APPLICATION OF THE C++ STANDARD LIBRARY TO PROBLEMS IN PATTERN RECOGNITION AND RECONSTRUCTION

T. Burnett
University of Washington

Abstract

An extremely important consideration in designing analysis and reconstruction code is the use of efficient data structures. The basic theme of these lectures is the application of techniques based on the Standard Template Library (STL), particularly generic programming, to problems encountered in reconstruction of data from detectors. The objective is to become familiar enough with the STL to apply it to data-handling problems.

1 INTRODUCTION

All aspects of software for High Energy Physics (HEP) experiments are in the midst of a revolution as basic paradigms change. The lectures of A. Dunlop[1] are focussed on the high-level design issues implied by use of object orientation. These notes address a complementary aspect at the other end of the spectrum, so to speak: what tools does one have to implement algorithms? For example, how are lists of objects stored and maintained? Such questions may not only influence the over-all design, but can have profound affects on programmer productivity, performance, readability, maintainability and even correctness of the code.

I refer to an integral part of the specification of the C++ language, a set of classes called the Standard Library, and especially to the component called the Standard Template Library, or STL. As I hope will become clear, there is an enormous benefit to be gained by being familiar with the techniques involved with this revolutionary development. This style is called *generic programming.* It has been observed [2] that it represents yet another programming paradigm that is not really OO, and isn't supported by the standard OO design methodologies.

The design of the STL is made possible by the "template" feature of the C++ language, which can also be described as parameterized types, or compile-time polymorphism. This allowed the ideas of Stepanov[3] to achieve recognition and wide use.

My objective in these lectures and the accompanying exercises is to start you on the road to putting the Standard Library in your toolbox. For those in the process of learning C++, there are a number of less obvious elements of the language itself that we shall have to deal with. It is clearly impossible to cover all there is to know. The reference book[4] has some 600 pages, probably corresponding to at least a full computer science course. But you should take away an understanding of the concepts, some experience in choosing which elements to apply to specific problems, and, most importantly, familiarity with resources. The ultimate goal is to learn to extend the library.

2 FROM FORTRAN/ZEBRA TO STL

It is often easiest to start with a simple example. Thus I have extracted (and simplified a little) a typical code fragment from the routine `GDRELA of GEANT 3.21.`

```
C      Mixture/compound : Loop over chemical constituents.
C
       DEDX = 0.
       DO 10 L=1,NLMAT
          AA = Q(JMIXT+L)
```

```
            ZZ = Q(JMIXT+NLMAT+L)
            WGHT = Q(JMIXT+2*NLMAT+L)
            CALL GDRELP(AA,ZZ,DENS*WGHT,T,DEDXC)
            DEDX = DEDX + WGHT*DEDXC
 10     CONTINUE
```

It calculates the dE/dx at energy `T` for a mixture of `NLMAT` elements, with density `DENS`. The element parameters are stored in a Zebra bank starting at `Q(JMIXT+1)`. A subroutine `GDRELP` calculates dE/dx for a given element. So the elements of this simple calculation are:

- A structure with the parameters for each element
- A container grouping these structures
- Memory allocation and management
- A scheme to loop over the element structures, performing a calculation

A sidelight of the above code is that since the index `JMIXT` is in a common block, and an external routine is called within the loop, optimization of the several index calculations that would have to assume that `JMIXT` is constant is not possible. This is a serious problem for this sort of code.

Now let's look at how some of the above would be rephrased in C++ with the STL. First, define a structure for the atomic element:

```
struct Element {
    Element(float a=0,float z=0,float w=0)
        :aa(a),zz(z),wght(w){};
    float aa,zz,wght;
};
```

This differs from a C struct by having a constructor with defaults, which is needed to be in a container. (I use `struct` instead of `class` for simple structures without access control.) Now, let's define a container of `Element` objects:

```
vector<Element> mix;
```

This is perhaps the most useful of the STL parameterized container classes.

One of the guidelines for using STL containers is to avoid explicit loops, especially when an appropriate *algorithm* exists. In this case it's called *accumulate*. To use it we have to provide an object that does the computation in the loop. It is a *function object* :

```
struct Gdrelp {
    double operator()(double dedx, const Element& e){
          return dedx + gdrelp(e, dens, t );
    }
};
```

Note that it defines an `operator()` that adds a quantity depending on an Element, the second argument, to its first argument and returns it. Now we accomplish our goal with only one statement:

```
double dedx = accumulate( mix.begin(), mix.end(),0.0, Gdrelp() );
```

This uses another of the basic elements of the STL, the *iterator*. The class `vector`, and in fact all of the container classes, have member functions `begin()` and `end()` that return pointer-like iterators to the valid range, where `begin` refers to the first element, and `end` to one past the last one.

A skeptic might comment here, "Hey that looks elegant, but it's gotta be less efficient: I see four function calls in addition to the one gdrelp call per element."

Indeed function calls are to be avoided. But we are really taking advantage of one of the STL's design principles, that use of STL will always be at least as efficient as hand-coded C. Indeed, this is an objective of C++ in general, one should not compromise performance for elegance of expression. It works because all of the function calls above in fact can be resolved by *inline* code. Thus the STL depends on three features of C++ that are not in C: templates, function overloading, and inline code.

3 THE STANDARD TEMPLATE LIBRARY

The above example illustrates the major elements of the STL portion of the Standard Library, which are,

- Containers: besides `vector`, there are `list`, `deque`, `set`, `multiset`, `map`, and `multimap`
- Iterators*:* there are five kinds, known as input, output, forward, bidirectional, and random access. The example would work with any of the last four.
- Algorithms: there are more than 90 different algorithms, like accumulate, that take as arguments iterators to determine a range, and perhaps a "predicate".
- Function Objects: a generalization of the notion of a function.
- Adaptors: a set of templated classes and functions that modify some behavior of one of the other components

(We will not cover the other element, allocator, which allows customization of memory allocation.)

The unique aspect of the STL is its separation into the above components, with the idea that the algorithms can be designed generically, without regard to details of the container or storage mechanism. Thus there are few function members of container classes that implement algorithms, the exceptions being only if the nature of the storage mechanism allows a more efficient implementation, searching being the best example. The relationships are illustrated by Figure 1. Generic algorithms access container data via iterators.

Figure 1. Illustrating the relationships among the major STL components

Another design goal is efficiency, as mentioned above. There are performance guarantees for most STL operations, expressed in terms of the size of the container, *n*. For example the standard search is linear, so takes a time of *O(n)*. Other possibilities are constant time, $O(1)$, $O(n^2)$, $O(\log(n))$, and $O(n \log(n))$.

STL retains a close connection to its roots in the C++ language. Containers, iterators, and function objects are C++ objects, yet all have equivalents in standard C or C++. For example,

```
int a[10]={1,2,3,4,5,6,7,8,9,10};
cout << accumulate(a, a+10, 0);
```

produces the output 55. It demonstrates that an ordinary C array is a container, and that an ordinary C pointer behaves like an iterator. [It also shows the nice possibility of initializing an array.]

It should be clear that such a rich and complex system cannot be learned quickly: these notes are only a brief introduction to the basic ideas, with some detailed examples. The book by Plauger et al.[4] is not only a good reference, but has many more detailed examples. In addition, there is a good online reference [5], also by Plauger.

Rather, I hope to make an impression of not only the utility of the various STL elements, but the usefulness of the generic programming style. STL is designed not only to be used, as is, but also to be extended. Thus I continue with some semi-realistic examples from HEP pattern recognition and reconstruction, to cover many of the features of STL, and acquaint the reader with the style implied by use of the STL. Since there is no substitute for actual experience, the exercises accompanying these lectures will challenge students to extend these examples.

4 SILICON STRIPS: AN EXAMPLE

We will now examine a prototypical example from HEP reconstruction. It is a hodoscopic array of particle detectors, exemplified by silicon strip detectors used in many modern experiments. A schematic representation is shown in Figure 2.

4.1 Hits

The basic "unit" of information is a single "hit", characterized by the identity of the strip. We'll assume that only the presence of a hit is stored, that there was enough ionization to exceed a discriminator threshold. For now at least, this is a simple integer, the index of the strip within the array. To distinguish it, and to allow replacement with a class if appropriate, it is important to give a name for this concept:

```
typedef unsigned StripId;
```

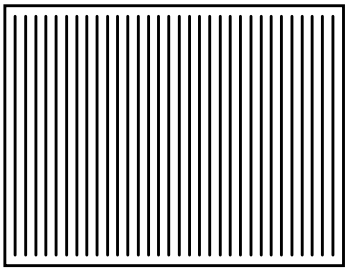

Figure 2. Schematic illustration of a silicon strip particle detector. The parallel lines correspond to strips, each of which can be "hit"

4.2 Collections of Hits

Now the data from such a detector is a collection of hit strips. An immediate design decision is how to represent the data, or, in the present context, which STL collection class is most appropriate. Considerations are access convenience and performance. Since what we want to represent is the *set* of hit strips, then perhaps `set` is the appropriate choice. Other choices are `vector` and `list:` the former is best for random access to elements of the collection, the latter for constant-time insertions anywhere. But `set` has the nice property of ordering the elements for efficient searching. Again we express our choice with a `typedef` statement. Thus let us try

```
typedef set<StripId> StripList;
typedef StripList::iterator StripIterator;
```

The template header for `set` is:

```
template<class Key, class Pred = less<Key>,
      class A = allocator<T> >
class set { /* ... */};
```

This specifies that `StripId` is a *key*. It must support an `operator<()` to allow sorting, which is what the default function template `less<Key>` provides.

Now we can use the `set` member function, `const_iterator find(const Key& key) const;` to efficiently determine if a given strip is set.

4.3 Cluster finding

As an example of an algorithm to apply to our set of hit strips, consider one of the first tasks of pattern recognition, namely grouping hits into *clusters* of neighboring strips. The idea is pretty simple, just looping over the hits, and marking the positions where non-adjacent breaks occur. Thus we want to encapsulate the notion of adjacent strips in a way that can be used by STL algorithms, that is, by defining a special function object:

```
struct Adjacent : public binary_function<StripId,StripId,bool>
{
       result_type operator()(
              first_argument_type x,
              second_argument_type y)const
              { return abs(x-y)==1; }
};
```

Compared with the function object example that we introduced before, this has several important features:

- It inherits from `binary_function.` This allows it to be used with function adapters, like `not2`.
- The function argument and result types use the definitions of `binary_function`
- The function is declared `const`, allowing it to be used with const objects.

Furthermore, we want to create a new structure to record the position and width of the cluster. Since strip id's already "know" their position, all we need is to associate the number of adjacent clusters with the lowest strip id. This is quite easily done using a *map*, with `StripId` serving as the key. Thus let's define a type.

```
typedef map<StripId,unsigned> ClusterList;
```

Now for some code: New STL facilities are indicated by bold face.

```
StripList hits;
//
//      load hits with data ...
//
ClusterList clusters; // declare a map of clusters
iterator begin = hits.begin(), end=hits.end();

for(iterator k = begin; k !=end; ) {
    iterator ks=k;  // first iterator for cluster
    k = adjacent_find( k, end, not2(Adjacent()) );
    if( k!=end ) ++k;
    clusters[*ks]=distance(ks,k); //insert a cluster
}
```

This bit of code uses several STL features, pointed out by

- The *template function* `not2` is used to reverse the meaning of `Adjacent`. The workings of the templates involved dictated the form we used for `Adjacent,` as opposed to the simpler declaration

```
struct Adjacent {bool operator()(StripId x, StripId y)const;};
```

- The algorithm `adjacent_find` is used to find the next adjacent pair of elements in the set that satisfy the given predicate. In this case it searches from the current position to the next one that is not adjacent to the next one. Like all such algorithms, it returns `end()` to indicate failure. (A form without the predicate tests that they are equal, which cannot happen in a set, for which all elements must be unique.)
- The template function `distance` is used to determine the number of strips in the cluster by taking the iterator difference.

- I used map's handy `operator[]` to insert a new element and `operator=` set its value. This is equivalent to, but more expressive than the following:

```
clusters.insert(make_pair(*ks, distance(ks,k)));
```

Also note the heavy use of iterators, typical of this style of programming.

Now that we have a set of clusters, here is an example showing how to access the contents of a map:

```
cout << clusters.size() << " clusters found." << endl;
for( ClusterList::iterator it= clusters.begin();
    it!=clusters.end(); ++it) {
      cout << "(" << (*it).first << "," << (*it).second << ") ";
}
cout << endl;
```

Since the `value_type` of a map is `pair<key,referent>,` we use the data members `first` and `second` to access the key and referent above. Finally, assuming that we have defined

```
operator<<(ostream&, const ClusterList::value_type&),
```

we can rewrite the `for` loop using STL to avoid the explicit loop:

```
copy( clusters.begin(), clusters.end(),
      ostream_iterator<ClusterList::value_type> >(cout, " ") );
```

This introduces the extremely useful `copy` algorithm, and the template class `ostream_iterator,` which creates an output iterator suitable for copy.

5 AN ARRAY OF STRIPS DETECTORS

Now, let use extend our single strip detector of the last section to a group of such detectors, arranged as parallel planes. This adds another dimension, or index, to the notion of a strip id, since now we have to identify which detector has a particular strip.

5.1 Generalized StripId

One possible strategy is to define a new container of the sets of strip ids, say `vector<StripList>.` The disadvantage of this, especially from the STL standpoint, is that iterations require two loops, with two different types of iterators. So we'll implement instead a scheme whereby `StripId` becomes smarter. Instead of being an `unsigned,` we'll represent it by a pair of `unsigned`'s, one for the strip index as before, the other for the layer. This requires a class definition that meets the requirements for being in a "controlled sequence" as containers are known. In particular, ordered sequences, like set and map, require that a predicate exist that defines a strict ordering. The default is `less<T>`, which is defined as follows:

```
template<class T>
    struct less : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const{return x<y;}
    };
```

Since `set<T>` uses `less<T>`, `T` must be intrinsic or a class defining an `operator<().`

The result is that our new definition of StripId is:

```
class StripId : private pair<int,int> {
public:
      StripId(int strip=0, int layer=0);
      bool operator<(const StripId& rhs)const;
      bool adjacent(const StripId& rhs)const;
private:
      int m_layer, m_strip;
};
```

Note that we define a member function `adjacent` to facilitate testing for adjacency, and since the subtraction operator is not defined for strips in different layers.

5.2 A pattern recognition algorithm: the Hough transformation

The Hough transformation (HT) technique was reviewed in the context of imaging techniques relevant to HEP reconstruction at the last School [6]. Comprehensive reviews also exist [7]. The HT consists of transforming each data point to a set of possible points in a parameter space (slope and intercept for our 2-d example), then making a histogram in the parameter space, called the accumulator, and choosing the parameters that contribute to the largest bin or bins in the histogram. This gives us more opportunity to apply STL components to a real-world application.

The first step is to design a class to represent the point in parameter space, and to consider what functionality such a class should have. Since the meaning of the parameters is to describe a line, we might as well give it the responsibility of calculating the line, thus encapsulating the parameter representation. So it may also be a function object! Here is the proposal:

```
class Trajectory {
public:
      Trajectory(double intercept, double slope);
      double operator()(double z)const{return z*m_slope+m_intercept;}
private:
      double m_intercept, m_slope;
};
```

Note the `operator()` that evaluates the function. $f((a,b),z) = az+b$, where a and b are slope and intercept, respectively. [I define the coordinate system such that strip planes are perpendicular to the z-axis, and strips along the y-axis, so that an individual strip is an x measurement.]

Next, we need a mechanism to evaluate the inverse function: that is, given a point in our detector space, and the value for an independent parameter, say the intercept, return the other parameter, or slope. That is, $g((x,z),b) = (x-b)/z$. (Beware z=0.) A function may be adequate,

```
double inverse_trajectory(double intercept, const StripId& strip)
{
      return strip.x()-intercept)/strip.z();
}
```

where we have extended the StripId capability to calculate, or have calculated, the coordinates of its strips. Better is probably to make a Trajectory object from a strip and an intercept with an alternate constructor.

5.2.1 A binning iterator

The heart of the HT algorithm is devising a sensible strategy for defining, and binning the parameter space. We'll do something naïve, since we are concentrating on how such algorithms are implemented here. Another ingredient is a way to represent the list of intercepts. Since we need to iterate over this list, it sounds like a job for a special iterator. Writing such an iterator is a chore at first, and much harder than simply writing a function to calculate bins. But it fits well into the generic programming environment, and can be used or adapted.

We'll write the simplest iterator to implement, an input iterator. It needs only to be dereferenceable, define both pre- and post-increment operators, and a comparison operator to check for the end of the sequence (which has to be defined). Here it is:

```
class bin_iterator : public iterator<input_iterator_tag, double, void> {
public:
    typedef bin_iterator _Myt;
    bin_iterator(double start=0, double delta=0, unsigned count=0)
      : m_start(start), m_delta(delta), m_count(count), m_index(0){}

    const value_type  operator*()const
    { return m_start+m_index*m_delta;}

    _Myt & operator++()
    {
      if(m_index < m_count) ++m_index;
      return *this;
    }

    _Myt operator++(int)
    {
      _Myt tmp(*this);
      ++*this;
      return tmp;
    }

    bool operator !=(const _Myt & rhs)const
    {
      if( rhs.m_count == 0 )
          return  m_index < m_count;
        return m_index != rhs.m_index ;
    }

private:
    double m_start, m_delta;
    unsigned m_count;
    unsigned m_index;
};
```

Note that the default constructor creates an object that can be compared with the end of the sequence. Also, it inherits from a STL templated class iterator, which defines some types. As an example of its use,

```
copy(bin_iterator(0,1,5), bin_iterator(),
      ostream_iterator<double>(cout, “, “) );
```

generates the output “0, 1, 2, 3, 4,”. This also shows usage of the copy algorithm, and the very convenient `ostream_iterator`.

5.2.2 The Hough accumulator

Now we are close to our goal. The algorithm, briefly, is:

1. Generate, or read in, a list of `StripId` hits. We represent this by an STL set.
2. For each hit, and for each chosen value from a list of possible intercepts, create a `Trajectory` object that calculates the slope using the inverse function.
3. Make the slope discrete (i.e., bin it)
4. Add the `Trajectory` object to a list that will allow analysis of accumulations in parameter space: in this case equivalent to a Lego histogram. This is conveniently done with an STL map, with `Trajectory` as the key, and a unsigned as the referent. This is the accumulator in Hough terminology.
5. Find the largest bin or bins in the accumulator, and look up the corresponding parameters.

The following code illustrates this:

```
typedef set<StripId> StripList;
typedef map<Trajectory, unsigned> Accumulator;
StripList data;
Accumulator acc;
// read in or create the data
for( StripList::iterator dit=data.begin(); dit!=data.end(); ++dit) {
    for(bin_iterator iit(-0.2, 0.05, 9); iit!=bin_iterator(); ++iit) {
        Trajectory t(*iit, *dit);   //create a trajectory
        t.discretize( 0, 0.01);     //make its slope discrete
        acc[t]++;                   // add it to the map
    }
}
```

The `Trajectory` class is as defined above, with the addition of a constructor to compute it from a StripId object. In order to be a key, it now has an operator<(). Also, it has a `discretize` function member that modifies either or both parameters. Finally, notice how easy it is to make a histogram using a map. The expression `acc[t]++` is almost magical: it creates an entry for `t` if one doesn't already exist, then increments the corresponding counter.

The final step, which we won't go over here, is to analyze the accumulator. STL algorithms could be used to remove bins below a certain threshold, or to create a new list sorted on the bin count.

6 SUMMARY

We have examined the consequences of applying the STL to some simple problems encountered in HEP reconstruction, and emphasized that the generic programming philosophy behind the STL can be used to simplify implementation of complex algorithms.

References

[1] A. Dunlop, *Modern Object-Oriented Software Development*, these proceedings.

[2] A. Langer and K. Kreft, *Combining Object-Oriented Design and Generic Programming;* C++ Report, March 1997.

[3] Alexander Stepanov and David R. Musser: *Algorithm-Oriented Generic Software Library Development*. HP Laboratories Technical Report HPL-92-65.

[4] P. J. Plauger, Alexander A. Stepanov, Meng Lee; *Standard Template Library : A Definitive Approach to C++ Programming Using STL*; Prentice Hall, Publication date: August 15,1997; ISBN: 0134376331

[5] http://www.dinkumware.com/htm_stl/index.html

[6] R. K. Bock, *Techniques of image processing in high-energy physics*, 1996 CERN School of Computing. A

[7] P. Ballester, *Hough transform for robust regression and automated detection*, Astron. Astrophys. 286, 1011 (1994).