The Awesome Power of Generic Programming

> *Civilization advances by extending the number of important operations we can perform without thinking.*
>
> *– Alfred North Whitehead*

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. This paper shows how the generic algorithms in the C++ Standard Template Library (STL) support a declarative style of programming at a level higher than most programmers are accustomed to.

Algorithms + Data Structures = Programs

The key design decision that made STL so revolutionary was to separate algorithms from the data structures they work with, and to allow these to interact on demand via *iterators*, a sequence-traversal abstraction based on pointers. Because of this separation, programmers can create their own algorithms that work with STL containers or their own containers with associated iterators that can 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 checking because of type erasure.

C++ changed all that by introducing templates. The following class template definition, for example, constitutes instructions for generating an abstraction named `Sequence` that is intended to behave like an expandable array:

```
template<class T>
class Sequence {
public:
    class iterator; // Defined elsewhere
    void append(T t);
    void insert(iterator it, T t);
    void erase(iterator it);
    int size() const;
    iterator begin();
    iterator end();
private:
    T* data;
    // Implementation details omitted.
};
```

`Sequence` is not a class, but a *template* for a generating a class at compile time. To create a `Sequence` object holding integers, for example, one would write

```
Sequence<int> mySeq;        // int version generated on demand
mySeq.append(4);            // OK
mySeq.append("hello");      // error!
```

Sequence<int> is a distinct type from Sequence<string>, so the compiler flags type errors like the one in the last line above.

There are no member functions to search, sort, or otherwise process a `Sequence`. These operations are left to stand-alone, *generic algorithms*. The following statement, for instance, obtains the position of the first occurrence of the smallest sequence value:

```
Sequence<int>::iterator pos = min_element(mySeq.begin(), mySeq.end());
```

The `min_element()` algorithm provided by the STL takes a pair of iterators and returns an iterator to the first occurrence of an element with the smallest value (according to the less-than operator for the contained type, `T`). `min_element()` works with sequences of any type, including *arrays*, as illustrated below:

```
string a[] = {"e","a","c","b"};
string* pos = min_element(a, a+4);
```

This call causes an implicit instantiation of a `string` version of the function, because `min_element()` is itself a function template:

```
// A typical implementation of std::min_element
template<class Iter>
Iter min_element(Iter b, Iter e) {
    if (b == e)
        return e;
    Iter pos = b;
    while (++b != e)
        if (*b < *pos)
            pos = b;
    return pos;
}
```

(Note that the end-delimiter, `e`, logically points to a position *one past the end* of the sequence—this is idiomatic for C++.) It doesn't matter whether `min_element()` is called with an iterator type (like `Sequence::iterator`) or a pointer type (like `int*`), as long as those types support the needed operations (`++`, `*`, `==`, and `!=`). Iterators are just types that *act* like pointers (via operator overloading), but refer to elements of sequences other than arrays. All STL sequences provide iterator types via the nested type name `iterator`.

The Stuff of Computer Science

Robert 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.

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

The following program illustrates some of the query algorithms.

```
// A "greater-than-10" predicate function
bool gt_10(int n) {
    return n > 10;
}

// A convenience function for displaying output
void display(int* start, int* end, int* pos) {
    if (start != end)
        cout << "found " << *pos << " in position " << pos-start << endl;
}

int main() {
    int a[] = {10, 1, 20, 2, 2, 1, 30, 3};
    const int N = sizeof a / sizeof a[0];
    int* p = find(a, a+N, 3);
    display(a, a+N, p);
    p = find_if(a, a+N, gt10);
    display(a, a+N, p);
    cout << "# of 2's: " << count(a, a+N, 2) << endl;
    cout << "# > 10: " << count_if(a, a+N, gt_10) << endl;
}
```

Output:

```
found 3 in position 7
found 20 in position 2
# of 2's: 2
# > 10: 2
```

On the call to `find`, the sequence delimiters are pointers to `int`, so `int*` is inferred as the iterator type; consequently, an `int*` is returned to indicate the location of the sought-after value. The end-delimiter (`a+N`) is returned if the value is not present in the sequence. Similarly, `find_if` returns the location of the first sequence value that satisfies the predicate function passed as its third argument (`gt_10`).

The example above uses a fixed-size array, but could be easily altered to read an unknown number of values from a file into an expandable container:

```
typedef vector<int>::iterator iter;
void display(iter start, iter end, iter pos) {
    if (start != end)
        cout << "found " << *pos << " in position " << pos-start << endl;
}

int main() {
    vector<int> a;
    copy(istream_iterator<int>(cin), istream_iterator<int>(),
         back_inserter(a));
    iter p = find(a.begin(), a.end(), 3);
    display(a.begin(), a.end(), p);
    p = find_if(a.begin(), a.end(), gt10);
    display(a.begin(), a.end(), p);
    cout << "# of 2's: " << count(a.begin(), a.end(), 2) << endl;
    cout << "# > 10: " << count_if(a.begin(), a.end(), gt_10) << endl;
}
```

The iterator type here is `vector<int>::iterator`. Iterator objects “pointing” to the first and one past the last elements are obtained via the vector member functions `begin()` and `end()`. The `copy` algorithm expects three iterators: two to delimit the input sequence and one to locate the output sequence. In this case, the `istream_iterator` *iterator-adaptor* function creates an iterator that “traverses” the standard input channel as the input sequence. The second `istream_iterator` represents the one-past-end marker for input streams. The `back_inserter` iterator adaptor wraps an STL sequence in an iterator that appends a value to that sequence by calling the appropriate `push_back()` function whenever it is written to.

Function Objects and Adaptors

A function object in C++ is an instance of a class that can be called as a function because it has a member function named `operator()`.Using function objects in C++ is somewhat like using `lambda` in functional languages. To illustrate, here is a function object type that allows comparing its argument to an arbitrary value:

```
class gt_n {
    int n;
public:
    gt_n(int the_n) {
        n = the_n;
    }
    bool operator()(int m) {
```

```
        return m > n;
    }
};
```

An instance of this class could be created at runtime as follows:

```
p = find_if(a.begin(), a.end(), gt_n(10));
```

The advantage here is that *any value* can be made a comparator; it is not hard-coded into the function as it was in `gt_10()`.

Since elementary operations such as greater-than are so often used, STL provides a number of pre-defined function object types, a sample appearing in the following table.

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

With the exception of `negate`, these types generate *binary* function objects. This is handy when sorting. The following call sorts a sequence of strings in descending order:

```
sort(slist.begin(), slist.end(), greater<string>());
```

The `sort` algorithm uses the `less` function object if its third argument is omitted.

The `find_if` algorithm expects a *unary* function object as its third argument, such as `gt_n`, seen earlier. It is not necessary to write `gt_n`, though, because there is a way to *adapt* `greater` to be used as a unary function by fixing one of its arguments:

```
p = find_if(a.begin(), a.end(), bind2nd(greater<int>(), 10));
```

The `bind2nd` function is a *function-object adaptor*: it takes a binary function object and returns a unary function object that stores the original binary function and a value to be used as its second argument, leaving the first argument open.

Applications

To show 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, the input file

```
how now brown cow
```

would result in the output file

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

Given a suitable quoting function, a call to 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);
```

The call to `transform` sends each string in the stream, `infile`, as a parameter to `quote`, and writes the result to the output stream, `outfile`.

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 and Haskell. In ML, the following function, 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 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).

Summary

The generic style of programming as 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 leads 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!

---

[1] Sedgewick, R., Algorithms, Addison-Wesley, 1983, p. 4.