Proceedings of the 10th International Conference on Applied Informatics
Eger, Hungary, January 30–February 1, 2017. pp. 129–136
doi: 10.14794/ICAI.10.2017.129

Detecting Misusages of the C++ Standard Template Library

Gábor Horváth, Attila Páter-Részeg, Norbert Pataki

Dept. of Programming Languages and Compilers
Fac. of Informatics, Eötvös Loránd University, Budapest
xazax.hun@gmail.com, athilarex730@gmail.com, patakino@elte.hu

Abstract

The C++ Standard Template Library (STL) is the most well-known and widely used library that is based on the generic programming paradigm. The STL takes advantage of C++ templates, so it is an extensible, effective but flexible system. Professional C++ programs cannot miss the usage of the STL because it increases quality, maintainability, understandability and efficacy of the code.

However, the usage of C++ STL does not guarantee bugfree or error-free code. Contrarily, incorrect application of the library may introduce new types of problems. Unfortunately, there is still a large number of properties are tested neither at compilation-time nor at run-time. It is not surprising that in implementation of C++ programs so many STL-related bugs may occur.

It is clearly seen that the compilation validation is not enough to exclude the misuage of STL. Our paper introduces different approaches for the validation of the C++ STL's usage. We take advantage of metaprogramming techniques, static analysis based on the Clang compiler infrastructure and gdb debugging tool as well.

Keywords: C++, STL, validation

MSC: 68N15 Programming languages

1. Introduction

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach [1]. In this way containers are defined as class templates and many algorithms can be implemented as function templates [15]. Furthermore, algorithms are implemented in a container-independent way, so one can use them with

different containers [8]. 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 [2]. 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. The expression problem is solved with this approach [16]. STL also includes adaptor types which transform standard elements of the library for a different functionality [11]. By design, STL is implemented with application of C++ templates to ensure the efficiency and cannot be compiled in advance. However, we can take advantage of this fact in our methods.

The usage of STL increases the code quality and helps to avoid the classical C/C++ problems (e.g. memory leak). On the other hand, the STL introduces new kinds of problems that come from the generic approach. We have figured out different methods to detect the misusage of STL. We present a metaprogramming-based solution, a static analysis tool and debugger-oriented framework.

This paper is organized as follows. In section 2 we present many STL-related difficulties that may result in runtime errors. Compilers typically do not detect these problem. We present our different approaches to realize the STL-related problems: the first one deals with template metaprogramming, the second one is Clang-based static analysis tool and the third one is a debugger-driven framework. Finally, this paper concludes in section 4.

2. Motivation

C++ STL is handy template library that helps to overcome the classical C/C++ bugs but it introduces new kinds of issues. 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]. Violating requirement of special preconditions (e.g. sorted ranges) is not checked, but results in runtime bugs [13]. 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 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 [6].

The previously mentioned `unique` algorithm has uncommon precondition. Elements with the same value should be in consecutive groups. In general case, using `sort` algorithm is advised to be called before the invocation 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. 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 [3].

Unfortunately, there is 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.

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 [9].

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.

3. Approaches

3.1. Metaprogramming

This approach is based on the template construct which is able to evaluate Turing-complete checks at compilation time [14]. However, this approach cannot deal with abstract syntax trees, only template instantiations can be considered. This approach is typically, non-intrusive, so we have to modify the STL implementation itself. On the other hand, these evaluations is part of the usual compilation process and cannot be avoided.

The first goal is to emit custom warning messages from the compilers even if the compiler is not open source. We use the following template function for generating warning:

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

This template generates warning when it is instantiated and in the warning message the template argument is highlighted, so new kind of warning requires a

dummy type:

```
struct DO_NOT_CALL_FIND_ALGORITHM_ON_SORTED_CONTAINER
{
};
```

One can instantiate the template with this dummy type easily:

```
warning( DO_NOT_CALL_FIND_ALGORITHM_ON_SORTED_CONTAINER() );
```

Different compilers emit the warning in different ways. Microsoft Visual Studio presents the following warning:

```
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
        ]
```

The g++ compiler emits the warning:

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

With this approach we can modify our STL implementation to evaluate specific instantiations: we have to add the warning method to `vector<bool>` specialization's constructor or to the `auto_ptr` partial specializations. The conversion of reverse iterators also can be detected with this technique. There are some cases when more sophisticated approaches are required. The `iterator_traits` type also can be extended with extraordinary members to pass metainformation from the containers to the algorithms (e.g. sortedness of container). The metaprogramming facilities also can be used these metadata and emit compilation warnings in specific cases, e.g. `count` algorithm on a sorted container [12]. Typically programmers may use believe-me marks to disable specific warnings.

3.2. Static analysis

This approach is based on pattern matching on the syntax tree of the analysed program source code. We use the syntax tree that is generated by the Clang [5]

compiler. The syntax tree of a compiler contains sufficient amount of information to answer several questions regarding the source code.

However, in order to parse the source of the program we need to know the exact compiler arguments that were used to compile that application. This is necessary because the compiler arguments can modify the semantics of the source code for example macros can be defined using compiler arguments.

To collect the compilation arguments the most efficient and portable way is to use fake compilers that are logging their parameters and forwards them to a real compiler afterwards. This way the logging itself is independent of the make system that is used. The source code is parsed with the same compilation parameters that was logged.

After we retrieved the syntax tree of the analysed program from the compiler the pattern matching process begins. Multiple patterns are matched lazily on the syntax tree with only one traversal. The source positions for the matched nodes in the syntax trees are collected.

The source positions in the collected results are filtered based on exclude lists that contains of the false positive matches. These exclude lists have to be maintained by the user of the tool. Afterwards the positions are translated into user-friendly warning messages.

One of the downsides is that, the compiler can only parse one translation unit at a time. Some useful information might reside in a separate translation unit making it impossible to detect some class of issues. Fortunately due to the structure of STL most of the library code is available in system header files. For this reason if a translation unit is utilizing some STL features, the corresponding headers are likely to be the part of that unit. This structure mitigates the limitations of the compiler, translation unit boundaries are not likely to be a problem when analysing STL misuse patterns.

Each of the checkers that are able to detect certain class of bad smells are implemented as a predicate on the syntax tree. These predicates are loosely coupled. We have focused on the extendibility, thus it is very easy to add further checkers to our tool.

This approach is more subtle than metaprogramming and it is able to detect bugs and smells that cannot be discovered by metaprograms: using the "swap-trick" to decrease a vector's unnecessary allocated capacity, one can use `shrink_to_fit` method since C++11 [4]. It is also can be detected if someone calls an algorithm that removes element(s) from a container but does not call the container's `erase` method to remove the element(s) actually. This tool also can detect instantiation of a `vector<bool>` or a container of `auto_ptr` (COAP). Using COAP objects is forbidden.

3.3. Automatic runtime validation

This approach is non-intrusive, runtime approach for validating the usage of STL. This approach is based on the `gdb` debugger tool. We set breakpoints to specific

locations in the library and when the execution is stopped we can evaluate the intervals, iterators, the state of the containers, etc. After evaluation, we can continue the execution or indicate an error.

We have developed gdb script for validating the usage of STL. For instance, the hereinafter `checkinterval` script validates if an interval is sorted based on `operator<`.

```
define checkinterval
  if $argc == 2
    set $first = $arg0
    set $last = $arg1
    set $l = 1
    while $first._M_current < ($last._M_current - 1) && $l == 1
      set $act = $first
      set ++$first._M_current
      if  *($first._M_current) < *($act._M_current)
        set $l = 0
      end
    end
    if $l == 1
      printf "PASSED\n"
    else
      printf "ERROR, interval not sorted\n"
    end
  end
end
```

We can set breakpoints to our specific library implementation, for instance in gdb we can define two breakpoints that stop the execution when one of the overloaded `std::unique` algorithm is invoked:

```
break /usr/include/c++/5.3.1/bits/stl_algo.h:992
break /usr/include/c++/5.3.1/bits/stl_algo.h:1022
```

After evaluation at specific points of the STL implementation one can continue the execution. However, the gdb scripts are not sophisticated properly, so we have developed an automation tool. This tool has many goals:

- Preparation: launches the debugger with logging and added breakpoints
- Data processing: logging the debugger's output and process it. The important data must be filtered and forward it to the analysing component.
- Administration: this component maintains the state of objects because the STL object's may change between two breakpoints. For instance:
  - Iterator administration: keep iterators' data update: name, validity, referred memory address.

  - Container administration: keep containers' state update: type, capacity, size, iterators etc.

- Analysis: this component evaluates the maintained objects in an iterative way. It drives the the debugger, the data processing component and passes input for the logging component and finally makes the gdb to continue the execution.

- Logging: the framework returns a comprehensive list of every critical STL-related code snippet. Every item of this list appears in a logfile and describes if the test is fails, passes or warns a potential misuages.

The tool detects problems related to algorithms: using `find` and `count` algorithms on sorted containers, using algorithms with special precondition (e.g. `lower_bound` and misusages of copying and removing algorithms. This tool is able to detect iterator invalidation, as well. Problems with iterator conversion also can be realized. Special instantiation of containers (e.g. COAP and `vector<bool>`) is discovered, as well.

4. 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 or affect the performance. In this paper we argue for approaches to make the STL's usage safer. With our different techniques one can checks if the library's usage is incorrect. We present three alternative implementations: metaprograms that evaluated by the C++ compiler, a Clang-based static analysis tool, and a gdb-driven framework that searches for misusages at runtime without any modification in the library implementation.

References

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

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

[3] Dévai, G., Pataki, N.: *A Tool for Formally Specifying the C++ Standard Template Library*, Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica, **31** (2009), pp. 147–166.

[4] Horváth, G., Pataki, N.: *Clang matchers for verified usage of the C++ Standard Template Library*, Annales Mathematicae et Informaticae **44** (2015), pp. 99–109.

[5] Lopes, B. C., Auler, R.: "Getting Started with LLVM Core Libraries", Packt Publishing (2014).

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

[7] Pandey, M., Sarda S.: "LLVM Cookbook", Packt Publishing (2015).

[8] Pataki, N.: *C++ Standard Template Library by Ranges*, in Proc. of the 8th International Conference on Applied Informatics (ICAI 2010) Vol. 2., pp. 367–374.

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

[10] Pataki, N.: *Compile-time advances of the C++ Standard Template Library*, Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica **36** (2012), pp. 341–353.

[11] Pataki, N.: *Safe Iterator Framework for the C++ Standard Template Library*, Acta Electrotechnica et Informatica **12(1)** (2012), pp. 17–24.

[12] Pataki, N., Porkoláb, Z.: *Extension of Iterator Traits in the C++ Standard Template Library*, in Proc. of the Federated Conference on Computer Science and Information Systems (2011), pp. 919–922.

[13] Pataki, N., Szűgyi, Z., Dévai, G.: *Measuring the Overhead of C++ Standard Template Library Safe Variants*, Electronic Notes in Theoret. Comput. Sci., **264(5)** (2011), pp. 71–83.

[14] Porkoláb, Z.: *Functional Programming with C++ Template Metaprograms*, in Proc. of Central European Functional Programming School, Revised Selected Lectures, Lecture Notes in Computer Science, **6299** (2009), pp. 306–353.

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

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

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