Annales Univ. Sci. Budapest., Sect. Comp. 36 (2012) 341–353

COMPILE-TIME ADVANCES OF THE C++ STANDARD TEMPLATE LIBRARY

Norbert Pataki (Budapest, Hungary)

Communicated by Zoltán Horváth

(Received December 22, 2011; revised February 4, 2012;
accepted February 27, 2012)

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++ errors, 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. Every standard container offers a template parameter in order to customize the memory management. Allocator types are accountable for allocation and deallocation of memory.
In this paper we present some scenarios that may cause undefined or weird behaviour at runtime. These scenarios are related to allocators and reverse iterators. We emit warnings while these constructs are used without any modification in the compiler. We also present a general approach to emit "customized" warnings. We support the so-called believe-me marks in order to disable our specific warnings.

1. Introduction

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach [2]. In this way containers are defined as class templates

Key words and phrases: C++, STL, allocators, iterators, warning.
2010 Mathematics Subject Classification: 68N19.
1998 CR Categories and Descriptors: D.3.2.
The Research 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).

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 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 [6]. 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 [4]. The expression problem [25] 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 bug-free or error-free code [8]. 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 [26, 27]. Violating requirement of special preconditions (e.g. sorted ranges) is not checked, but results in runtime bugs [19]. 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 [7].

Another common mistake is related to algorithms which are deleting elements. 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` and `unique` algorithms do not actually remove any element from a container [12].

The previously mentioned `unique` algorithm has uncommon precondition. Equal elements should be in consecutive groups. In general case, using `sort` algorithm is advised to be called before the invokation of `unique`. However, `unique` cannot result in an undefined behaviour, but its result may be counter-intuitive at first time.

Some 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 [23] . 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 properties tested neither at compilation-time nor at runtime. 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.

Associative containers (e.g. `multiset`) use functors exclusively to keep their elements sorted. Algorithms for sorting (e.g. `stable_sort`) and searching in ordered ranges (e.g. `lower_bound`) are typically used with functors because of efficiency. These containers and algorithms need *strict weak ordering*. Containers become inconsistent, if the used functors do not meet the requirement of strict weak ordering [13].

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.

Whereas C++ STL is pre-eminent in a sequential realm, it is not aware of multicore environment [3]. For example, the Cilk++ language aims at multicore programming. This language extends C++ with new keywords and one can write programs for multicore architectures easily. However, the language does not contain an efficient multicore library, just the C++ STL only, which is an efficiency bottleneck in multicore environment. We develop a new STL implementation for Cilk++ to cope with the challenges of multicore architectures [24]. This new implementation can be a safer solution too. Hence, our safety extensions will be included in the new implementation. However, the techniques presented in this paper concern the original C++ STL too.

In this paper we argue for an approach that generates warning during compilation when the STL is used in some improper ways. For example, we want to warn the programmer if a stateful allocator is in use, which is prohibited. We also emit warning, if a reverse iterator is converted to iterator by the `base` method.

This paper is organized as follows. In section 2 we present some motivating examples that can be compiled, but at runtime they can cause problems. In section 3 we present an approach to generate "customized" warnings at compilation time. Then, in section 4 the implementation details of allocator-related warning emission is discussed. Section 5 presents the solution of the problem related to reverse iterators. The related work is being discussed in section 6. Finally, this paper concludes in section 7.

2. Motivation

In this section two different scenarios are shown. The first one is related to reverse iterators and present a flabbergasting phenomenon. The second one is related to allocators and may cause undefined behaviour.

The concept of reverse iterator is straightforward. Reverse iterators iterate through the container from the end to the begin. So, it is easy to find the last occurrence of the value `x` with the `find` algorithm and reverse iterators. Unfortunately, the `erase` method requires `iterator` and cannot take `reverse_iterator`. We can convert it with its `base` method. However, the following code snippet has a strange effect:

```
std::vector<int> v;
int x;
// ...
v.erase( std::find( v.rbegin(),
                    v.rend(),
                    x ).base() );
```

This code snippet does not erase the value of `x` from the container, but it does erase the next element from the container. The call of `base` returns iterator, which points the next element of the container. This behaviour is counter-intuitive. The iterator that returns by the `base` is perfect when one uses it as a position to insert some elements, but erroneous if someone deletes it. Our implementation generates warning if one uses the `base` method.

Every standard container offers a template parameter in order to customize the memory management. Allocator types are accountable for allocation and deallocation of memory. According to the standard the STL implementations may suppose that allocators with the same type are equal, hence many operations can take advantage of this. For instance, the `list`'s `splice` method can be implemented in an easy and exception-safe way: copying is not necessary, only some pointers are set:

```
template <typename T>
class SpecialAllocator {...};

typedef SpecialAllocator<Widget> SAW;

list<Widget, SAW> L1;
list<Widget, SAW> L2;
...
L1.splice( L1.begin(), L2 );
```

When `L1` is being destructed, all elements must be deallocated. It can only use `L1`'s allocator even if the element is allocated by `L2`'s allocator. Hence, the allocation and deallocation cannot depend on any other information. If an allocator has state and the allocation and deallocation depends on this state, it results in an undefined behaviour. The compilers do not check this requirement and it cannot be discovered at runtime either. Hence, allocators should be stateless types. Our implementation generates warning at compilation-time, if the user applies a non-stateless allocator.

3. 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 [9]. We do not want to modify the compilers, so we have to enforce the compiler to indicate these kinds of potential problems. Although `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. C++ templates are ideal constructs to do compile-time checks and emit warnings and errors [24].

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

struct
DO_NOT_CALL_FIND_ALGORITHM_ON_SORTED_CONTAINER
{
};

// ...

warning(
  DO_NOT_CALL_FIND_ALGORITHM_ON_SORTED_CONTAINER()
);
```

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 warnings to indicate unused parameters. Compilation of the `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<
DO_NOT_CALL_FIND_ALGORITHM_ON_SORTED_CONTAINER
>(T)'
being compiled

        with
        [
            T=DO_NOT_CALL_FIND_ALGORITHM_ON_SORTED_CONTAINER
        ]
```

And g++ emits the following message:

```
In instantiation of 'void warning(T)
      [with T = DO_NOT_CALL_FIND_ALGORITHM_ON_SORTED_CONTAINER]':
... 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 his or her own compiler.

This approach of warning generation has no runtime overhead because the compiler optimizes the empty function body. On the other hand – as 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.

This method can be used to detect possible defects based on compile-time information. It can detect many potential errors, e.g. incorrect instantiations [15], problematic algorithm parameters [16], erroneous base types [13] and many more. However, it cannot be used to deal with runtime information. Nevertheless, this method has limitations related to compile-time information, too. It works at compilation time, but this approach cannot deal with the syntax tree or more detailed contexts.

4. Stateless allocators

It is important to make sure that the allocators with the same type are equal, hence they do not have any state. They may not have any nonstatic data members because STL implementations take advantage of this idiosyncrasy. However, compilers do not check it, and the error cannot be discovered at runtime either. Allocators with state may ruin the containers.

Our solution consists of the following steps. First, we write a class which defines the message of the warning and transform the `warning` template into a template class.

```
class ALLOCATOR_WITH_STATE
{
};

template<bool b, class Allocator>
struct __WARNING
{
  __WARNING()
  {
    warning( ALLOCATOR_WITH_STATE() );
  }
};

template <class Fun>
struct __WARNING<true, Fun>
{
};
```

Next, a wrapper class is developed in order to trigger the compile-time check. Hence, this class inherits from the allocator type, and every public method is available just like the original allocator. The compile-time test uses the Boost type traits library, in which statelessness test is being implemented [10]:

```
template <class Alloc>
class __Stateless: public Alloc
{
  WARNING< boost::is_stateless<Alloc>::value, Alloc > ___;
};
```

There is a small change in the implementation of the STL. Containers use the __Stateless template instead of the allocator, just like the hereinafter example:

```
template <class T, class Alloc = allocator<T> >
class list
{
  __Stateless<Alloc> allocator;
  // ...
};
```

If any user defined allocator violates the rule, compilation warning is emitted. The programmer gets a warning diagnostics that the allocator may be problematic. The ALLOCATOR_WITH_STATE identifier also can be seen in the warning message. Believe-me marks are not supported, because stateful allocators are not reasonable.

5. Reverse iterators

Reverse iterators seem to be easy and straightforward, but their conversion is not easy and straightforward at all. The base method is not intuitive, because it returns an iterator that points to an other element in the container. So, we help the library users with warning emission to be careful.

Two adaptor classes are affected: reverse_bidirectional_iterator and reverse_iterator. We present our approach on reverse_iterator:

```
struct BASE_ITERATOR_POINTS_TO_THE_NEXT_ELEMENT{};
struct I_Know_What_Base_Returns{};

#define I_KNOW_WHAT_BASE_RETURNS I_Know_What_Base_Returns()

template <typename Iterator>
class reverse_iterator: public
  std::iterator<
    typename std::iterator_traits<Iterator>::iterator_category,
    typename std::iterator_traits<Iterator>::value_type,
    typename std::iterator_traits<Iterator>::difference_type,
    typename std::iterator_traits<Iterator>::pointer,
    typename std::iterator_traits<Iterator>::reference >
```

```
{
  // ...
public:
  Iterator base() const
  {
    warning( BASE_ITERATOR_POINTS_TO_THE_NEXT_ELEMENT() );
    return base( I_Know_What_Base_Returns() );
  }

  Iterator base( I_Know_What_Base_Returns ) const
  {
    // original implementation of base
  }
};
```

The implementation is not difficult. The `base` method emits warning with the assistance of the previously presented `warning` template function. We overload the `base` method. The original version emits the warning and invokes the parametrized one.

Generally, warnings should be eliminated. On the other hand, the usage of `base` does not necessarily mean problem. It can be used safely. However, we cannot disable the generated warning if it is in use.

Believe-me marks [11] are used to identify the points in the program text 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 programmer is conscious of the `base`. This way we enforce the user to reason about the usage of the library:

```
std::vector<int> v;
int x;
// ...
v.erase( std::find( v.rbegin(),
                    v.rend(),
                    x ).base( I_KNOW_WHAT_BASE_RETURNS ) - 1 );
```

The implementation of `reverse_bidirectional_iterator` is straightforward.

6. Related work

The most well-known tool that detects the incorrect usage of the STL is STLlint [9]. It is an online tool, but its support is cancelled. However, it is based on a modified compiler, and it is closed-source and it cannot work with third party STL-like containers. Nevertheless, it is required to invoke another tool after successful compilation by the programmer. Another important remark is semantical issues generally related to the libraries, not the compilers. Compilers cannot know all the libraries. In constrast, our approach is extensible for non-standard elements too. Our solution is more platform-independent, because we do not deal with compiler, only the library itself is modified. However, STL is template library, so it cannot be compiled previously, its source is available on every platform.

Our approach has been used in many different ways. The instantiations of STL containers have been analyzed. Usage of containers of auto pointers (COAP) is prohibited, `vector<bool>` as a full specialization of the `vector` does not meet the requirements of the Standard. Our approach is able to detect if one of these containers is instantiated [15]. User-defined functor objects have been inspected from the view of proper adaptability [13]. Usage of STL algorithms has been safer with the extension of iterator traits type [16].

Some properties can be checked only at runtime [20]. For instance, usage of invalid iterators are discovered at runtime [19]. Special preconditions of STL algorithms also can be evaluated at runtime [18]. Functors for sorting activity are required to be strict-weak orderings. We test them at runtime automatically [13]. STL takes intervals as two independent iterators. The ranges are natural abstraction of this approach. A range-based implementation of the STL is considered [14].

The `printf` function of the C standard library takes parameters. The first argument is a formatting string, which specifies how to print the following arguments to the output. The value of the formatting string is not handled by the compiler, hence the usage of `printf` may result in runtime errors. We have developed a metastring library in which the values of the strings are compile-time information. We have implemented a safer version of `printf` with these metastrings that is able to check the type information and able to detect the incorrect usage during compilation [22].

Warning generation can be used for other purposes, too. They are applied successfully when C++ template metaprograms are visualized. The chain of instantiations is detected by the generated warnings [5].

7. Conclusion

C++ STL is the most important library based on the generic programming. It is a handy, useful standard library that contains many indispensable containers and primary algorithms, etc.

However, the incorrect usage of the library may result in an undefined behaviour that should be avioded. Some reasonable scenarios have weird effects. In this paper we argue for an extension to make the STL safer. With our extension the compiler is able to generate warning if one of these dangerous constructs is in use. We do not modify the compiler itself, but small changes are done in the implementation of the STL.

In this paper we present a generic approach that is able to generate "customized" warnings by the compiler. Our STL implementation is able to detect if someone uses stateful allocator, which is strictly prohibited. Our implementation warns the programmer if a reverse iterator's `base` method is called that is dubious construct. We support believe-me marks to disable our specific warnings.

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., R.A. Towle and A.A. Stepanov,** Range partition adaptors: a mechanism for parallelizing STL, *ACM SIGAPP Applied Computing Review*, **4(1)** (1996), 5–6.

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

[5] **Borók-Nagy, Z., V. Májer, J. Mihalicza, N. Pataki and Z. Porkoláb,** Visualization of C++ Template Metaprograms, in: J. Vinhu., C. Marinescu (Eds.) *Proc. of Tenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010)*, 167–176.

[6] **Czarnecki, K. and U.W. Eisenecker,** *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2000.

[7] **Dévai, G. and N. Pataki,** Towards verified usage of the C++ Standard Template Library, in: Z. Horváth, L. Kozma, V. Zsók (Eds.) *Proc. of the 10th Symposium on Programming Languages and Software Tools, (SPLST) 2007,* 360–371.

[8] **Dévai, G. and N. Pataki,** A tool for formally specifying the C++ Standard Template Library, *Ann. Univ. Sci. Budapest. Comput.*, **31** (2009), 147–166.

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

[10] **Karlsson, B.,** *Beyond the C++ Standard Library: An Introduction to Boost*, Addison-Wesley, 2005.

[11] **Kozsik, T.,** Tutorial on Subtype Marks, *Lecture Notes in Comput. Sci.*, **4164** (2005), 191–222.

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

[13] **Pataki, N.,** Advanced Functor Framework for C++ Standard Template Library, *Studia Universitatis Babeş-Bolyai, Informatica*, **LVI(1)** (2011), 99–113.

[14] **Pataki, N.,** C++ Standard Template Library by Ranges, in: A. Egri-Nagy, E. Kovács, G. Kovásznai, G. Kusper, T. Tómács (Eds.) *Proc. of the 8th International Conference on Applied Informatics (ICAI 2010), Volume 2*, 367–374.

[15] **Pataki, N.,** C++ Standard Template Library by template specialized containers, *Acta Universitatis Sapientiae, Informatica*, **3(2)** (2011), 141–157.

[16] **Pataki, N. and Z. Porkoláb,** Extension of iterator traits in the C++ Standard Template Library, in: M. Ganzha, L. A. Maciaszek, M. Paprzycki (Eds.) *Proc. of the Federated Conference on Computer Science and Information Systems, FedCSIS 2010,* 911–914.

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

[18] **Pataki, N., Z. Szűgyi and G. Dévai,** C++ Standard Template Library in a safer way, in: Z. Porkoláb, N. Pataki (Eds.) *Proc. of Workshop on Generative Technologies 2010, (WGT 2010),* 46–55.

[19] **Pataki, N., Z. Szűgyi and G. Dévai,** Measuring the overhead of C++ Standard Template Library safe variants, *Electronic Notes in Theoret. Comput. Sci.*, **264(5)** (2011), 71–83.

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

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

[22] **Szűgyi, Z., Á. Sinkovics, N. Pataki and Z. Porkoláb,** C++ Metastring Library and its applications, *Lecture Notes in Comput. Sci.*, **6491** (2010), 461–480.

[23] **Szűgyi, Z., Á. Sipos and Z. Porkoláb,** Towards the modularization of C++ concept maps, in: Z. Porkoláb, N. Pataki (Eds.) *Proc. of Workshop on Generative Programming (WGT 2008),* 33–43.

[24] **Szűgyi, Z., M. Török and N. Pataki,** Multicore C++ Standard Template Library in a generative way, *Electronic Notes in Theoret. Comput. Sci.*, **279(3)** (2011), 63–72.

[25] **Torgersen, M.,** The expression problem revisited – Four new solutions using generics, *Lecture Notes in Comput. Sci.*, **3086** (2004), 123–143.

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

[27] **Zólyomi, I.and Z. Porkoláb,** Towards a general template introspection library, *Lecture Notes in Comput. Sci.*, **3286** (2004), 266–282.

N. Pataki
Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
H-1117 Budapest, Pázmány P. sétány 1/C
Hungary
patakino@elte.hu