Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 264 (5) (2011) 71–83
www.elsevier.com/locate/entcs

Measuring the Overhead of C++ Standard Template Library Safe Variants

Norbert Pataki[1] Zalán Szűgyi[2] Gergely Dévai[3]

Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary

Abstract

The C++ Standard Template Library is a widely-used library that is based on the generic programming paradigm. The usage of this library does not warrant bug-free programs. Furthermore, many new errors may arise from the inaccurate use of the generic programming paradigm, like dereferencing invalid iterators or misunderstanding remove-like algorithms.
Most of the STL algorithms have preconditions which are checked neither at compilation time nor at runtime. Violation of such a precondition results in undefined behaviour.
In this paper we propose solutions for a subset of these problems. The techniques we describe help programmers use generic algorithms on sorted intervals in a safer way. We present a new iterator adaptor type and tag as well as safe containers which keep track their iterators' validness. We measure the runtime overhead of these extensions.

Keywords: C++ STL, iterator adaptor, container

1 Introduction

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach. In this way containers are 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 [21]. C++ STL is widely-used because it is a very handy, standard C++ library that contains beneficial containers (like list, vector, map, etc.), a lot of algorithms (like sort, find, count, etc.) among other utilities.

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.

[1] Email: patakino@elte.hu
[2] Email: lupin@ludens.elte.hu
[3] Email: deva@elte.hu

doi:10.1016/j.entcs.2011.06.005

Iterators bridge the gap between containers and algorithms [5]. The expression problem [24] is solved with this approach. STL also includes adaptor types which transform standard elements of the library for a different functionality [1].

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 [20].

One of the problems is, that the error diagnostics are usually complex, and very hard to figure out the cause of a program error [25,26]. Violation of the requirement of strict weak ordering in comparison functors also means strange bugs [11]. This results in incosistent containers at runtime. A different kind of stickler is that if we have an iterator object that pointed to an element in container, but the element is erased or the container's memory allocation has been changed, then the iterator becomes *invalid*. Another common mistake is according to removing algorithms. The algorithms are container-independent, hence they do not know how to erase elements from a container just put them to the end of the container, and we need to invoke a specific erase member function to remove the elements phisically. Since, for example the `remove` algorithm cannot actually remove any element from a container [15].

Most of the properties are checked at compilation time. For example, the code does not compile if one uses sort algorithm with the standard list container, because the list's iterators do not offer random accessibility [13]. Other properties are checked at runtime. For example, the standard vector container offers an `at` method which tests if the index is valid and it raises an exception otherwise [17].

Unfortunately, there is still a large number of some properties are tested neither at compilation-time nor at run-time. Observance of these properties is in the charge of programmers. Let us consider the following code snippet:

```
std::vector<int> v;
int x;
//...
std::vector<int>::iterator i =
  std::lower_bound(v.begin(), v.end(), x);
```

The purpose of `lower_bound` is to find an element in an ordered range. It is a version of binary search, hence it has logarithmic complexity. We assume that we can find an element in a vector in logarithmic time because of the sortedness of the vector. However, it causes undefined result, if the vector is not ordered [18]. Implementations of these algorithms do not test if the range is sorted appropriately. Many STL algorithms expect ordered range: `equal_range`, `binary_search`, `set_difference`, etc.

In addition, the following algorithms are typically used with ordered ranges, though they do not require them: `unique` and `unique_copy`.

Furthermore, sortedness of container is not enough. We must make sure that the same sorting function object is used for sorting and for searching. The following code snippet also results in undetermined behaviour:

```
std::vector<int> v;
int x;
//...
std::sort(v.begin(), v.end());
std::vector<int>::iterator i =
  std::lower_bound(v.begin(), v.end(), x, std::greater<int>());
```

Other typical STL-related mistakes are related to iterator invalidation. This problem occurs when a container that is being processed using an iterator has its shape changed during the process, for example anything that causes a vector's reallocation (increase in the result of `capacity()`) will invalidate all iterators. When one use an invalid iterator also causes an undefined result [10]. Let us consider the following code:

```
std::vector<int> v;

//...

std::vector<int>::iterator i = v.begin();

// ...
// vector's capacity has been changed...
std::cout << *i;
```

When `*i` is referred, it causes undefined result because `i` has become invalid.

Some useful software like STLlint [12] can help us finding STL-related bugs as a compile-time tool, but these have shortcomings and are not extensible.

In this paper we present an extension of the C++ STL, that is able to check iterators' validness at runtime. We measure the overhead of this approach. We also describe a technique that can use generic algorithms on sorted intervals in a safer way.

This paper is organized as follows. In section 2 we provide a new iterator adaptor that is able to extend the standard iterators with range checking. After that, we analyze its properties and present utilities for a more comfortable way in section 3 that makes the adaptor more comfortable to use. In section 4 we argue for a new implementation technique of the containers: Our implementation is safer with a minimal runtime overhead. We have measured the runtime of our extensions, and present the results in section 5. We conclude our results and present some directions about our future work in section 6.

2 New iterator type

First, we work out a new iterator adaptor type, that is able to check the sortedness of a range. This type is a template that inherits from a usual iterator type [4], but its constructor takes three iterators: the second and third argument describes the interval which must be checked, the first one stands for where it points to.

Let us consider the following code:

```
template <class Iterator, class Comp>
bool is_sorted(Iterator first, Iterator last, Comp c)
{
     for(CIt i = first; i!=last-1; ++i)
        if(!c(*i, *(i+1)))
           return false;
     return true;
}

template <class Container,
          class Compare = std::less<typename Container::value_type> >
class range_check_const_iterator:public Container::const_iterator
{
  typedef typename Container::const_iterator CIt;

  void check(CIt first, CIt last, Compare c)
  {
     if(!is_sorted(first, last, c))
        throw not_sorted();
  }

public:
  range_check_const_iterator(CIt curr, CIt first, CIt last)
    :Container::const_iterator(curr)
  {
    check(first, last);
  }
};
```

This is an adaptor type, it transforms the original const_iterator type into a range checked iterator type. Its constructor checks the sortedness, and throws an exception if fails.

Using iterators in this way enhances the power of the STL insofar as algorithms can now be applied in a safer way.

3 Conveniences

One can use the previous iterator adaptor easily. Let us consider the following code:

```
std::vector<int> v;
int x;
//...
std::vector<int>::iterator i =
  std::lower_bound(
```

```
    range_check_const_iterator<std::vector<int> >
        (v.begin(), v.begin(), v.end()),
    range_check_const_iterator<std::vector<int> >
        (v.end(), v.end(), v.end()),
    x);
```

The first argument of `lower_bound` checks the interval's sortedness. The second argument of algorithm checks an empty interval because we do not want to perform the sortedness checking twice in the same interval. The `lower_bound` searches the value `x` in the range `v.begin(), v.end()`. Due to range checking, this call takes linear time instead of the original logarithmic time. Since this solution makes a single call of an algorithm very difficult, we provide function templates that able to create safe iterators. See the code below:

```
template <class Container>
range_check_const_iterator<Container>
    iterator_check_begin(const Container& c)
{
  return range_check_const_iterator<Container>(c.begin(),
                                               c.begin(),
                                               c.end());
}

template <class Container>
range_check_const_iterator<Container>
    iterator_nocheck_begin(const Container& c)
{
  return range_check_const_iterator<Container>(c.begin(),
                                               c.begin(),
                                               c.begin());
}
```

Similar function templates are necessary for end iterators and arbitrary function objects too. These techniques is quite typical in STL: different function templates for standard behaviour and for arbitrary one. Developing function templates for parameter deduction is very common in the STL. However, there is a problem in this case: when we have an iterator as an argument of `lower_bound`, we cannot use these template functions.

In the STL algorithms can be overloaded on *iterator tags*, for example `distance` and `advance` can take advantage of different iterators, it runs in constant time with random access iterators, but takes linear time when the arguments are bidirectional iterators.

First, we introduce a new iterator tag, to deal with mentioned technique and save the category of base iterator:

```
struct checked_iterator_tag {};
```

```
template <class Container,
          class Compare = std::less<typename Container::value_type> >
class range_check_const_iterator:public Container::const_iterator
{
public:
  typedef iterator_category checked_iterator_tag;

  typedef typename std::iterator_traits
                      <Container::const_iterator>::iterator_category
                    base_category;
  // ...
};
```

Second, we try to adopt the `lower_bound` to our new iterator category. STL references (e.g. [2]) describe the complexity of `lower_bound` depends on the argument iterators' category and also can take advantage of the random access iterators. Hence, the `lower_bound` is overloaded on iterator tags. Therefore, we can create our `lower_bound` for our `checked_iterator_tag` in the following way:

```
template <class Iterator, class T, class Comp>
Iterator lower_bound(Iterator first, Iterator last,
                             const T& a, Comp c,  checked_iterator_tag)
{
  if (!is_sorted(first, last, c))
    throw not_sorted();
  else
    return lower_bound(first, last, a, c,
                       Iterator::base_category());
}
```

Constructor of safe iterator does not check the sortedness anymore, because the overloaded algorithm is able to check it and call the original algorithm. Some other function templates should be overloaded to this category according to the STL implementation (like `distance` and `advance`) in a similar way.

4 Overcome of invalid iterators

In this section we present a technique that can be used to avoid the undefined behaviour of invalid iterators' usage. The technique is adaptable for all standard and nonstandard containers. Different containers invalidate iterators in different ways, however, this technique can be transformed to `list`, `deque` or other third party defined containers too. In a more sophisticated solution the invalidation behaviour should be parametrized. We present the technique as an extension of STL's vector template.

In our implementation the vector objects keep tracks their iterators which have a member to describe if the iterator is valid. When the vector reallocates itself, it

sends a message to its iterators that they become invalid. If one accesses an element via an invalid iterator, then an exception is raised.

Let us consider the following code snippet:

```
template <class T, class Alloc = std::alloc, bool debug=false>
class vector
{
  T* p;
  int cap, s;
  std::list<iterator*> iterators;

public:

  struct iterator: std::iterator<std::random_access_iterator_tag, T>
  {
  private:
     bool isvalid;
     T* curr;
  public:
     iterator(T* c):curr(c), isvalid(true) {}

     T& operator*()
     {
        if (!isdebug)
          return *curr;

        if(isvalid)
          return *curr;
        else
           throw invalid_iterator();
     }

     iterator& operator++()
     {
       ++curr;
       return *this;
     }

     iterator operator++(int)
     {
       iterator tmp(*this);
       ++curr;
       return tmp;
     }
```

```
    // ...

  };

private:
  void realloc()
  {
    cap*=2;
    T* t = new T[cap];
    std::copy(p, p+s, t);
    delete [] p;
    p = t;
  }

  void invalid()
  {
    for(typename std::list<iterator*>::iterator it = iterators.begin();
        it != iterators.end();
        ++it)
    {
      (*it)->isvalid = false;
    }
  }

public:
  vector():cap(1), s(0)
  {
    p = new T[cap];
  }

  vector()
  {
    delete [] p;
  }

  void push_back(const T& a)
  {
    if (s<cap)
      p[s++] = a;
    else
    {
      realloc();
      invalid();
      push_back(a);
    }
```

```
  }

  iterator begin()
  {
    iterator i(p);
    iterators.push_back(&i);
    return i;
  }

  iterator end()
  {
    iterator i(p+s);
    iterators.push_back(&i);
    return i;
  }

  // ...

};
```

Of course, the testing can depend on a preprocessor macro or something else. Legacy STL-based codes can be easily transformed to use this vector container with extra checks. Just an extra parameter should be passed to the vector type. However, there is no trivial assignment and copy between an untested and tested vector container, but a special template copy constructor and assignment operator can be added.

Naturally, we can create a specialization for the safe and unsafe versions. This makes our implementation faster, but now we just proof our concept.

Similarly, we can create a safe iterator implementation that is able to pursue the vector's pointer. In this case, an exception is thrown when an iterator is referred which point at an erased element.

It is also should be considered if invalidation includes the end iterators. Also causes runtime problems if end iterators are dereferenced. It can be handled in an orthogonal way.

If we want to invalidate all iterators that points to an erased element, we should write the `erase` method in the following way:

```
template <class T, class Alloc = std::alloc, bool debug = false>
class vector
{
  // like the previous code ...
public:
  void erase(iterator i)
  {
     if (debug && !i.isvalid)
```

```
        throw invalid_iterator();

      std::copy(i.curr+1, p+s, i.curr);
      for(typename std::list<iterator*>::iterator it = iterators.begin();
          it != iterators.end();
          ++it)
      {
         if(it->curr==i.curr)
            it->isvalid = false;
      }
  }
};
```

5 Measuring the overhead

In this section we evaluate the runtime overhead of our vector implementation presented in section 4. We work out some variants to manage the contained iterators.

Some directions can be mentioned to deal with the contained iterators too. It is an interesting question to how to deal with them. In the destructor of `vector::iterator` we can execute code to manage them. Hence, the iterator objects should hold a pointer which points to the vector object on which the `begin()` or `end()` method is called. This methods can be extended to pass a pointer to `this` and iterator's constructor can be modified to take it, as well.

The first variant is the simplest one, nothing is done with contained iterators, all constructed iterators can be found in the list. The disadvantage of this approach is that the number of contained iterators increases, and too many unnecessary invalidation is executed at reallocation. This variant can be seen in the previous section. This version is marked with 1st variant on the tables.

Another approach is that the list is maintained when the iterators are destructed. It is easy to erase the iterator from the list in the destructor of `vector::iterator`. The advantage of this approach is that in the list only the existing iterators can be found. This is marked with 2nd variant in the comparisons.

We developed some various general-purpose iterator-intense codes to measure the overhead of our implementation. These codes contain significantly more construction of iterators than usual applications. Hence, efficacy of our implementation is better in ordinary usage. We measured the runtime of code with our two variants and with SGI STL implementation. We always compared the speed of our solutions to the SGI STL's implementation. Thus the running time on `std::vector` is the 100%.

In our first test case we created a lot of vectors with few elements and performed a `sort` algorithm of STL.

std::vector	100%
1st variant	114%
2nd variant	299%

The second test case created a lot of vectors with moderate amount of elements and performed different iterator operations (such as dereference, incrementation, decrementation etc.).

std::vector	100%
1st variant	72%
2nd variant	84%

In the third test case we created one huge vector and performed the same iterator operatons as previous test case.

std::vector	100%
1st variant	608%
2nd variant	726%

In the last test case we increased the size of vector step-by-step to see how the reallocation affects the performance and applied the `accumulate` algorithm of STL on this vector.

std::vector	100%
1st variant	106%
2nd variant	110%

In those applications where the iterator operations are dominating our solution is six or seven times slower (see. third test case). However, in general usage of vectors, where the amount of vector operations is balanced with the amount of iterator operations, our solution is just slightly slower than STL's implementation. In the second test case our implementations are a bit faster, presumably our reallocation strategy suits better for this test. It also can be seen, that the first variant has better efficacy in all test cases, maintenance of contained iterators is not worth while in this way.

6 Conclusion and future work

In this paper we presented some extensions for the C++ STL. These extensions can be used for avoid some runtime problems, including invalid iterators and violated preconditions. In the original implementations these problems cause undefined be-

haviour but with our one can deal with exceptions. We took advantage of many STL-based techniques for make our solution more comfortable. We presented actual extensions, reverse compatibility works with the original code. Furthermore, we measured the efficacy of our implementations.

In this paper we did not deal with multithreaded programs, so we did not analyze our implementation in a multithreaded environment. We are going to consider, examine and prepare our implementation for threaded programs [3].

In the future we consider how the invalidation can be passed as a trait. It would be quite useful if one can parametrize the strategy of invalidation and pass it to containers. Maintenance of contained iterators is also can be more sophisiticated, and it also should be a trait. Another direction can be mentioned according to the type system. It would be elegant to support checks with the type system of C++ to caching the results and avoid unnecessary runtime checks.

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] Pirkelbauer, P., Parent, S., Marcus, M., Stroustrup, B.: *Runtime Concepts for the C++ Standard Template Library*, In Proc. of the 2008 ACM symposium on Applied computing, pp. 171–177.

[20] 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.

[21] Stroustrup, B.: "The C++ Programming Language", Addison-Wesley(1999).

[22] 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.

[23] Szűgyi, Z., Sinkovics, Á., Pataki, N., Porkoláb, Z.: *C++ Metastring Library and its Applications*, In Proc. of Generative and Transformational Techniques in Software Engineering 2009, LNCS **6491**, pp. 467–486.

[24] Torgersen, M.: *The Expression Problem Revisited – Four New Solutions Using Generics*, in Proc. of European Conference on Object-Oriented Programming (ECOOP) 2004, LNCS **3086**, pp. 123–143.

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

[26] 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.