Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 367–374.

C++ Standard Template Library by Ranges[*]

Norbert Pataki

Dept. of Programming Languages and Compilers,
Fac. of Informatics, Eötvös Loránd University, Budapest
e-mail: patakino@elte.hu

Abstract

The *C++ Standard Template Library (STL)* is the most popular library based on the generic programming paradigm. *STL* is widely-used, because the library is part of the *C++* Standard. It consists of many useful generic data structures and generic algorithms, that are fairly irrespective of the used container. Iterators bridge the gap between containers and algorithms. As a result of this layout the complexity of the library is greatly reduced and we can extend the library with new containers and algorithms simultaneously.

Iterators introduced many different problems. First, hard to write hand-crafted iterators. Some people argue for the range types which can be a superior abstraction. Ranges can be aggregated in a better way. They have better checking abilities.

In this paper we present an overview about the differences between ranges and iterators. We examine a set of algorithms with ranges. Creating new range types is also considered.

Keywords: STL, iterators, ranges

MSC: 68N19 Other programming techniques

1. Introduction

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach [7, 16]. In this way some containers defined as class templates and many algorithms can be implemented as function templates. Furthermore, algorithms are implemented in a container-independent way, so one can use them with different containers [20]. C++ STL is widely-used because it is a very useful, standard C++ library that contains many useful containers (like list, vector, map, etc.), many algorithms (like sort, find, count, etc.) among other utilities.

[*]Supported by TÁMOP-4.2.1/B-09/1/KMR-2010-0003.

The STL was designed to be extensible. We can add new containers that can work together with existing algorithms. On the other hand, we can extend the set of algorithms with a new one that can be work together with existing containers. Iterators bridge the gap between containers and algorithms [5]. STL also includes some adaptor types which transform standard elements of the library for a different functionality [1, 14].

However, the usage of C++ STL does not mean bugless or error-free code [9]. Contrarily, incorrect application of the library may introduce new types of problems [19]. One of the root causes is that algorithms take two iterators as the input interval. One range is passed by two objects. Parameters are handled independently and the connections between them are analyzed in a very difficult way.

This paper is organized as follows. Some known iterator-related problems are described in section 2. The notation of *ranges* is introduced and advantages of ranges are present in section 3. A range-based STL implementation can be found in section 4. After all, we conclude our results and give some directions about the future work in section 5.

2. General problems

Some problems come from the STL's generic approach. This approach prepossesses the software metrics [19]. In this section we present some typical STL-related mistakes.

One the most trivial problem is according to the algorithms' precondition [18]. For example, some algorithms need sorted input and they take advantage of sortedness, but input is not checked neighter at compilation time nor at runtime. Algorithms like `binary_search`, `lower_bound`, `equal_range` result in undefined behaviour when they called on an unsorted range. However, it is very difficult to check the input's sortedness without algorithm modification [13].

Another typical problem is related to the iterator invalidation [10]. This appears when a contiguous-memory container (for example, `vector`) reallocates itself when its capacity is full. We have constructed iterators before the reallocation, and use one of these iterators after the reallocation results in an undefined behaviour. Let us consider the following code snippet:

```
std::vector<int> v;
v.push_back(1);
std::vector<int>::iterator it = v.begin();
// vector's capacity changes:
for( int i = 0; i < 100; ++i)
  v.push_back(i);

int o = *it;
```

The main weakness of iterators that every algorithms take two iterators as one interval [4]. For instance, here are some declarations from the STL:

```
template <class InputIterator, class UnaryFunction>
UnaryFunction for_each( InputIterator first,
                        InputIterator last,
                        UnaryFunction f);

template <class RandomAccessIterator>
void sort( RandomAccessIterator first,
           RandomAccessIterator last );

template <class ForwardIterator, class LessThanComparable>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const LessThanComparable& value);
```

Another typical iterator related bug appears when one misuses output iterators:

```
std::vector<int> v;
v.push_back( 3 );

std::list<int> l;
std::copy( v.begin(), v.end(), l.begin() );
```

The elements of the vector should be copied to the list in the previous code, but it causes runtime error. `copy` assumes that it can copy to the output, there is allocated memory, but in an empty list no one allocates space for the elements. It does not causes problem, if the list's size is not less than vector's size. `back_inserter` and `front_inserter` iterators can be used for to force `push_back` and `push_front` method, respectively.

In this section we described a set of problems which comes from the generic approach of the STL.

3. Ranges

Range is the abstraction over iterators. Ranges are introduced in the D programming language.

The simplest range is the notation of input ranges. Their interface can be defined in the following way:

```
template <class T>
class InputRange
{
  void pop_front();
  T& front();
  bool empty() const;
};
```

We can express the very same functionality with two input iterators and one input range in a different syntax. But as range is a type, one can check many properties. For instance, sorted range's constructor can test if the range is checked and it ensures the correct behaviour without algorithm modification.

A typical implementation for contiguous-memory range is the following:

```
template <class T>
class ContRange
{
  T *first, *last;
public:
  bool empty() const
  {
    return first == last;
  }

  void pop_back()
  {
    ++first;
  }

  T& front()
  {
    return *first;
  }
};
```

This way we can easily check range-related properties. For example, we can make sure if front is called on empty range:

```
  T& front()
  {
    assert( !empty() );
    return *first;
  }
```

Call of algorithms with mixture of ranges is much more easier by ranges than by iterators. For instance, let us consider the following code fragment:

```
template <class R1, class R2>
R2 copy( R1 r1, R2 r2 );


std::vector<float> v;
std::list<int> s;
std::deque<double> d;


std::copy( chain( v, s ), d );
```

Let us consider a copy algorithm which copies the first to the second range and returns the untouched portion of r2, and we have a function called `chain` which returns a range which is the concatenation of the arguments. To describe this scenario in an STL-way is very difficult.

Iterator hierarchy is grouping of iterator based on iterators' capabilities. Iterator hierarchy cannot be appeared by language constructs [23], but it can appear by range types and inheritance.

One can easily implement an adaptor type, that reverses the traversal:

```
template<class R>
class Retro
{
  R r;
public:
  bool empty() const
  {
    return r.empty();
  }

  void pop_front()
  {
    return r.pop_back();
  }

  void pop_back()
  {
     return r.pop_front();
  }

  E<R>::Type& front()
  {
    return r.back();
  }

  E<R>::Type& back()
  {
    return r.front();
  }

};
```

Iterators' advantage is the reverse-compatibility with the built-in arrays because of the pointer-arithmetic. Every standard algorithms work with arrays too.

In this section we present ranges' basic idea. We argue for ranges, because they support range-related properties that iterators cannot do among other advantages.

4. C++ Standard Template Library by Ranges

In this section we overview a potential implementation of the STL algorithms in which ranges can be used instead of iterators.

For instance, we can write our `for_each` algorithm in the following:

```
template <class Range, class Fun>
Fun for_each( Range r, Fun f )
{
  while ( !r.empty() )
  {
    f( r.front() );
    r.pop_front();
  }
}
```

This implementation is quite straightforward, because it takes two parameters: the range and a functor object to be called on the elements in the range.

There are some pitfalls with this implementation. In this implementation `list` and `deque` containers can be work as ranges, but `vector` is not allowed.

Every STL algorithms can be defined in this way too. It does not constraint the genericity of algorithms. The library is kept generic.

Usage of standard stream iterators are lengthy, uncomprehensible, unmaintenable. With the help of ranges these applications can be easier too. Let us consider the following two different implementations of the similar code which copies the standard input to standard output:

```
std::copy( std::istream_iterator<char>( std::cin ),
           std::istream_iterator<char>(),
           std::ostream_iterator<char>( std::cout ) );

copy ( istream_range<char>( std::cin ),
       ostream_range<char>( std::cout ) );
```

In this section we argue for ranges by present an STL implementation by ranges. Many algorithms can be called easier in this way.

5. Conclusion and Future Work

STL is widely-used generic library in which iterators play an important role. The STL itself introduces new potential errors based on the iterators. Unfortunately, these errors cannot handle in a non-intrusive way.

The ranges are an abstraction over iterators. The most important feature of ranges that properties can be checked in the range type. An other feature is the mixture of ranges. STL algorithms can be rewritten by ranges. Usage of algorithms can be easier especially when one deals with stream iterators.

Abstraction penalty occurs when ranges are used. Measuring the overhead of the runtime checks is necessary. In this paper we do not deal with *output ranges*. An elegant interface is necessary when output ranges are used.

References

[1] Alexandrescu, A.: "Modern C++ Design" Addison-Wesley (2001)

[2] Austern, M. H.: "Generic Programming and the STL: Using and Extending the C++ Standard Template Library", Addison-Wesley (1998)

[3] Austern, M. H., Towle, R. A., Stepanov, A. A.: *Range partition adaptors: a mechanism for parallelizing STL*, in ACM SIGAPP Applied Computing Review 1996 **4(1)**, pp. 5–6,

[4] Baus, C., Becker, T.: *Custom Iterators for the STL*, in Proc. of First Workshop on C++ Template Programming.

[5] Becker, T.: *STL & generic programming: writing your own iterators*, C/C++ Users Journal 2001 **19(8)**, pp. 51–57.

[6] Biczó, M., Pócza K., Forgács, I., Porkoláb, Z.: *A New Concept of Effective Regression Test Generation in a C++ Specific Environment*, Acta Cybernetica 2008 **18(3)**, pp. 408–501.

[7] Czarnecki K., Eisenecker, U. W.: "Generative Programming: Methods, Tools and Applications," Addison-Wesley (2000)

[8] Das D., Valluri, M., Wong, M., Cambly, C.: *Speeding up STL Set/Map Usage in C++ Applications*, LNCS **5119** (2008), pp. 314-321.

[9] Dévai, G., Pataki, N.: *Towards verified usage of the C++ Standard Template Library*, In Proc. of The 10th Symposium on Programming Languages and Software Tools (SPLST) 2007, pp. 360–371.

[10] Dévai, G., Pataki, N.: *A tool for formally specifying the C++ Standard Template Library*, In Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica **31**, pp. 147–166

[11] Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., Lumsdaine, A.: *Concepts: linguistic support for generic programming in C++*, in Proc. of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA 2006), pp. 291–310.

[12] Gregor, D., Schupp, S.: *Stllint: lifting static checking from languages to libraries*, Software - Practice & Experience, 2006 **36(3)**, pp. 225–254

[13] Järvi, J., Gregor, D., Willcock, J., Lumsdaine, A., Siek, J.: *Algorithm specialization in generic programming: challenges of constrained generics in C++*, in Proc. of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2006), pp. 272–282.

[14] Matsuda, M., Sato, M., Ishikawa, Y.: *Parallel Array Class Implementation Using C++ STL Adaptors*, In Proc. of the Scientific Computing in Object-Oriented Parallel Environments, LNCS **1343**, pp. 113-120.

[15] Meyers, S.: “Effective STL - 50 Specific Ways to Improve Your Use of the Standard Template Library,” Addison-Wesley(2001).

[16] Musser, D. R., Stepanov, A. A.: *Generic Programming*, in Proc. of the International Symposium ISSAC’88 on Symbolic and Algebraic Computation, LNCS **358** 1988, pp. 13–25.

[17] Pataki, N., Porkoláb, Z., Istenes, Z.: *Towards Soundness Examination of the C++ Standard Template Library*, In Proc. of Electronic Computers and Informatics, ECI 2006, pp. 186–191.

[18] Pataki, N., Szűgyi, Z., Dévai, G.: *C++ Standard Template Library in a Safer Way* , In Proc. of Workshop on Generative Technologies 2010 (WGT 2010), pp. 46-55.

[19] Porkoláb, Z., Sipos, Á., Pataki, N.: *Inconsistencies of Metrics in C++ Standard Template Library*, In Proc. of 11th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering QAOOSE Workshop, ECOOP 2007, Berlin, pp. 2–6

[20] Stroustrup, B.: “The C++ Programming Language”, Addison-Wesley(1999)

[21] Szűgyi, Z., Sipos, Á, Porkoláb, Z: *Towards the Modularization of C++ Concept Maps*, in Proc. of Workshop on Generative Programming (WGT 2008), pp. 33–43.

[22] Zolman, L.:*An STL message decryptor for visual C++*, In C/C++ Users Journal, 2001 **19(7)**, pp. 24–30.

[23] Zólyomi, I., Porkoláb, Z.: *Towards a General Template Introspection Library*, in Proc. of Generative Programming and Component Engineering: Third International Conference (GPCE 2004), LNCS **3286**, pp. 266-282.

Norbert Pataki
Pázmány Péter sétány 1/c., H-1117 Budapest, Hungary