Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 911–914

ISBN 978-83-60810-22-4

Extension of Iterator Traits in the C++ Standard Template Library

Norbert Pataki, Zoltán Porkoláb
Department of Programming Languages and Compilers,
Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
Email: {patakino, gsd}@elte.hu

***Abstract*—The C++ Standard Template Library is the flagship example for libraries based on the generic programming paradigm. The usage of this library is intended to minimize classical C/C++ error, but does not warrant bug-free programs. Furthermore, many new kinds of errors may arise from the inaccurate use of the generic programming paradigm, like dereferencing invalid iterators or misunderstanding remove-like algorithms.**

In this paper we present typical scenarios, that can cause runtime problems. We emit warnings while these constructs are used without any modification in the compiler. We argue for an extension of the STL's iterator traits in order to emit these warnings. We also present a general approach to emit "customized" warnings. We support the so-called believe-me marks to disable warnings.

I. Introduction

THE *C++ Standard Template Library* (STL) was developed by *generic programming* approach [2]. 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 [15]. C++ STL is widely-used because it is a very handy, standard 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 the existing algorithms. On the other hand, we can extend the set of algorithms with a new one that can work together with the existing containers. Iterators bridge the gap between containers and algorithms [3]. The expression problem [16] 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 guarantee bugfree or error-free code [5]. Contrarily, incorrect application of the library may introduce new types of problems [14].

One of the problems is that the error diagnostics are usually complex, and very hard to figure out the root cause of a program error [17], [18]. Violating requirement of special preconditions (e.g. sorted ranges) is not checked, but results in runtime bugs [6]. A different kind of stickler is that if we have an iterator object that pointed to an element in a container, but the element is erased or the container's memory allocation has been changed, then the iterator becomes *invalid*. Further reference of invalid iterators causes undefined behaviour [13].

Another common mistake is related to removing algorithms. The algorithms are container-independent, hence they do not know how to erase elements from a container, just relocate them to a specific part of the container, and we need to invoke a specific erase member function to remove the elements phisically. Therefore, for example the `remove` algorithm does not actually remove any element from a container [10].

Some of the properties are checked at compilation time. For example, the code does not compile if one uses the `sort` algorithm on a standard list container, because the list's iterators do not offer random accessibility [8]. 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 [12].

Unfortunately, there are still a large number of properties are tested neither at compilation-time nor at run-time. Observance of these properties is in the charge of the programmers. On the other hand, type systems can provide a high degree of safety at low operational costs. As part of the compiler, they discover many semantic errors very efficiently.

Certain containers have member functions with the same names as STL algorithms. This phenomenon has many different reasons, for instance, efficiency, safety, or avoidance of compilation errors. For example, as mentioned, list's iterators cannot be passed to `sort` algorithm, hence code cannot be compiled. To overcome this problem list has a member function called `sort`. List also provides `unique` method. In these cases, although the code compiles, the calls of member functions are preferred to the usage of generic algorithms.

In this paper we argue for an approach that generates warning when the STL is used in an improper way or a better approach is available in certain cases. For example we want to warn the programmer if the `copy` or `transform` algorithm is used without inserter iterators. We argue for an extension of STL's trait type of iterators insted of compiler modification. Algorithms can be overloaded based on this extension and warnings can be trigerred in the overloaded versions.

This paper is organized as follows. In section II we present some motivating examples, that can be compiled, but at runtime they can cause problems. Then, in section III we present new properties that should be added to the STL's iterator traits. In section IV we present an approach to generate "customized" warnings at compilation time. In section V we argue for overloading algorithms on the new traits. In section VI the so-called *believe-me marks* are introduced to disable our specific warnings. Finally, this paper concludes in section VII.

II. Motivation

STL's `copy` and `transform` algorithm can be used to copy an input range of objects into a target range. These algorithms neither allocate memory space nor call any specific inserter method while coping elements. They assume that the target has enough, properly allocated elements where they can copy elements with `operator=`. Inserter iterators can enforce to use `push_back`, `push_front` or `insert` method of containers. But these algorithms cannot copy elements into an empty list, for instance. They do not know how to insert elements into the empty container. The following code snippet can be compiled, but it results in an undefined behaviour:

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

std::copy( vi.begin(),
           vi.end(),
           li.begin() );
```

In our opinion in this case a warning message should be emitted to the programmer, that this construct can be problematic.

III. New traits

Iterators are fundemental elements of the STL. They make connection between containers and algorithms. Iterators iterates through the containers or streams. They are the generalization of pointers, thus pointers also can be used in place of iterators.

Iterators have associated types. An iterator type, for instance, has an associated value type: the type of object that the iterator points to. It also has an associated type to describe the type of difference-based values. Generic algorithms often need to have access to these associated types; an algorithm that takes a pair of iterators, for example, might need to declare a temporary variable whose type is the iterators' value type. The class `iterator_traits` is a mechanism that allows such declarations. For every iterator type, a corresponding specialization of `iterator_traits` class template shall exist or default implementation works (see below). Another reason also can be mentioned for the usage of traits. It can be used for implementing generic functions as efficient as possible. For example, `distance` or `advance` can fully take advantage of the iterator capabilities, and can run at constant time when random access iterators the taken and run at linear time otherwise.

At this point we extend `iterator_traits` in order to overload STL algorithms on new traits and generate warning in some of them. First, we write two new types according to copying strategy.

```
class __inserting_iterator_tag {};
class __non_inserting_iterator_tag {};
```

The default `iterator_traits` is extended in the following way:

```
template <class T>
struct iterator_traits
{
  typedef typename T::iterator_category
    iterator_category;
  typedef typename T::value_type
    value_type;
  typedef typename T::difference_type
    difference_type;
  typedef typename T::pointer
    pointer;
  typedef typename T::reference
    reference;
  typedef
    __non_inserting_iterator_tag
      inserter;
};
```

We added one more attribute to default `iterator_traits` which is the copying strategy attribute called `inserter`. The `inserter` is a type alias to either `__inserting_iterator_tag` or `__non_inserting_iterator_tag`.

More traits can be mentioned, too. For instance, `find` and `count` algorithms are suboptimal if it is called on an *associative* container, because the algorithms cannot take advantage of sortedness. Hence, an attribute can be described if a container supports `find` or `count` method. Another attribute can define if a container supports a `unique` member function, such as `list`.

In this paper we do not deal with *safe* iterators [13]. However, safety can be an orthogonal attribute of iterator types which should be defined as a trait. Thus, STL algorithms can be overloaded on safe iterators, too.

In the specializations one have to set the new trait, too. In the different inserter iterator types and `ostream_iterator` types the `inserter` tag has to be set to `inserting_iterator_tag`. This can be easily done if the `iterator` base type is extended with the new trait.

IV. Generation of warnings

Compilers cannot emit warnings based on the semantical erroneous usage of the library. STLlint is the flagship example for external software that is able to emit warning when the STL is used in an incorrect way [7]. We do not want to modify the compilers, so we have to enforce the compiler to indicate these kind of potential problems [11]. However, `static_assert` as a new keyword is introduced in C++0x to emit compilation errors based on conditions, no similar construct is designed for warnings.

```
template <class T>
inline void warning( T t )
{
}

struct
COPY_ALGORITHM_WITHOUT_INSERTER_ITERATOR
{
};

// ...

warning(
  COPY_ALGORITHM_WITHOUT_\
INSERTER_ITERATOR()
);
```

When the `warning` function is called, a dummy object is passed. This dummy object is not used inside the function template, hence this is an unused parameter. Compilers emit warning to indicate unused parameters. Compilation of `warning` function template results in warning messages, when it is referred and instantiated. No warning message is shown, if it is not referred. In the warning message the template argument is printed. New dummy types have to be written for every new kind of warning.

Different compilers emit this warning in different ways. For instance, Visual Studio emits the following message:

```
warning C4100: 't' : unreferenced formal
  parameter
...
see reference to function template
  instantiation 'void
warning<
COPY_ALGORITHM_WITHOUT_INSERTER_ITERATOR
>(T)'
being compiled

        with
        [
            T=
COPY_ALGORITHM_WITHOUT_INSERTER_ITERATOR
        ]
```

And g++ emits the following message:

```
In instantiation of 'void warning(T)
      [with T =
COPY_ALGORITHM_WITHOUT_INSERTER_ITERATOR
      ]':
... instantiated from here
... warning: unused parameter 't'
```

Unfortunately, implementation details of warnings may differ, thus no universal solution is available to generate custom warnings. However, everyone can find a handy, custom solution for own compiler.

This approach of warning generation has no runtime overhead because the compiler optimizes the empty function body. On the other hand – as the previous examples show – the message refers to the warning of unused parameter, incidentally the identifier of the template argument type is appeared in the message.

V. Modification of algorithms

As an example, now we can overload `copy` algorithm. However, `transform` algorithm can be written likewise.

```
template <class InputIt,
          class OutputIt>
inline OutputIt copy(
  InputIt first,
  InputIt last,
  OutputIt result )
{
  return copy(
    first,
    last,
    result,
    typename
      iterator_traits<OutputIt>::
        inserter() );
}
```

Now, we write the “usual” version of the algorithm. In this case, no warning is emitted:

```
template <class InputIterator,
          class OutputIterator>
OutputIterator copy(
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  __inserting_iterator_tag )
{
  while( first != last )
  {
    *result++ = *first++;
  }
  return result;
}
```

Finally, we create the new version of the algorithm to indicate warnings:

```
template <class InputIterator,
          class OutputIterator>
OutputIterator copy(
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  __non_inserting_iterator_tag )
{
  warning(
   COPY_ALGORITHM_WITHOUT_\
INSERTER_ITERATOR()
  );
  return copy( first,
               last,
               result,
               __inserting_iterator_tag() );
}
```

VI. Believe-Me marks

Generally, warnings should be eliminated. On the other hand, the call of `copy` or `transform` without inserter iterators does not mean problem necessarily.

If the proposed extensions are in use, the following code snippet results in a warning message, but it works perfectly:

```
std::vector<int> vi;
// ...
std::list<int> li( vi.size() );
std::copy( vi.begin(),
           vi.end(),
           li.begin() );
```

Many similar patterns can be shown. We use `copy` or `transform` algorithm to a target, where enough allocated space is available. Moreover, we cannot disable these specific generated warnings by a compiler flag or a preprocessor pragma.

Believe-me marks [9] are used to identify the points in the programtext where the type system cannot obtain if the used construct is risky. For instance, in the hereinafter example, the user of the library asks the type system to "believe" that the target is already allocated in the proper way. This way we enforce the user to reason about the parameters of these algorithms.

```
std::vector<int> v;
// ...
std::list<int> li( v.size() );
std::copy( v.begin(),
           v.end(),
           li.begin(),
           transmogrify,
           ALREADY_ALLOCATED );
```

This can be created by a preprocessor macro:

```
#define ALREADY_ALLOCATED \
  __inserting_iterator_tag()
```

VII. Conclusion

STL is the most widely-used library based on the generic programming paradigm. It is efficient and convenient, but the incorrect usage of the library results in weird or undefined behaviour.

In this paper we present some examples that can be compiled, but at runtime their usage is defective. We argue for an extension of the iterator traits in the library, and based on this extension we generate warning messages during compilation.

The effect of our approach is similar to the STLlint software. STLlint analyzes the programtext and emits warning messages when the STL is used in an erronous way. STLlint is based on a modified compiler and this way it can emit better messages. On the other hand, it is not extensible. Our approach can be used for non-standard containers, iterators, algorithms, too. Compilers cannot know all the generic libraries.

We present an effective approach to generate custom warnings. Believe-me marks are also written to disable warning messages. We overload some algorithms of the STL based on the new traits in order to make the usage of the library safer.

Acknowledgment

The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).

References

[1] A. Alexandrescu, *Modern C++ Design*, Addison-Wesley, 2001.

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

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

[4] G. Dévai, N. Pataki, *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.

[5] G. Dévai, N. Pataki, *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.

[6] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine, *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.

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

[8] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek, *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.

[9] T. Kozsik, T., *Tutorial on Subtype Marks*, in Proc. of the Central European Functional Programming School (CEFP 2006), LNCS **4164**, pp. 191–222.

[10] S. Meyers, *Effective STL - 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley, 2001.

[11] N. Pataki, *Advanced Functor Framework for C++ Standard Template Library* Studia Universitatis Babeş-Bolyai, Informatica, Vol. LVI(1), pp. 99–113.

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

[13] N. Pataki, Z. Szűgyi, G. Dévai, *Measuring the Overhead of C++ Standard Template Library Safe Variants*, In Electronic Notes in Theoretical Computer Science (ENTCS) **264(5)**, pp. 71–83.

[14] Z. Porkoláb, Á. Sipos, N. Pataki, *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.

[15] B. Stroustrup, *The C++ Programming Language (Special Edition)*, Addison-Wesley, 2000.

[16] M. Torgersen, *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.

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

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