Standard Template Library

The joys of Generic Programming

Erik Hugne ehe02001@student.mdh.se
Juan Maluenda jma02002@student.mdh.se

18th May 2005

Abstract

This report is meant to provide the reader with insight into the Standard Template Library, hereby referred to as STL, and the concept of generic programming. Since STL relies heavily upon templates, we have dedicated a chapter to explain this feature in C++. The study focuses mainly on the implementation of STL, we will state problems and solve these by providing implementation examples where we combine different parts of STL into a solution. We will explain why generic programming using templates (STL) does not impair the efficiency in terms of execution time and code readability. On the contrary, STL decreases the programming effort required to solve a problem significantly, and fewer lines of code makes it easier to understand someone else's work. We will also give a brief introduction to Boost and its contributions to the C++ standard.

Contents

1 Introduction

In the late 1970's Alexander Stepanov first observed that some algorithms do not depend on some particular implementation of a data structure, but only on a few fundamental semantic properties of the structure. For example, how to step from one element in a data structure to the next. In 1992 He started writing a library together with Meng Lee with the intention to show that algorithms can be defined as generically as possible without loosing efficiency. This library would later be known as the STL, and was adopted as a C++ draft standard in 1994. In order to understand how STL works, it is important to familiarize yourself with templates, explained in section 3.

2 Why is STL needed?

This question should not need to be asked, it is like asking why you shouldn't reinvent the wheel. Generic programming is about writing your classes and algorithms general enough to be reused in a wide variety of situations. In C++, class and function templates are particularly effective for generic programming because they make the generalization possible without sacrificing efficiency. STL is the C++ programmers ultimate toolbox. It is a formidable collection of everyday algorithms and containers.
Do you need a dynamic container in you C++ program?
3 lines of code will give you a STL container of your choice. You do not need to worry about petty things like memory allocation and deallocation. basically the only thing you have to concider is what type of container to use.
Do you need the elements in your container sorted?
Just use one of the formidable sorting functions in the extensive STL algorithm library. The majority of C++ programmers in the beginner/intermediate level have probably used Vectors and Iterators in their projects, but STL is capable of so much more than just holding and accessing data in a container, we will try to give you some insight into what it can do for you.

3 Templates

To be able to write generic classes and functions, we need to find a way to define classes and functions so that they do not depend on what data type they process. C++ has a feature called templates which will solve this problem for us. The concept of templates can best be described with an example. Below we have a function that swaps the contents of two integer variables.

```
void swap (int& a, int& b)
{  int tmp = a;
   a = b;
   b = tmp;}
```

A problem arise if we want to swap floats and doubles too, we would have to write separate functions with the same logic but for different data types. The solution is to use a template, also called parameterized types.

```
template <class T>           //equivalent to template <typename T>
void swap (T& a, T& b)
{
   T tmp = a;
   a = b;
   b = tmp;
}
```

3.1 Operator overloading

We now have a generic algorithm that can swap the values of any data type. If we are writing a template function that involves using operators other than assignment on a parameterized type, the operator used in the template function must be defined for the types being compared. For example if we have a template function that returns the smallest of two elements.

```
template <class T>{
T& min (T& a, T&b) { return a < b ? a : b;}
```

And we want to compare two elements of the type CBitmap where operator< operator applies to the CBitmap's size.

```
Class CBitmap
{
private:
Pixel *data;
int width;
int height;
public:
bool operator< (const CBitmap& n) const
{
return (sizeof(Pixel)*this.width*this.height) <
(sizeof(Pixel)*n.width*n.height) ?
true : false;
}
}
```

We have overloaded the $<$ operator for the CBitmap class and can now use it in our template function min.

```
CBitmap picture1("pic1.bmp");
CBitmap picture2("pic2.bmp");
CBitmap *result;
*Result=min(picture1,picture2);
```

What really happens is that when the compiler/linker finds a reference to a template function in the source code, it creates an instance of the function for the specified type. Since this is done in deployment time, we will not notice any difference in runtime performance. The executable code is the same as if we would have written a separate function for each type used. There are two approaches for template instantiation, the Borland model and the CFront model, we will not discuss these since it is out of scope for this document.

4 Generic Algorithms

Now that we know how to separate the algorithms for solving a problem from the data being processed, let's make some good use of it. Take a look at this simple implementation of the binary search algorithm.

```
//array must be sorted
const int* binary_search (const int* array, int n, int& x)
{
   const int* lo = array, *hi = array + n, *mid;
   while(lo != hi)
   {
      mid = lo + (hi - lo) / 2;
      if (x == *mid) return mid;
      if (x < *mid) hi = mid; else lo = mid + 1;
   }
   return NULL;
}
```

This only works with sorted integer arrays, to make it generic we simply transform it into a template function.

```
template<class T>
//array must be sorted
const T* binary_search (const T* array, int n, const T& x)
{
   const T* lo = array, *hi = array + n, *mid;
   while(lo != hi)
   {
      mid = lo + (hi - lo) / 2;
      if (x == *mid) return mid;
```

```
        if (x < *mid) hi = mid; else lo = mid + 1;
    }
    return NULL;
}
```

In short, add the template<class T> or template<typename T> definition above your function, and replace relevant occurrences of int with T.

4.1 STL algorithm library

STL holds 60 different algorithms in 8 main categories.

- Nonmodifying Sequence Operations Extract information, finding a element, or moving through a container. Example: find()
- Modifying Sequence Operations All sequence operations that perform some change to the elements. Example: fill(), transform(), swap().
- Sorted Sequences Sorting and bound checking functions. Example: sort().
- Set Algorithms Creation of sorted unions, intersections etc. Example: set_union(), set_intersection().
- Heap Operations A set of operations for creating and interacting with a heap structure. Example: make_heap(), sort_heap().
- Minimum and Maximum Comparing elements Example: min(), max().
- Permutations Operations for finding permutations of a sequence of elements. Example: next_permutation().
- Numeric A set of generic numerical algorithms. Example: partial_sum().

5 Functors

Some of the generic algorithm's ends with _if, like replace_if() and remove_if(). This indicates that they will perform their task if a certain criteria is met. This criteria, the predicate, can be evaluated either true or false. There are two kinds of predicates, unary and binary. These predicates are known in C++ STL as Functors or Function Objects. You define these as a class, and add a definition of the operator().

5.1 Generalized functor

```
template <class T>
class is_greater_than
{
  bool operator() (const T& a, const T& b) const
    {return (x>y);}
};
```

We can now use the STL sort() function and sort the elements in a container according to our defined binary predicate.

```
sort(v.begin(), v.end(), is_greater_than);
```

The first and second parameters returns iterators of the container v, pointing to the beginning and end. We will discuss these in the Iterators section. Since the functors like the one defined above is also pretty general, the most commonly used are of course incorporated in STL. There are however some some cases when you need to define your own functor. You might want to sort a collection of CBitmap's according to the amount of red pixels in each bitmap. We will show how to do this in our case study of type specific functors

5.2 Generalization through subclassing

Functors can be generalized and defined as template classes just like functions. But as shown in our case study, you can also define type specific functors that perform more or less advanced operations on the type. It is also possible to write a functor for a certain base class that enforces necessary operations on the type for the functor. We could for example generalize the functor has_more_red_pixels, which compare bitmaps and returns a pointer to the bitmap with the most red pixels, to accept all classes derived from ImageObject. ImageObject is an abstract class that provides us with the functions required for the functor to do it's job. getWidth(), getHeight() and getPixel(). Note that the functor is still type specific even if we change it to conform to all derived classes from ImageObject.

```
class has_more_red_pixels : binary_function<ImageObject, ImageObject, bool>.
```

We will however benefit from this abstraction since we can use the functor on all types of classes implementing ImageObject. An alternative approach to this problem would be to

overload operator< in the ImageObject class, this implementation would look just like our has_more_red_pixels functor, and we can use the sort() algorithm like we would with a primitive type like int, without supplying a functor argument.

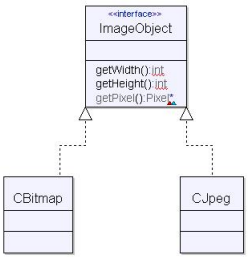

Figure 1: ImageObject derived classes can be sorted using our defined functor

6 Containers

Containers are as their name tell, an object that store something which in our case is data. This data can be normal intrinsic datatypes such as integers and doubles or they can be C++ defined object such as String and even user defined objects. This can of course be accomplished by using an array, but then you have to do all memory allocation/deallocation manually. The STL containers handle all sizing and memory allocation for you. Which means that you can never write outside the end of the container class such as you can with an array. They also contain utility methods and methods for insertion, deletion and copying. Utility methods are some simple but very helpful methods such as emtpy() which returns if the container is indeed empty or not, size() that returns the number of elements in the container and clear() that removes all elements in the container. Containers are very much optimized for their intended use and are divided into two main groups which we will now investigate further.

6.1 Sequence Containers

The first group are the sequence containers. These are designed to hold one type of object in a linear sequence. The values are also accessed in an order determined by their insertion, unless an sorting algorithm has been applied.

6.1.1 Vector

The first sequence container provided by STL is the Vector. It is very similar in apperance to the array and allows for array-type syntax (subscript operator), eg vector[3]. A vector allows for access of elements at any position at a constant time. Which means that the

access will always take the same amount ot time, no matter what position you choose to look at. This also leads to that vectors allow "random access" with little effort. Insertion and deletion at the end of a vector are also done at constant time. Insertion or deletion anywhere but the end is considered of linear time (the time it takes will depend on the size of the vector), since all of the elements must be traveresed and moved for the new value to be inserted. The following two examples do exactly the same thing, one operating on a vector, the other on a common array.

```
void printn(int n)
{
cout<<n<<endl;
}
int main(){
vector<int> vInts;
vInts.push_back(1);
vInts.push_back(3);
vInts.push_back(42);
std::for_each(vInts.begin(),vInts.end(),printn);
```

```
int* aInts;
int arraysize=0;
aInts=(int*) malloc(sizeof(int*));
aInts[arraysize++]=1;
aInts=(int*) realloc((void*)aInts,sizeof(int*)*2);
aInts[arraysize++]=3;
aInts=(int*) realloc((void*)aInts,sizeof(int*)*3);
aInts[arraysize++]=42;
for(int i=0;i<arraysize;i++)
      cout<<aInts[i]<<endl;
}
```

6.1.2 Deque

The second container, deque is very much like a vector with the main difference that it allows for insertion and deletion not just at the end, but also at the front.

6.1.3 List

The third container is the List. The list differs from the two former containers because its underlying structure is not like a common array, instead it has double linked list structure.

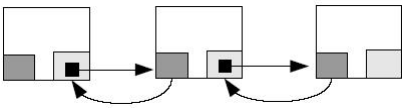

Figure 2: Double-Linked list structure

This means that the list does not support random access retrievals or subscript accessing. All retrievals will take a linear time, since it has to step through the list beginning at the front. But to its advantage insertion and deletion at any position will take constant time, since no elements need to be moved, mind that the time it takes to find the correct position is not constant.

Here is a small example of how you declare a STL container. list<int> cont; vector<int> cont; deque<int> cont; In this example you can very clearly see the use of templates that are a fundamental part of the STL.

6.2 Associative Containers

The second group of containers are the Associative Containers also known as Sorted Associative Containers. Associative containers are efficient at retrieving elements based on keys. This is beacause there internal structure are stored in a binary tree, which greatly enhances searches. They are sorted based on the key are the elements are automaticly sorted as inserted. These keys can then be the value or part of a pair where the key is used for retreving the value it is paired with.

6.2.1 Map and Multimap

The map container is like a index table. Each element in the map will store one unique key that will be used to index the map, and some kind of value that will be stored at the index position. The value can be a primitive or even a user defined class. The key can also be of user defined class, but then methods for comparing the keys must be provided when creating the map. eg. overloading operator< for the type, or supply an argument with a functor for comparing the types. When using keys of basic type the subscript operator can be used for accesing the elements.

A multimap is essentaly the same as a map, but it allows for creating non unique keys so that a key can be associated with several values. These will be stored as separate pairs.

6.2.2 Set and Multiset

A more simple version is the Set container. This is just a collection of elements with one datatype. The datatype serves as both value and key. Sets are useful for fast key retrievals or to test for prescence of special keys, they also support functions as unions, intersection and difference of sets for more advanced features. All values in a set must be unique.

The multiset container is a set container that allows non unique values, allowing duplicate entries.

As explained earlier all containers have methods or inserting och removing elements. Somethimes when the structure is of an array-type entity you can use a subscript operator , but in many situations this does not apply since the container class itself does not provide methods for direct access to its elements. Further more, what if you whant to manipulate or compare a group of elements in the container?

7 Iterators

The solution to this is the iterator. An iterator is an object that selects an element within the container and presents it for the user of the iterator, you can think of it as a more advanced pointer. It allows for traversing the container since this is something the container does not provide itself. So if an iterator points to one element in a range, it is possible to increment the iterator so that it points to the next element. The main advantage the iterator allows is that through it a level of abstraction is accuired. This abstraction is used to separate the details of accessing and manipulating elements in the container from the container, and allows for generic algorithms that apply on completely diffrenet types of containers. This way the container can be viewed as a simple sequence container to the user, regardless of its real type. The iterators used for accessing the elements are container specific. So the iterator's operations that will be available depend on what type of container we are using. And also or this reason, the iterator class definition is inside the container class.

```
Vector<int>::iterator it;
```

For different functionality there are different type of iterators. These together form a hierarchy that allow for more advanced types of iterators.

7.1 Input/Output Iterators

At the bottom hierarchy we have the input and output iterators. The input iterator only guarantees read access. It is possible to dereference an input operator to obtain the value it points to, but it is no possible to assign a new value. The output operator does the contrary, it allows or assigning new values but not to refer to that value. Both implement the incrementing operator ++ to step forward through the elements.

7.2 Forward Iterator

The forward iterator is a refinement of the input and output iterators, they support all their operations and also provide additional funtionality. It is also the first iterator that allows for multypass algorithms. It does not, as the name says support stepping backward through the elements.

7.3 Bidirectional Iterator

Bidirectional iterators support motion in both directions. The iterator may be incremented to obtain the next element, or decremented to obtain the previous.

7.4 Random Access Iterators

The most advanced iterators are the random access iterators they support addition of arbitary offsets, subscripting and subtraction of one iterator from another to say a few. Nedless to say is that all containers do not support every type of iterator implementation. An example would be the list container. Since it does not support random access, it does not return a random access iterator, instead it returns a bidirectional iterator. And can not be used with algoritms that demand random access iterators.

8 Boost

Boost is a very large header library, like STL, but it is not part of the C++ standard. Anyone can submit a component to Boost for evaluation, which is done by Boost users and developers. Since many components of Boost is experimental, they are not guaranteed to pass through all compilers. Moreover, many of the Boost components have been proposed as standard library additions.

8.1 Tuples

The tuple is a generalization of the std::pair<> template from STL. Unlike pair, which can only contain 2 elements, a default tuple can have up to 10 elements, the tuple can also be extended to hold more if necessary.

8.2 Smart Pointers

std::auto_ptr<> is used to store a pointer to an allocated object, and makes sure that the object it points to gets destroyed automatically when control leaves a block.

```
int f()
{
auto_ptr<T> pt( new T );
/*more code*/
return 0;
//auto_ptr and it's member ptr is deallocated automatically
}
```

std::auto_ptr<> have been around in STL for a while. One problem with this class is that it cannot be stored in STL standard containers. Boost have solved this with the

boost::shared_ptr<>. Which works like auto_ptr<>, with the extension that it can be stored in a standard container. The object is guaranteed to be deleted when the last shared_ptr<> pointing to it is destroyed or reset.

8.3 Lambda Functions

Boost have adopted some concepts from functional programming, namely Lambda Abstractions. These provide a convenient means of defining unnamed function objects for STL algorithms. This example iterates through a and prints out each element.

```
for_each(a.begin(), a.end(), std::cout << _1 << ' ');
```

Where we normally would place the reference to the functor for printing elements of a, we have the argument std::cout « _1 « ' ' It defines a unary function object where _1 is a placeholder for the function argument. For each iteration of for_each, cout« is called with an element of a as the actual argument. The following example shows a lambda expression assigning each element in v to 1.

```
list<int> v(10);
for_each(v.begin(), v.end(), _1 = 1);
```

Examples from Boost.org Lambda documentation.

9 Case studies

9.1 Time measurement

Lets say you whant to examine a realtime system. You whant to now the time of all methods called to a specific class. The information recieved will be a method name and a value. Since there is a high probabilty that the method calls will differ in measured time, and the result will be that we will have a duplicate of same method names but different time values. This would rule out the use of a map, since it requires a unique key. And the correct container for the task will be a multimap since it supports for nonunique keys. The information stored will be the method name as key, and its measured time as the value.

For our multi map we will need to include

```
#include <map>
#include <iostream>
using namespace std

typedef multimap<string,int> myMultiMap;

void myPrintFunc(pair<string,int> p)
{
cout<<p.first<<p.second<<endl;
}
```

Here we choose a string (method name) as key, and an int(measured time) as value.

```
myMultiMap methodmap;
methodmap.insert(pair<string,int>("method1",10);
methodmap.insert(pair<string,int>("method2",5);
methodmap.insert(pair<string,int>("method1",28);
methodmap.insert(pair<string,int>("method3",28);
```

Repeated until all methods are inserted. Note that since insertions are automaticly sorted, the third insert will be stored between the first two.

Lets find the values for method2. First we will need an interator pointing to the correct position.

```
myMultiMap::const_iterator it = methodmap.lower_bound("method2");
While(it != methodmap.upper_bound("method2")) {
cout << "Method2 took : " << it->second << endl;
++it;
}
```

Lower_bound(k) will find the first value associated with the key k, in form of a iterator ofcourse. Upper_bound(k) will return the first key whose value is higher then k. Another approach would be the use of equal_range() that returns a pair of iterators that mark the beginning and the end of the range.

```
pair <methodmap::const_iterator it,
methodmap::const_iterator it> range = methodmap.equal_range("method2");

for(methodmap::const_iterator it = range.first; i != range.second; ++i)
cout << it->first << it->second << endl;

//instead of the for statement, we can use an STL generic algorithm
for_each(range.first,range.second,myPrintFunc);
```

Accessing a specific element in the containers differ from container to container. In a map it is possible to use the subscript operator since all keys will be unique, so we are able to write like this.

```
cout << map["method2"];
```

But, this is the beatiful part, for traversing an entire container the syntax will always be identical, behold.

```
Vector<int> vector;
list<int> list;
map<int> map;

for_each(X.begin(),X.end(),myFunc);
```

Where X is exchanged to the desired container.

9.2 Type specific functor

Imagine we have a number of bitmap images, and we want to sort these according to the amount of red pixels in each bitmap. We will solve this problem using vector for holding the images, a custom functor for comparing images, and a STL generic sort function that we pass our functor to. This is our pixel struct, the smallest element of a bitmap.

```
struct Pixel
{
unsigned char r;
unsigned char g;
unsigned char b;
unsigned char a;
};
```

Class definition of the bitmap class.

```
class CBitmap
{
  Private:
    Pixel* pbuf;  //buffer to hold the pixels
    int width;
    int height;
  public:
    CBitmap(const string& file);
    ~CBitmap();
    Pixel* getPixel(int w, int h);  //returns the pixel at w/h
    int getHeight();
    int getWidth();
};
```

This is our functor, it returns true if a has more red pixels than b.

```
//inherit template binary_function for CBitmap
//because that's the only type we allow this functor to operate on
class has_more_red_pixels : binary_function<CBitmap, CBitmap, bool>
{
  public:
    bool operator() (const CBitmap& a, const CBitmap& b) const
    {
      int w,h;
      double aredpixels=0;
      double bredpixels=0;

      for(w=0;w<a.getWidth();w++)
```

```
      {
        for(h=0;h<a.getHeight();h++)
          if(a.getPixel(w,h)->r > 254){aredpixels++;}
      }
      for(w=0;w<b.getWidth();w++)
      {
        for(h=0;h<b.getHeight();h++)
          if(b.getPixel(w,h)->r > 254){bredpixels++;}
      }
    return (aredpixels>bredpixels);
  }
};

int main(void)
{
  //create our container and put some data in it
  std::vector<CBitmap*> bmpvec;
  bmpvec.push_back(new CBitmap("c:\\seal.bmp"));
  bmpvec.push_back(new CBitmap("c:\\pig.bmp"));
  bmpvec.push_back(new CBitmap("c:\\deer.bmp"));

  //sort bmpvec from beginning to end, using the generic sort algorithm
  //with has_more_red_pixels as binary predicate for comparison.
  sort(bmpvec.begin(),bmpvec.end(),has_more_red_pixels);
  return 0;
}
```

10 Conclusions

The joy of generic programming lies in doing as little of the programming by yourself as possible. The simplicity of using template code from STL, and it's wide area of application is one of the main reasons C++ has made it to be one of the most commonly used programming languages. Programmers with little or no previous experience can use complex datastructures and algorithms without knowing exactly what is happening when they for example push an element to a vector. This said, we hope that you are convinced that STL will make your life as a C++ programmer easier.

References

[1] J Skansholm *C++ Direkt, Studentlitteratur, 1996*

[2] http://www.sgi.com/tech/stl/ *SGI.com STL Documentation, 2005-05-17*

[3] http://www.boost.org/doc/html/ *BoosT Library Documentation, 2005-05-17*

[4] http://www.yrl.co.uk/phil/stl/stl.htmlx *Phil Ottewell's STL tutorial, 2005-05-17*

[5] http://www.josuttis.com/libbook/ *C++ STL Tutorial and Reference, 2005-05-17*

[6] http://www.camtp.uni-mb.si/books/Thinking-in-C++/DocIndex.html *Thinking in C++, 2nd ed. Volume 2 Revision 2, 2005-05-17*

[7] http://cplus.about.com/od/stltutorial/ *STL Tutorial - Overview of the Standard Template Library, 2005-05-17*