THE UNTAPPED POWER OF GENERIC ALGORITHMS

Charles D. Allison
Utah Valley State College
Orem, UT 84058
801-863-6389
allisoch@uvsc.edu

Civilization advances by extending the number of important operations we can perform without thinking.

– Alfred North Whitehead

ABSTRACT

Generic Programming is one of the most exciting programming paradigms to emerge since object orientation. It allows programmers to combine the utility of dynamic data structures and associated operations as they exist in languages like Lisp, Python, and PHP, with the static type safety of modern object-oriented languages. The generic algorithms found in C++'s Standard Template Library give programmers an elegant tool to craft high-level, high-performance, declarative code. The advantage is enhanced code readability, faster time to project completion, and fewer opportunities to introduce bugs.

INTRODUCTION

Computer scientists are essentially abstractionists. Consider the progress of the last half century. The essentials of machine language were abstracted as assembly language, which in turn gave rise to high-level languages. Nowadays no one uses assembly language unless they have to. The power of object-oriented technology in recent years has taken programming to yet another level—creating one's own abstractions via classes is an everyday activity.

The Whitehead quote that prefaces this article captures the essence of abstraction; its usefulness lies in allowing us to focus on the principal, high-level concerns of a problem while ignoring details best left to another context. Most software developers are familiar with templates in C++ and generics in Java and C#, but few look beyond "containers of T" to the power of generic *algorithms*. This paper shows how the generic algorithms in the C++ Standard Template Library (STL) support a declarative style of programming enabling very high-level expressions, akin to using higher-order functions in functional languages.

ALGORITHMS + DATA STRUCTURES = PROGRAMS

The key design decision that made STL so revolutionary was to separate algorithms from the data structures they work with, allowing them to interact on demand via *iterators*, a sequence-traversal abstraction based on pointers. Because of this orthogonal design, programmers can create their own algorithms that work with STL containers, or their own containers that work with STL algorithms. Before STL, programmers either repeated code (i.e., they implemented separate data types for lists of integers, lists of strings, etc.) or they used collections that held typeless references to objects, like `void*` in C or `Object` in Java and Smalltalk. The former strategy leads to maintenance nightmares while the latter loses static type safety because of type erasure.

C++ changed all that by introducing templates. One can process a container or any type with generic algorithms because the latter are templates, and are instantiated on demand according to how they are used. Consider how elegantly the following code partitions a random sequence.

```
int a[5];
generate_n(a, 5, rand);    // Use std::rand to obtain pseudo-random numbers
int* p = partition(a, a+5, bind2nd(greater<int>(),n));
```

In two lines of code we have populated and partitioned a sequence of five integers such that all those greater than `n` are in the range `[a,p)` and the rest are in the range `[p,a+5)`. We could provide any suitable unary function as the third argument to `partition`, and could have used a sequence of any type that supports greater-than comparison. We could even have employed `generate_n` to create an arbitrarily large sequence to begin with, using an expandable vector instead of an array:

```
vector<int> v;
generate_n(back_inserter(v), N, rand);  // N is arbitrary
vector<int>::iterator p =
    partition(v.begin(),v.end(),bind2nd(greater<int>(),n));
```

The `back_inserter` *iterator adaptor* wraps an STL sequence in an iterator that appends a value to that sequence by calling its `push_back()` method upon assignment.

Note that `partition` is a higher-order function, since it expects a (unary) function as its third argument. The `greater` standard function object is a *binary* function template, so we use the `bind2nd` *function-object adaptor* to treat it is as a unary function. `bind2nd` is a function template that takes any binary function (or function object) and returns a unary function object that stores the original binary function and the value to be used as its second argument, leaving the first argument open. We could have written a suitable unary function, but it is easier and more flexible to use and adapt what STL offers. The standard function objects are listed in Figure 1.

Category	**Function Object Types**
Predicates	greater, less, greater_equal, less_equal, equal_to, not_equal_to
Arithmetic	plus, minus, multiplies, divides, modulus, negate

Figure 1 – Standard C++ Function Object Types

THE STUFF OF COMPUTER SCIENCE

Sedgewick posited that “algorithms are the ‘stuff’ of computer science.” [1] The STL provides over seventy generic algorithms distributed among five conceptual categories that can save programmers valuable time and result in more readable code. A listing by category follows in Figure 2.

Category	**Algorithms**	
Queries		for_each, find, find_if, find_first_of, adjacent_find, count, count_if, mismatch, equal, search, search_n, find_end
Mutators		transform, copy, copy_backward, swap, iter_swap, swap_ranges, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n, generate, generate_n, remove, remove_if, remove_copy, remove_copy_if, unique, reverse, reverse_copy, rotate, rotate_copy, random_shuffle
Ordering	*Sorting*	sort, stable_sort, partial_sort, partial_sort_copy, nth_element, merge, inplace_merge, partition, stable_partition
	Set Operations	includes, set_union, set_intersection, set_difference, set_symmetric_difference
	Heap Operations	push_heap, pop_heap, make_heap, sort_heap
	Searching	binary_search, lower_bound, upper_bound, equal_range
	Permutations	next_permutation, prev_permutation
	Min/Max	min, max, min_element, max_element, lexicographical_compare
Numeric		accumulate, inner_product, partial_sum, adjacent_difference
Special		uninitialized_copy, uninitialized_fill, uninitialized_fill_n

Figure 2 – The Generic Algorithms in C++

To further illustrate the declarative nature of using STL algorithms, suppose it is necessary to read all words from a text file and create a new file with each string surrounded by quotes on a line by itself. For example, an input file containing

```
how now brown cow
```

would result in the output file

```
"how"
"now"
"brown"
"cow"
```

Given a suitable quoting function, a single invocation of the `transform` algorithm does the job:

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

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 stream iterators `ostream_iterator` and `istream_iterator` wrap a stream in an iterator object that automatically writes or reads from its file whenever objects are assigned or fetched, respectively.

To sum the numbers in a file, one could call `accumulate`:

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

Another version of `accumulate` works like the “fold-left” function found in functional languages like ML. In ML, the following function, `sqsum`, which uses `foldl`, computes the sum of squares of a list of integers:

```
fun sqsum nums = foldl (fn (a,b) => a*a + b) 0 nums;
```

The same thing can be accomplished in C++ as follows:

```
int sum_sofar(int b, int a) {
    return a*a + b;
}

int main() {
    int a[] = {1,2,3,4};
    cout << accumulate(a, a+4, 0, sum_sofar) << endl;    // 30
}
```

This particular overload of `accumulate` takes a binary function as a fourth argument, which it applies to each sequence element in turn along with the accumulated result at that point in the execution (which is 0 initially). This same version of accumulate can also be used to obtain the product of the numbers in a file:

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

CURRICULUM CONSIDERATIONS

Students at the CS1 and CS2 levels are still learning the intricacies of algorithm development, and it is important that they have experience implementing them from first principles. It is recommended, therefore, to provide only limited exposure to STL algorithms until they have mastered the concepts outlined for CS1 and CS2. Near the end of CS1 at Utah Valley State College, we have students write simple data analysis programs, such as finding the mean, median, mode, etc., using STL algorithms. After CS2 students have implemented the typical searching and sorting algorithms, we encourage them to use the comparable STL algorithms in subsequent projects. We also have a junior-level advanced C++ programming elective course that explores STL in great depth. Students of this particular course repeatedly extol the algorithms and function-object facilities of STL as a satisfying milestone in their technical maturity, and as a favorite part of their undergraduate experience. We also use STL briefly in our senior-level analysis of programming languages course to show how functional programming can be done in C++. Students never fail to be pleasantly surprised at the “hidden” power of a language that they thought they already “knew” for so long.

SUMMARY

The generic style of programming supported by C++ allows students and programmers to concentrate on the logical steps leading to a solution instead of on low-level details such as loop control and list-traversal mechanisms. Thinking at a higher level better manages complexity and tends to lead to correct solutions more quickly than traditional procedural/OO practices. Neither type safety nor performance is compromised, since type information is not lost, and the generated code is equivalent to hand-written code performance-wise, minus the bugs. Java programmers can also take advantage of these algorithms using the JGL Toolkit. [2] Leveraging the power of generic algorithms is consistent with the goals of CS curricula pertaining to teaching students to use high-level abstractions in problem solving by computer.

REFERENCES

1. Sedgewick, R., Algorithms, Addison-Wesley, 1983, p. 4.
2. Recursion Software, Dallas, TX. See www.recursionsw.com.