C++ toolkits™

STL in Action: Helper Algorithms

By Graham Glass

Abstract

This article introduces a new column called “STL in Action.” Each installment will describe a way to either utilize or extend the C++ Standard Template Library (STL). This article considers several approaches for making the STL algorithms easier to use and less prone to programmer error.

Background on STL

Most programmers who write C++ programs have to write or purchase a set of collection classes such as vectors, lists, and sets. Until recently, commercial collection classes have had one or more of the following drawbacks.

1. **Lack of portability**
   Since there was no standard set of C++ collection classes, versions available from different sources were incompatible. Therefore, it was very hard to decide how to return a collection of results from an object since one user of an object might be using collections from one class library, whereas another user might be using collections from another library.
2. **Lack of efficiency**
   Many early C++ collection classes were influenced by Smalltalk. Thus, they made heavy use of inheritance virtual functions. Although these features support very flexible software, they tend to reduce performance somewhat. Although this is not always a problem, the culture of C and C++ is highly driven by performance. Therefore, many programmers will not use libraries unless they perform within a few percentage points of hand-coded C equivalents.
3. **Lack of functionality**
   Dealing with the hidden `vptr` in classes with virtual functions makes it difficult to portably instantiate objects in shared memory, store them persistently, and transfer them across a network. These capabilities are increasingly important for many applications.
4. **Lack of extensibility**
   Traditional C++ collection classes place the code for algorithms that worked on the collections within the collections themselves. For example, code for sorting often ended up in a collection classed `SortedCollection`, and code for applying a function to every element in a collection often ended up in an abstract base class called `Collection`. This approach made it hard to extend existing classes to add new algorithms without editing and recompiling the vendor's source code, which for maintenance reasons is usually best to avoid.

5. **Lack of type-safety**
   Many traditional collection classes were not type-safe. Use of these collections required heavy use of casting. This goes against one of the important goals of C++: strong type-checking supported by the compiler. Although RTTI is now becoming available to support strong type-checking at run-time, there is a non-trivial performance overhead associated with using RTTI.
6. **Lack of flexible memory management**
   Most traditional C++ collection classes had their memory allocation policies woven deeply into the code of the containers. This made it hard (if not impossible) for programmers to allocate space for a collection from shared memory instead of from the local heap.

In response to this situation, the ANSI/ISO C++ standards committee decided to search for a standard set of collection classes to overcome these limitations. Alex Stepanov and Meng Lee proposed STL as the standard. STL was based on their successful work on the topic of "generic programming." In July 1994, STL was selected to become an important part of the ANSI/ISO C++ standard library.

A number of articles on the design and structure of STL and the draft C++ standard library have appeared in *C++ Report*. My columns will focus on ways to utilize and extend the Standard Template Library (STL). In this month's column I consider the use of "Helper Algorithms," which make the STL algorithms easier to use and less prone to programmer error.

Helper Algorithms

Feedback from the C++ community regarding the ease of use of the STL has been very encouraging. After a brief period of adjustment to the new philosophy, users seem to enjoy their new-found power. However, some programmers are concerned that STL algorithms can be accidentally misused. The following describes some problems with using STL and evaluates several solutions.

Potential Errors with Using STL

Consider the STL `count()` algorithm:

```
template< class InputIterator, class T, class Size >
void count
  (
  InputIterator first,
  InputIterator last,
  const T& value,
  Size& n
  );
```

This algorithm counts the number of elements in the range (`first, last`) that match `value` using `operator==` and add this count to `n`.

The `count()` algorithm makes several assumptions about how it is used. For example, it expects iterators that reference the same container. It also expects that `n` is not automatically initialized to zero prior to the counting procedure. However, there is nothing to stop a programmer from messing up like this:

```
vector<int> v1 (10);
vector<int> v2 (10);
// ... fill v1 and v2 with values.
int n; // Oops, n should be initialized to zero!
count (v1.begin (), v2.end (), 42, n); // Oops!
```

The Class Wrapper Solution

One possible way to help programmers to avoid this is to add a new derived class to each STL container that contains "user-friendly" versions of the most common algorithms. For example:

```
template < class T >
class nonstandard_vector : public vector<T>
  {
  public:
    int count (const T& value) const
    {
      int n = 0;
      // Call standard.
      count( begin(), end(), value, n );
      return n;
    }

    // ... rest of easier interface goes here ...
  };
```

This approach would allow you to write:

```
nonstandard_vector<int> v1 (10); // NONSTANDARD VECTOR!
int n = v1.count (42);        // Easier.
```

An alternative approach is to encapsulate every STL container in a nonstandard and easier-to-use version:

```
template< class T >
class nonstandard_vector
  {
  public:
    int count (const T& value) const
      {
      int n = 0;
      // Call standard.
      count( v.begin(), v.end(), value, n );
      return n;
      }

    // ... rest of easier interface goes here ...

    vector<T>& standard()
      {
      return v; // Return encapsulated vector.
      }

  private:
    vector<T> v; // Encapsulated standard version.
  };
```

This approach would also allow you to write:

```
nonstandard_vector<int> v1 (10); // NONSTANDARD VECTOR!
int n = v1.count (42);        // Easier.
```

Evaluating the Inheritance Solution

Although these approaches seem reasonable at first glance, there are several problems:

- They double the number of classes.
- They require the user-friendly version of the algorithm to be declared in every extra nonstandard class.
- They require programs to be littered with nonstandard types, like `nonstandard_vector`, which in some ways defeats the purpose of having a standard in the first place.

- The first approach reduces efficiency by introducing unnecessary inheritance and virtual destructors.
- Both approaches only provide user-friendly versions of the algorithm for the nonstandard derived classes, and cannot be extended to provide user-friendly algorithms that work with other vendors' containers.

In other words, the class-wrapper approach is contrary to the original aims of the STL design.

The Helper Algorithm Solution

An alternative helper algorithm solution does not have any of these problems and is in perfect harmony with the STL philosophy. A helper algorithm is a templatized nonmember function that does some of the error-prone work that a programmer would otherwise have to do manually, but is just as efficient as the original. For example, here's the source code of a helper function called `ez_count()` that makes the standard `count()` algorithm easier to use:

```
template< class Container, class T >
inline int // Inline for efficiency.
ez_count (const Container& c, const T& value)
  {
  int n = 0;
  // Call standard algorithm.
  count (c.begin (), c.end (), value, n);
  return n;
  }
```

Here is an example that uses this helper algorithm:

```
vector<int> v1 (10);           // Standard vector!
int n = ez_count (v1, 42);     // Easiest!
```

Evaluating the Helper Algorithm Solution

The significance of helper algorithms becomes more apparent when you consider an operation such as removing items from a container. Without the helper algorithms, the following code is required to remove all of the `42`s from the vector `v`:

```
v.erase( remove( v.begin (), v.end (), 42 ), v.end () );
```

The `ez_erase()` helper algorithm allows you to write the following code instead:

```
ez_erase (v, 42);
```

Here is an implementation of the `ez_erase()` helper algorithm:

```
template< class Container, class T >
inline void
ez_erase (Container& c, const T& t)
  {
  // Call standard algorithm.
  c.erase (remove (c.begin (), c.end (), t), c.end ());
  }
```

Another candidate helper function is `ez_release()`, which assumes that every element of its container argument is a pointer to some heap and deletes each item:

```
template< class Container >
inline void
ez_release (Container& c)
  {
  Container::iterator i;
  for (i = c.begin (); i != c.end (); i++)
    delete *i;
  }
```

Helper versions can be written for many of the STL algorithms.

For additional technical information on Recursion Software
products and programs or for information on how to order and evaluate
Recursion Software technology, contact us today!

Recursion Software, Inc.
2591 North Dallas Parkway, Suite 200
Frisco, Texas 75034
1.800.727.8674 or 972.731.8800
www.recursionsw.com

RECURSION SOFTWARE, Inc.