Acta Univ. Sapientiae, Informatica, **3,** 2 (2011) 141–157

C++ Standard Template Library by template specialized containers

Norbert PATAKI
Dept. of Programming Languages and Compilers
Faculty of Informatics, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest,
Hungary
email: patakino@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 the number of 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.

In this paper we present some typical scenarios that may cause runtime or portability problems. We emit warnings and errors while these risky constructs are used. We also present a general approach to emit "customized" warnings. We support the so-called "believe-me marks" to disable warnings. We present another typical usage of our technique, when classes become deprecated during the software lifecycle.

1 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 [23]. C++ STL is widely-used because it is a

Computing Classification System 1998: D.3.2
Mathematics Subject Classification 2010: 68N19
Key words and phrases: C++, STL, template, compilation

very handy, standard C++ library that contains beneficial containers (like list, vector, map, etc.) and a large number of algorithms (like sort, find, count, etc.) among other utilities [5].

The STL was designed to be extensible [14]. 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 [26] 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 bugless or error-free code [7]. Contrarily, incorrect application of the library may introduce new kinds of problems [22].

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 [27, 28]. Violating requirement of special preconditions (e.g. sorted ranges) is not tested, but results in runtime bugs [20]. 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* [17]. Further reference of using invalid iterators causes undefined behaviour [19].

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. Since, for example the `remove` algorithm does not actually remove any element from a container [13].

Some of the properties are checked at compilation time [8]. 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 [10]. Other properties are checked at runtime [21], like the standard vector container offers an `at` method which tests if the index is valid and it raises an exception otherwise [18].

Unfortunately, there are still a large number of properties that are tested neither at compilation-time nor at run-time. The observance of these properties is in the charge of the programmers [6]. 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 used functors do not meet the requirement of strict weak ordering [15].

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 before list's iterators cannot be passed to `sort` algorithm, hence code with this mistake cannot be compiled [24]. To overcome this problem list has a member function called `sort`. In these cases, although the code compiles, the member function calls are preferable 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. Although 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[25]. This new implementation can be safer solution, too. Hence, our safety extensions will be included in the new implementation. However, the advised techniques presented in this paper concern to the original C++ STL, too.

In this paper we argue for an approach that generates warnings or errors when a template container is instantiated with improper parameters. These instantiations mean erroneous, unportable code or other weird compilation effects. A general technique is presented to express custom warnings at compilation time. Our technique is able to indicate the usage of deprecated classes.

This paper is organized as follows. In Section 2 we present an approach to generate "customized" warnings at compilation time. After, in Section 3 we describe the specialized vector container which contains boolean values. We show why this container is problematic, and argue for warnings when it is in use. We explain the forbidden *containers of auto pointers* and present an approach to disable their usage by template specializations. In Section 5 the so-called *believe-me marks* are introduced. Finally, this paper concludes in Section 7.

2 Generation of warnings

Compilers cannot emit warnings based on the erroneous usage of the library. STLlint is the flagship example for external software that is able to emit warnings 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. However, `static_assert` as a new keyword is introduced in C++0x to emit compilation errors based on conditions, but no similar construct is designed for warnings.

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

struct VECTOR_BOOL_IS_IN_USE { };

// ...

warning( VECTOR_BOOL_IS_IN_USE() );
```

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 [16]. No warning message is shown if it is not referred. In the warning message the template argument is referred. New dummy type has 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<VECTOR_BOOL_IS_IN_USE>(T)'
being compiled

        with
        [
            T=VECTOR_BOOL_IS_IN_USE
        ]
```

And g++ emits the following message:

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

Unfortunately, implementation details of warnings may differ, thus there is no universal solution to generate custom warnings.

This approach of warning generation has no runtime overhead inasmuch as 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.

3 The weirdest vector

In this section we present the basic idea behind the specialized `vector<bool>` container. We present the pros and cons of this weird type. We argue for generate warnings at compilation-time if a programmer uses `vector<bool>` because it is the embodiment of the weird container.

Many programmers think that the `vector<bool>` is the instantiation of STL's `vector` template, but it is not true. On many platforms `sizeof( int ) == sizeof( bool )` because of reverse compatibility. (In the C programming language `int` type has been used to represent Boolean values.) Hence, the `vector<bool>` is a template specialized container to develop a more advanced, denser implementation for boolean values. This representation is able to represent 32 boolean values on 4 bytes.

The following code sketch represents the connection between `vector<bool>` and `vector` template:

```
template <class T, class Alloc = std::alloc>
class vector
{
  T* p;
  size_t capacity;
  size_t size;
public:
  vector()
```

```
  {
    // ...
  }

  void push_back( const T& t )
  {
    // ...
  }

  // ...
};

template <class Alloc>
class vector<bool, class Alloc>
{
  // dense representation of vector bool
  // No bool* member
public:
  // public interface is similar to the previous one

  void push_back( const bool& t )
  {
    // ...
  }

  vector()
  {
    //...
  }
};
```

So, the `vector<bool>` has a special representation to handle dense boolean values. It is designed to be effective when someone stores boolean values. But it has weird behaviour compared to the `vector` template:

```
std::vector<int> a;
a.push_back( 3 );
int* p = &a[0];
```

```
std::vector<bool> b;
b.push_back( true );
bool* q = &b[0];
```

The previous code does not compile because of the `bool* q = &b[0];` assignment. However, when the template `vector` is in use, its counterpart does compile. It is a contradiction in terms, because this way the `vector<bool>` cannot meet the requirements of C++ Standard. Hence, it is not advised to use. Let us see the background of this compilation issue:

```
template <class T, class Alloc = std::alloc>
class vector
{
  T* p;
  //..
public:
  T& operator[]( int idx )
  {
    return p[idx];
  }

  const T& operator[]( int idx ) const
  {
    return p[idx];
  }
  // ...
};

template <class Alloc>
class vector<bool, class Alloc>
{
  // dense representation of vector bool
  // No bool* member
public:

  class bool_reference
  {
     // ...
  };
```

```
  bool_reference operator[]( int idx )
  {
     // ...
  }
};
```

Because the `vector<bool>` does not hold actual bool values it cannot return `bool&`. Hence, a proxy class is developed which actually simulates `bool&`. However, conversions cannot be defined between *pointer to a* `bool_reference` and a *pointer to a* `bool`. This behaviour can be much more appalling, when the programmer uses `vector` as a base class. Arcane error messages are emitted when the subtype is instantiated with `bool`.

Unfortunately, most of STL references hardly mention that `vector<bool>` is not the instantiation of template, but a completely different class. It would be useful if the compiler indicated if the programmer used `vector<bool>` container, even intentionally or inadvertently.

Now it is not difficult to emit warning with the presented function. Fortunately, `vector<bool>` is still a class template because the type of its allocator is a template parameter. So, the compilation warning is emitted only when this template class is instantiated, hence someone uses it:

```
template<class Allocator>
class vector<bool, Allocator>
{
  // ...
public:
  vector()
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
    // ...
  }

  template<class InputIterator>
  vector( InputIterator first, InputIterator last )
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
    // ...
  }
```

```
  vector( size_t n, const bool& value = bool() )
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
    // ...
  }

  vector( const vector& rhs)
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
    // ...
  }

};
```

In Section 2 the emitted warning message can be seen.

4 Containers of auto pointers

In this section the containers of auto pointers are detailed. We present their motivation and reason why are they problematic. We present a solution to forbid the usage of these kinds of containers.

Usually, auto pointers (`std::auto_ptr` objects) make easier to manage objects in the heap memory. This class assists in memory management. The auto pointers deallocate the pointed memory when they are gone out of scope [23]. Hence, they prevent memory leaks:

```
void f()
{
  std::auto_ptr<int> p( new int( 5 ) );
  // no memory leak
}
```

Containers of STL are template classes, so technically they should be instantiated with auto pointers and store auto pointers that point to the heap:

```
std::vector<std::auto_ptr<int> > v;
v.push_back( new int( 7 ) );
// ...
```

The previous code snippet seems to be safe. On the other hand, the C++ Standardization Committee forbid the usage of *containers of auto pointers (COAPs)*. The motivation behind this idea is that the copy of auto pointers is strange:

```
std::auto_ptr<int> p( new int( 3 ) );
std::auto_ptr<int> q = p;
// At this point p is null pointer
```

The copied auto pointer becomes null pointer. Only one auto pointer is able to point to any object in the heap. This one is responsible for the deallocation.

So, if COAPs are not be forbidden, the following code snippet results in a very strange behaviour:

```
struct Auto_ptr_less
{
  bool operator()( const std::auto_ptr<int>& a,
                   const std::auto_ptr<int>& b )
  {
    return *a < *b;
  }
};

std::vector<std::auto_ptr<int> > v;
v.push_back( new int( 7 ) );
// ...
std::sort( v.begin(), v.end(), Auto_ptr_less() );
```

Some of the pointers may become null pointer because of the assignments during swapping vector's elements when it is necessary. This is the reason why COAPs are forbidden.

Unfortunately, some of the compilers and STL platforms are still permitting the usage of COAPs, some of them are not. This inhibits the writing of portable code [13].

We argue for an extension to emit compilation error if COAPs are in use. We have to create specializations for auto pointers. The trick that is we do not write the implementation for the auto pointer specializations. Thus, these specializations are declared, but are not defined types. For instance, the vector declaration can be the following:

```
template <class T, class Alloc>
class vector< std::auto_ptr<T>, Alloc>;
```

The instantiation of a COAP results in the hereinafter error message:

```
error: aggregate 'std::vector<std::auto_ptr<int>,
  std::allocator<std::auto_ptr<int> > > v'
  has incomplete type and cannot be defined
```

We have to develop these declarations for all standard containers. These declarations mean bugless and more portable code.

5 Believe-me marks

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

Believe-me marks [12] 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 specialized vector container. This way we enforce the user to reason about the parameters of containers.

First, we create a new type which stands for the believe-me mark:

```
struct I_KNOW_VECTOR_BOOL { };
```

After, we extend the vector template container with one new template parameter. The new template parameter has default parameter value, so it is reverse compatible with the original container. This parameter has not been taken advantage of, and has no effect on the implementation:

```
template <class T, class Alloc = std::alloc, class Info = int>
class vector { };
```

Let us consider, that the original implementation of `vector<bool>` which does not generate warning has been removed to a new template class:

```
template <class Alloc>
class __VectorBool
{
  // original implementation of vector<bool>
};
```

The new template parameter has effect on the `vector<bool>` specialization:

```
template <class Alloc>
class vector<bool, I_KNOW_VECTOR_BOOL, Alloc>:
  public __VectorBool<Alloc>
{ };

template <class Alloc>
class vector<bool, Alloc, I_KNOW_VECTOR_BOOL>:
  public __VectorBool<Alloc> { };

template <class Alloc, class Info>
class vector<bool, Alloc, Info>:
  public __VectorBool<Alloc>
{
public:
  vector(): __VectorBool<Alloc>()
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
  }

  template<class InputIterator>
  vector( InputIterator first, InputIterator last ):
    __VectorBool<Alloc>( first, last )
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
  }

  vector( size_t n, const bool& value = bool() ):
    __VectorBool<Alloc>( n, value )
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
  }

  vector( const vector& rhs): __VectorBool<Alloc>( rhs )
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
  }
};
```

In this case no compilation warning is emitted if the last added template parameter is `I_KNOW_VECTOR_BOOL`, otherwise the mentioned warning can be seen during compilation.

6 Deprecated classes

In section 3 we generated warnings when a template-specialized class was used. A similar idea can be mentioned. It would be useful to generate warnings when the usage of classes becomes unsupported.

A common idea during a software lifecycle is, that some of the classes are not deleted from the project, but their usage is not advised. These classes are called *deprecated*. Deprecated annotation can be added to classes in Java. Instantiation of deprecated classes results in compilation warnings [11]. However, no similar technique is used in C++.

First, we create some utility classes for warning generation:

```
struct DeprecatedClass { };

template <class DEPRECATED>
struct Deprecated
{
  Deprecated()
  {
    warning( DeprecatedClass() );
  }
};
```

The role of the template parameter in `Deprecated` struct is to pass the identifier of deprecated class to the emitted warning.

Now, let us consider that the following class becomes deprecated during software lifecycle:

```
class Foo
{
  // ...
public:
  Foo( int a, int b)
  {
    // ...
  }
};
```

The user has to add one more base class to the deprecated class. This does not mean limitation because the C++ programming language supports multiple inheritance. For example:

```
class Foo: public Deprecated<Foo>
{
  // very same...
};
```

The following warning is received from the compiler:

```
In instantiation of 'void warning(T)
  [with T = DeprecatedClass]':
...   instantiated from
  'Deprecated<DEPRECATED>::Deprecated()
  [with DEPRECATED = Foo]'
...   instantiated from here
... warning: unused parameter 't'
```

However, this message is received irrespectively of its usage. If the usage is important, the deprecated class or a called method or constructor must be a template. This transformation cannot be executed automatically with the respect of client code. Our future work is to overcome this situation.

We do not advise to make believe-me marks for the deprecated classes inasmuch as always exists a better approach to use.

7 Conclusions

C++ 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 argue for some extensions to make the STL itself safer. Not supported or not advised instantiations result in compilation warnings and errors to prevent unportable or defective code.

We present an effective approach to generate custom warnings. Believe-me marks are also written to disable warning messages. With our technique classes can be marked deprecated, too.

Acknowledgements

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: Generic Programming and Design Patterns Applied*, Addison-Wesley, Reading, MA, 2001. ⇒142

[2] M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley, Reading, MA, 1998. ⇒141

[3] M. H. Austern, R. A. Towle, A. A. Stepanov, Range partition adaptors: a mechanism for parallelizing STL, *ACM SIGAPP Applied Computing Review* **4,** 1 (1996) 5–6. ⇒143

[4] T. Becker, STL & generic programming: writing your own iterators, *C/C++ Users Journal* **19,** 8 (2001) 51–57. ⇒142

[5] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, Reading, MA, 2000. ⇒142

[6] G. Dévai, N. Pataki, Towards verified usage of the C++ Standard Template Library, *Proc. 10th Symposium on Programming Languages and Software Tools (SPLST) 2007*, Dobogókő, Hungary, pp. 360–371. ⇒142

[7] G. Dévai, N. Pataki, A tool for formally specifying the C++ Standard Template Library, *Ann. Univ. Sci. Budapest. Comput.* **31** (2009) 147–166. ⇒142

[8] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine, Concepts: linguistic support for generic programming in C++, *Proc. 21st Annual ACM SIGPLAN 2006 Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, Portland, Oregon, USA, pp. 291–310. ⇒142

[9] D. Gregor, S. Schupp, Stllint: lifting static checking from languages to libraries, *Software – Practice & Experience*, **36,** 3 (2006) 225-254. ⇒144

[10] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek, Algorithm specialization in generic programming: challenges of constrained generics in C++, *Proc. ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI 2006)*, Ottawa, Canada, pp. 272–282. ⇒142

[11] Z. Juhász, M. Juhás, L. Samuelis, Cs. Szabó, Teaching Java programming using case studies, *Teaching Mathematics and Computer Science* **6,** 2 (2008) 245–256. ⇒153

[12] T. Kozsik, Tutorial on subtype marks, in *Proc. Central European Functional Programming School (CEFP 2005), Lecture Notes in Comput. Sci.* **4164** (2006) 191–222. ⇒151

[13] S. Meyers, *Effective STL – 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley, Reading, MA, 2001. ⇒142, 150

[14] D. R. Musser, A. A. Stepanov, Generic programming, *Proc. International Symposium ISSAC'88 on Symbolic and Algebraic Computation, Lecture Notes in Comput. Sci.* **358** (1989) 13–25. ⇒142

[15] N. Pataki: C++ Standard Template Library by safe functors, *Abstracts 8th Joint Conference on Mathematics and Computer Science (MaCS'10)*, Komárno, Slovakia, July 14-17, 2010, p. 44. ⇒143

[16] N. Pataki, Advanced functor framework for C++ Standard Template Library, *Stud. Univ. Babeş-Bolyai Inform.* **56,** 1 (2011) 99–113. ⇒144

[17] N. Pataki, C++ Standard Template Library by ranges, *Proc. 8th International Conference on Applied Informatics (ICAI 2010)* Vol. 2., pp. 367–374. ⇒142

[18] N. Pataki, Z. Porkoláb, Z. Istenes, Towards soundness examination of the C++ Standard Template Library, *Proc. Electronic Computers and Informatics (ECI 2006)*, pp. 186–191. ⇒142

[19] N. Pataki, Z. Szűgyi, G. Dévai, C++ Standard Template Library in a safer way, *Proc. Workshop on Generative Technologies 2010 (WGT 2010)*, Paphos, Cyprus, pp. 46–55. ⇒142

[20] N. Pataki, Z. Szűgyi, G. Dévai, Measuring the overhead of C++ Standard Template Library safe variants, *Electronic Notes in Theoret. Comput. Sci.* **264,** 5 (2011) 71–83. ⇒142

[21] P. Pirkelbauer, S. Parent, M. Marcus, B. Stroustrup, Runtime concepts for the C++ Standard Template Library, *Proc. ACM Symposium on Applied Computing 2008*, Fortaleza, Ceará, Brazil, pp. 171–177. ⇒142

[22] Z. Porkoláb, Á. Sipos, N. Pataki, Inconsistencies of metrics in C++ Standard Template Library, *Proc. 11th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE)*, Berlin, Germany, pp. 2–6. ⇒142

[23] B. Stroustrup, *The C++ Programming Language (Special edition)*, Addison-Wesley, Reading, MA, 2000. ⇒141, 149

[24] Z. Szűgyi, Á. Sipos, Z. Porkoláb, Towards the modularization of C++ concept maps, *Proc. Workshop on Generative Programming (WGT 2008)*, Budapest, Hungary, pp. 33–43. ⇒143

[25] Z. Szűgyi, M. Török, N. Pataki, Towards a multicore C++ Standard Template Libary, *Proc. Workshop on Generative Programming (WGT 2011)*, Saarbrücken, Germany, pp. 38–48. ⇒143

[26] M. Torgersen, The expression problem revisited – Four new solutions using generics, *Proc. European Conference on Object-Oriented Programming (ECOOP) 2004, Lecture Notes in Comput. Sci.* **3086** (2004) 123–143. ⇒142

[27] L. Zolman: An STL message decryptor for visual C++, *C/C++ Users Journal*, **19(7)** (2001) 24–30. ⇒142

[28] I. Zólyomi, Z. Porkoláb: Towards a general template introspection library, *Proc. of Generative Programming and Component Engineering: Third International Conference (GPCE 2004), Lecture Notes in Comput. Sci.* **3286**, 266–282. ⇒142

Received: March 16, 2011 • Revised: July 20, 2011