Integration of the Standard Template Library and the Microsoft Foundation Class

Paul Wolfgang and Yang Song

1 Introduction

Both the Microsoft Foundation Class (MFC) [1] and the Standard Template Library (STL) [2] provide generalized containers and a facility to iterate over all of the objects within a container. However, the approach taken is different. Within the MFC the iteration mechanism is dependent upon the container, while in the STL there is a common iteration mechanism so that an algorithm can operate on each element of a container without knowledge of the container's type. The MFC containers support persistent storage, which is not a feature of the STL.

This paper presents a small example of a Windows® application using the STL containers in place of the corresponding MFC containers.

2 Example – Scribble.

2.1 Description of the Problem

The MFC Tutorial [3] includes a simple graphics application known as Scribble. The purpose of Scribble is to let the user draw a set of strokes with the mouse. The result is saved in a file (called a document) which can be opened and updated by adding additional strokes. (There is no method for deleting a stroke.) The user also has the option of specifying the thickness and color of the pen.

Scribble's data structure consists of one or more strokes. Each stroke is the record of the mouse position from the time when the user clicks on left mouse key to the time when the user releases the mouse button. In MFC approach, a new class `CStroke`, which is derived from class `CObject`, is defined. This contains a data member of `CArray<CPoint, CPoint>` with other data members to record and work on each stroke. The document class, `CScribbDoc`, is derived from `CDocument`. It contains a list of stokes using the MFC template class `CTypedPtrList< CObList, CStroke* >`.

Figure 1 illustrates the document data structure.

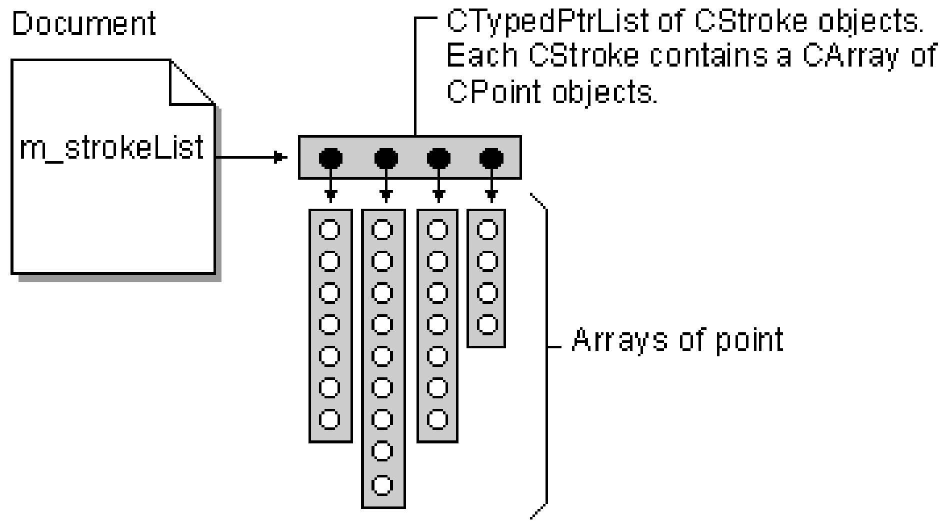
Document
m_strokeList
CTypedPtrList of CStroke objects. Each CStroke contains a CArray of CPoint objects.
Arrays of point

Figure 1 Scribble Document Structure

2.2 Type definitions

2.2.1 MFC Implementation

In the MFC implementation, the list of strokes is stored in the member `m_strokeList` which is defined as a `CTypedPtrList< CObject, CStroke* >`. The class `CStroke`, in turn, contains a `CArray< CPoint, CPoint >` to contain the array of points that constitute the stroke.

2.2.2 STL Implementation

The class `CStroke`List is defined to replace the `CTypedPtrList< CObject, CStroke* >` as follows:

```
class CStrokeList: public CObject,
public std::list<CStroke*>
{
public:
   CStrokeList(){}
   CStrokeList(const CStrokeList&);
   DECLARE_SERIAL(CStrokeList)

public:
   virtual
   void Serialize(CArchive& ar);
};
```

The `CArray<CPoint, CPoint>` in `CStroke` is replaced by a `std::list<CPoint>`.

2.3 Using the STL algorithms and iterators

2.3.1 Inserting points into a stroke

In the original MFC implementation, points were added to a stroke by the following statement:

```
m_pStrokeCur ->
   m_pointArray.Add(point);
```

In the STL implementation this becomes:

```
m_pStrokeCur –>
   m_pointArray.push_back(point);
```

2.3.2 Drawing all strokes

In the original MFC implementation, the list of strokes was traversed and each stroke drawn by the following code:

```
POSITION pos = strokeList.front();
while (pos != NULL)
{
   CStroke* pStroke =
      strokeList.GetNext(pos);
   pStroke->DrawStroke(pDC);
}
```

We make two changes. The first obvious change is to replace the MFC list iteration with the corresponding STL iteration. The result is as follows:

```
for (std::list<CStroke*>::iterator
      i = strokeList.begin();
      i != strokeList.end(); ++i)
   (*i)->DrawStroke(pDC);
```

The second change is to apply the `for_each` algorithm. Unfortunately, the `for_each` algorithm takes as its third argument a function of one argument, that argument being the type obtained by de-referencing the iterator. Specifically, we must convert the expression:

```
      (*i)->DrawStroke(pDC);
```

into a call to a function of one argument, where that argument is the dereferenced iterator. Stroustrup[4] shows how to do this using the binders and adapters. The function `mem_fun1` is a function of one parameter, the a pointer to member function that takes an arbitrary argument. The result of this function, is a function object that takes two arguments, the first of which is a pointer to a class, and the second is the same arbitrary second argument. Thus, the expression

```
      (*i)->DrawStroke(pDC);
```

may be replaced by

```
mem_fun1(&CStroke::DrawStroke)
   (*i, pDC);
```

We can now apply the `bind2nd` binder to convert this expression into a call to a function taking one agrument:

```
bind2nd(mem_fun1(
   &CStroke::DrawStroke), pDC)(*i);
```

The loop can now be replaced by a call to the `for_each` algorithm:

```
for_each(strokeList.begin(),
   strokeList.end(),
   bind2nd(mem_fun1(
      &CStroke::DrawStroke), pDC));
```

2.3.3 Drawing a stroke

The original code to draw a stroke was as follows:

```
pDC->MoveTo(m_pointArray[0]);
for (int i=1;
   i < m_pointArray.GetSize(); i++)
{
   pDC->LineTo(m_pointArray[i]);
}
```

We also make two changes. The first is to use the vector iterator as follows:

```
pDC->MoveTo(m_pointArray.begin());
for (vector<CPoint>::iterator i=
m_pointArray.begin();
   i != m_pointArray.end(); ++i)
   pDC->LineTo(*i);
```

Now the member function we are calling is not a member of the class pointed to by the objects in the container, but rather it is a member of the class `CDC`, which encapsulates the Windows® drawing context. The same `mem_fun1` adapter may be used as follows:

```
   mem_fun1(&CDC::LineTo)(pDC, *i);
```

Since `LineTo` is an overloaded function, we need to give the compiler some help resolving the ambiguity. This is done as follows:

```
typedef BOOL
(CDC::*ptr_to_fcn_of_POINT)(POINT);
ptr_to_fcn_of_POINT p =
   &CDC::LineTo;
   mem_fun1(p)(pDC, *i)
```

Since the loop variable is now the second argument, and the `pDC` is the first, we use `bind1st` to call the `for_each` algorithm as follows:

```
for_each(m_pointArray.begin(),
   m_pointArray.end(),
   bind1st(std::mem_fun1(p), pDC));
```

2.4 Serialization

2.4.1 Brief description of the MFC serialization

MFC provides a method for saving and retrieving a class to/from a file. The general approach is to write/read the raw bytes to the file preceded by some type identification. This is accomplished using the class `CObject` as an abstract base class, the virtual function `serialize`, and the class `CArchive`. `CArchive` encapsulates the file and provides overloaded insertion (<<) and extraction (>>) operators. These operators are provided for the built-in types, the standard Windows® types such as `WORD`, `DWORD`, and `POINT`, and pointers to `CObject`. The insertion operator for pointers to `CObject` writes type identification to the file, and then calls the `serialize` member function. The extraction operator verifies the type identification and then calls the `serialize` member function.

2.4.2 The serialize function

The generalize scheme for the `serialize` function is as follows:

```
void CMyClass::Serialize(
   CArchive& ar)
   {
      CObject::Serialize(ar);
      if (ar.IsStoring())
      {
         // insert the member
         // objects into ar
      }
      else
      {
         // extract the member
         // objects from ar
      }
   }
```

2.4.3 Serializing the stroke list and the stroke

The straightforward implementation of `serialize` for `CStroke`List and `CStroke` are as follows:

```
void CStrokeList::Serialize(
   CArchive& ar)
{
   CObject::Serialize(ar);

   if(ar.IsStoring())
   {
      ar << (WORD) size();
      for(list<CStroke*>::iterator
            it = begin();
            it != end(); ++it)
         ar << *it;
   }
   else
   {
      WORD s;
      ar >> s;    // get size
      clear();
      for (int i = 0; i !=s; i ++)
      {
         ar >> temp;
         push_back(temp);
      }
   }
}

void CStroke::Serialize(
   CArchive& ar)
{
   CObject::Serialize(ar);
   if (ar.IsStoring())
   {
      ar << (WORD)m_nPenWidth;
      ar <<
      (WORD) m_pointArray.size();
      for (vector<CPoint>::iterator
         i = m_pointArray.begin();
         i != m_pointArray.end();
         ++i)
         ar << *i;
   }
   else
   {
      WORD w;
      ar >> w;    // pen width
      m_nPenWidth = w;
      m_pointArray.clear();
      ar >> w;    // array size
      CPoint point;
      m_pointArray.reserve(w);
      for (int i = 0; i < w; ++i)
      {
         ar >> point;
         m_pointArray.push_back(
            point);
      }
   }
}
```

2.4.4 Archive iterators

Since `CArchive` is defined to work like `istream` and `ostream`, it is desirable to define a `CArchive_input_iterator` and a `CArchive_output_iterator` that will work like the `istream_iterator` and `ostream_iterator`. The loops can then be replaced by a call to the copy algorithm.

2.4.4.1 CArchive_output_iterator

Adapting the output_iterator to become the `CArchive_output_iterator` is very straightforward. Merely replace ostream with `CArchive`. The result is as follows:

```
template <class T>
class CArchive_output_iterator :
   public
std::iterator<std::output_iterator_
tag, void, void>
{
protected:
   CArchive* archive;
public:
   CArchive_output_iterator(
      CArchive& s) : archive(&s) {}
   CArchive_output_iterator<T>&
      operator=(const T& value)
   {
      *archive << value;
      return *this;
   }
   CArchive_output_iterator<T>&
      operator*() { return *this; }
   CArchive_output_iterator<T>&
      operator++() { return *this;
}
   CArchive_output_iterator<T>&
      operator++(int)
         { return *this; }
};
```

2.4.4.2 CArchive_input_iterator

The loop to read a stroke must terminate after all of the points for that stroke have been read. This is not at the end of the input stream. We will need to pass to the copy algorithm a `CArchive_input_iterator` that represents the current file position as the first argument, and a `CArchive_input_iterator` that represents the position at which the copy operaton should stop. If we were copying from a standard array, the copy call would look something like this:

```
copy(start, start+n, destination);
```

Therefore, we overload the `+` operator so that a `CArchive_input_iterator` plus an `int` will result in a `CArchive_input_iterator` object that can be used by the equality operator called by the copy algorithm to terminate the loop.

In `istream_iterator` the data value is extracted from the `istream` when the iterator object is first created and whenever the `++` operator is called. The dereferencing operator (`*`) then retrieves the saved value. Since we do not want to extract an object from the `CArchive` once the specified number of objects have been retrieved, we set a flag to indicate the need to retrieve a new object, and have the actual extraction performed by the `*` operator. The resulting implementation of `CArchive_input_iterator` is as follows:

```
template <class T>
class CArchive_input_iterator :
std::iterator<std::input_iterator_t
ag, T, ptrdiff_t>
{
friend bool operator==(const
CArchive_input_iterator<T>& x,
      const
CArchive_input_iterator<T>& y);
friend bool operator!=(const
CArchive_input_iterator<T>& x,
      const
CArchive_input_iterator<T> & y);
protected:
   CArchive* archive;
   T value;
   bool flag;     // True to
// indicate that value is defined
   int count;     // Count of the
// number of times ++ has been
// applied Or the target value for
// the number of advances
public:
   CArchive_input_iterator() :
      archive(0), flag(false),
         count(0) {}
   CArchive_input_iterator(
      CArchive& s) :
         archive(&s), count(0),
         flag(false) {}
   const T& operator*()
   {
      if (flag)
      {
         return value;
      }
      else
      {
         *archive >> value;
         flag = true;
         return value;
      }
   }
   CArchive_input_iterator<T>&
      operator++()
   {
      ++count;
      flag = false;
      return *this;
   }
   CArchive_input_iterator<T>
      operator++(int)
   {
      CArchive_input_iterator<T>
         tmp = *this;
      ++*this;
      return tmp;
   }
   CArchive_input_iterator<T>
      operator+(int delta)
   {
      CArchive_input_iterator<T>
      tmp = *this;
      tmp.count += delta;
      return tmp;
   }
};

template <class T>
inline bool operator==(const
   CArchive_input_iterator<T>& x,
   const
   CArchive_input_iterator<T>& y)
{
   return x.archive == y.archive &&
      x.count == y.count;
}

template <class T>
inline bool operator!=(const
   CArchive_input_iterator<T>& x,
   const
   CArchive_input_iterator<T>& y)
{ return !(x == y); }
```

2.4.5 Serializing CStrokeList and CStroke using Archive Iterators

The loops in `CStrokeList::serialize` and `CStroke::serialize` are then replaced by calls to the copy algorithm as follows:

Output of a `CStroke`List:

```
CArchive_output_iterator<CStroke*>
      oi(ar);
   copy(begin(), end(), oi);
```

Input of a `CStroke`List

```
CArchive_input_iterator<CStroke*>
      ii(ar);
   copy(ii, ii + s,
      back_inserter(*this));
```

Output of a `CStroke`

```
CArchive_output_iterator<CPoint>
      oi(ar);
   copy(m_pointArray.begin(),
      m_pointArray.end(), oi);
```

Input of a `CStroke`

```
CArchive_input_iterator<CPoint>
      ii(ar);
   copy(ii, ii + w,
      back_inserter(m_pointArray));
```

3 Conclusion

The STL containers can be easily adapted to be used in Windows® applications in place of the MFC collection classes. By adding the `CArchive` input and output iterators, the use of the standard algorithms can be applied to the serialization process.

The STL containers are more general than the MFC collection classes. By separating the iterators from the containers themselves, and by providing a common interface for all containers, the STL container choice is independent of the algorithm. The Scribble example uses a linked list of arrays to represent the strokes. This was replicated in the STL version. The STL version could easily be changed to use an array (vector) of lists, or a list of lists, or an array of arrays.

We first developed this example using Microsoft's Visual C++ version 5.0. We experienced numerous difficulties requiring us to define a class, MYCPoint to extend the CPoint class and developing modified versions of the function binders and adapters. Microsoft's Visual C++ version 6.0 does not require these work-arounds.

4 References

[1] Shepherd, George and Wingo, Scot. *MFC Internals*. Addison-Wesley, 1996.
[2] Stepanov, A. A. and Lee, M. *The Standard Template Library*. Technical Report HPL-94-34, Hewlet-Packard Laboratories, April 1994.
[3] The Microsoft Developer Studio Help Files and Example Files.
[4] Stroustrup, Bjarne. *The C++ Programming Language Third Edition*. Addison-Wesley, 1997.

Paul Wolfgang is a Staff Engineer at Boeing and an Adjunct Professor at Temple Univesity. He can be reached at Paul.Wolfgang@Boeing.com or at wolfgang@falcon.cis.temple.edu.

Yang Song is a graduate student at Temple University. She can be reached at yangsong@astro.ocis.temple.edu.