, expanding the vector to `n` elements as you go? Do this:

```
vector<int> v;
fill_n(back_inserter(v), n, 0);
```

The `back_inserter` function wraps a sequence container in an iterator object that appends to its container every time the wrapper object itself is assigned a value. If you want random numbers, use `generate_n` with `rand`:

```
generate_n(back_inserter(v), n, rand)
```

```
ifstream f("myfile");
vector<string> v;
copy(istream_iterator<string>(f), istream_iterator<string>(),
     back_inserter(v));
```

Listing 2

As shown in listing 2, using `back_inserter` makes short work of populating a container from a file.

The `istream_iterator` adaptor function wraps a file stream in an object that behaves like a begin iterator—it reads from the file whenever it is accessed. The second occurrence of `istream_iterator` (without an argument) behaves like a past-the-end iterator.

Take a breath, and let's keep going.

If you do your own taxes, you've seen the phrase "If the amount is less than zero, enter zero" (much to our dismay). The following variation of the `replace` algorithm replaces all negative numbers in an array with zero:

```
replace_if(a, a+5, bind2nd(less<int>(), 0), 0)
```

To sum the numbers in a file, call:

```
accumulate(istream_iterator<double>(infile),
           istream_iterator<double>(), 0.0);
```

Since `accumulate` uses the "+" operator between its operands, you can use it to combine a sequence of strings into a single string:

```
accumulate(v.begin(), v.end(), string(""));
```

Another version of `accumulate` takes a fourth argument, which replaces the default addition operator with one of your choice. Here is an example that multiplies the numbers from a file together:

```
accumulate(istream_iterator<double>(infile),
           istream_iterator<double>(),
           1.0, multiplies<double>());
```

To compute the sum of the squares of each element requires a custom function, as shown in listing 3.

```
int sum_sofar(int b, int a) {
    return a*a + b;
}
accumulate(a, a+N, 0, sum_sofar) << endl;
```

Listing 3

This arrangement takes advantage of the fact that `accumulate` uses its running result as the first argument to each call of the binary function passed in the fourth position.

Suppose you want to read all words from a text file and create a new file with each string surrounded by quotes on a line by itself. Given a suitable quoting function, a single call to the `transform` algorithm does the job, as shown in listing 4.

This call to `transform` sends each string in the stream `infile` as a parameter to `quote` and writes the result to the output stream, `outfile`. The `ostream_iterator` adaptor wraps a stream in an object that writes the value it is assigned to the stream, followed by a separator character.

```
string quote(const string& s) {
    return '"' + s + '"';
}
transform(istream_iterator<string>(infile),istream_iterator<string>(),
          ostream_iterator<string>(outfile,"\n"), quote);
```

Listing 4

I like to play the game Text Twist on my PDA. You get the letters of a six-letter word in random order with the goal of discovering all words greater than or equal to length three. To continue from one screen to the next, you need to find at least

one of the possible six-letter words within the given time limit. Just for grins, I wrote a program to discover the n-letter words in an n-letter string. First I read the words from an open source dictionary into a set data structure as shown in listing 5.

```
set<string> words;
ifstream ifs("2of12inf.txt");
string word;
while (ifs >> word)
    words.insert(word);
```

Listing 5

Next I sort the six letters in preparation for examining all possible permutations:

```
sort(s.begin(), s.end());
    // s contains the six letters
```

Now it's just a matter of determining each possible arrangement of the letters and searching the dictionary for a match, as shown in listing 6.

```
do {
   if (words.find(s) != words.end())
      cout << s << endl;
} while (next_permutation(s.begin(), s.end()));
```

Listing 6

The `next_permutation` algorithm replaces its sequence with the next arrangement in lexicographical order, so it's important to sort the letters first or you'll miss possibilities. This isn't a one-line program, but it's readable and concise.

The implementation of these functions and algorithms is actually fairly straightforward, but you don't have to know how to implement them to use them. Once you get used to the idea of iterators, function objects, and adaptors, you can use the STL algorithms to write more declarative code. This means using fewer brain cycles and less code real estate spent in crafting loops and function definitions. Java programmers can find these algorithms in JGL from Recursion Software. **{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.*