Towards soundness examination of the C++ Standard Template Library

Norbert Pataki, Zoltán Porkoláb, and Zoltán Istenes
Eötvös Loránd University, Faculty of Informatics
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
email: patakino@elte.hu, gsd@elte.hu, istenes@inf.elte.hu

Abstract

The Standard Template Library (STL) is an essential part of professional C++ programs. STL is a type-safe template library which is based on the generic programming paradigm and helps to avoid some possible dangerous C++ constructions. With its usage, increases the efficiency, safety and quality of the code. However, the C++ standard gives the definition of STL by informal description that can lead to ambiguous explanations. In this paper we create a formal specification of STL. With these instruments we prove the soundness of STL implementations, libraries over the STL and programs that use STL. Our solution is based on the Hoare-method, that we extend to describe the claims of the generic programming paradigm.

Keywords: program correctness, C++, STL

1. Introduction

Generic programming is a software development technique generalizing software components so that they can be easily reused. Using this approach we can greatly reduce the complexity of a software library. The generic programming most typical appearance is the C++ Standard Template Library (STL). Generative programming [6] is mightly based on the generic programming. The aim of the generative programming is automatic code generation.

Every programmer wants to avoid bugs in own code. Usually the test cases cannot cover the all of the potentials and, of course, cannot prove the code's soundness in mathematical way. But there are several methods to prove it like a mathematical theorem, what is more, there are some opportunity to do it automatically. One of the most widely used method is the Hoare-method [9]. It is more important to prove a widely-used library's soundness, because the usage of the library appears in thousands of programs. Because in the last years Java and C# were extended to be qualified to generic programming, more important to make a mathematic based soundness proving to this paradigm.

The STL framework works well for years, the codes that use STL are very common. Alas, the standard does not give a formal definition of the components. This can lead to ambiguous explanations. For example, the standard does not define well, what the order is in a multiset, where the keys are equivalent. Another example is the for_each algorithm: it is a non-modifying algorithm, but it is able to modify the object. The problem is more important when the application runs in serious situation. The lack of formal definition is more serious if we check a STL's new implementation: very hard to prove soundness of an implementation without framework.

The usage of STL does not mean, that the programs do not contain mistakes. For example, vector's operator [] does not check if the index is correct. The vector has a member function called at(), this throws an exception, when the index is not correct. When an end() iterator or an iterator which does not point anywhere, is dereferenced, it can also cause undefined behaviour. These mistakes can be

Supported by GVOP-3.2.2.-2004-07-0005/3.0

avoided by a mathematical foundation.

In view of the relationship between generative and generic programming, it is important to make correct generic programs.

2. The architecture of the Standard Template Library

STL is part of the C++ standard library. It consists of many useful template data structures that we can instantiate with our types, algorithms, that work together with these containers [3, 4, 15]. They can work well because the containers offer iterators, and the algorithms work with the iterators. This construction supports that we can extend the library with new containers and/or algorithms.

The standard guarantees the complexity of the operations, that every implementation has to perform.

2.1 Containers

STL offers several template containers: sequence containers (vector, deque, list) and associative containers (set, multiset, map and multimap). There are three data structures (stack, queue and priority_queue) in the STL, they are the adapters. They have no iterators, and they work over a sequence container with a short interface.

The containers hold objects of the same type, other words a container is a sequence of objects.

Every container has some member functions, so they can work without the algorithms. For example, every sequence container has a push_back() member function, and every container offers the size() member function.

One of the most widely-used containers is the vector, because it guarantees a more powerful, and more secure application, than the lowly C-style arrays. It manages the memory for own, so it can be grow, when it needs, does not cause memory-leaks, and we can avoid a lot of problems.

The associative containers hold objects ordered, so they need a compare function on the keytype. Most STL implementations use red-black trees for the associative containers [5]. This guarantees efficiency.

2.2 Iterators

Iterators are the generalization of pointers [13]. Iterators have two essential roles in the STL. Firstly, the iterators guarantee access for the elements in a container. Secondly, they enable the containers and algorithms to work together. Iterators have some elementary functions (e.g. operator * – dereferences an iterator, operator ++ – makes the iterator step to next element in the container, operator == – returns true if two iterators point at the same element, etc.).

Every container offers iterators (except for the adapters), but the belonging different container iterators have not the same set of operations. Iterators have a hierarchy. We can speak about: InputIterators, OutputIterators, ForwardIterators, BidirectionalIterators and RandomAccessIterators.

Every container offers four different iterator types: iterator, const_iterator, reverse_iterator and const_reverse_iterator.

2.3 Algorithms

The STL offers about sixty algorithms, implemented as template functions. The algorithms can be classified: there are the non-modifying algorithms (for_each, find, etc.), modifying algorithms (reverse, replace_if, etc.), ordered ranges algorithms (merge, binary_search, etc.), set operations (set_intersection, etc.), heap operations (make_heap, sort_heap, etc.), minimum and maximum algorithms (min_element, etc.) and the permutation algorithms (next_permutation, etc.).

3. Tools for safe coding

In this section we give an overview on existing tools and methods of either writing safe code or proving the correctness of code.

3.1 Algebraic specification

This is a formal tool for specificate containers and algorithms (e.g. [2]). It supports the soundness proving, but for it we must apply structural induction, which works very hard at STL's size. Gibbons works on the STL's algebraic specification, and he worked to make the specification in Haskell (functional) programming language [8].

3.2 Hoare-method

The Hoare-method is an old method to prove the soundness of abstract programs. It uses pre- and postconditions as a state of a program. The method defines rules for the program constructions. With these rules the procedure is sound and complete. But, of course it has some limitations. The Hoare-method works on an abstract machine, there are no pointers. Assigments execute properly, that is not true in C++. There are no overloaded functions or constructors. Note that the Hoare-method has an extension for multithreaded environments [14].

3.3 Concept checking

Concept checking is programming method, what able to use the template parameters in a safety way [11]. C++ uses lazy instantion to the template parameters [16, 17, 19]. This means, if a function uses an operation on the template type, it is checked, when the call of function compiles. If this function not used, it will not compile. Therefore if the operation does not exist on the type, it can be problem. Other languages (e.g. Ada) check is it at the instantion, and give an error message, if the operation does not exist, not depends on usage. This is safer, but not so flexible as lazy instantion.

The Boost has a Concept Checking library [18], which able to use STL more easier, because it make possible to work with C++, like Ada. The implementation of the library use template metaprogramming techniques [1]. We can read about the improvement of this library [21].

3.4 STLlint

STLlint [10] is a static checking tool that tries to find semantic errors in the usage of the STL. STLlint analyses user programs and reports error messages for any construct it cannot prove correctness with respect to the semantics of the STL. STLlint is an online program, everybody can send a program over the internet, and gets back the message. It is useful software, but there are a lot of imperfection: several known errors are uncaught by STLlint.

3.5 Tecton

Tecton is a specification language for describing and using abstract concepts in formal software development [12, 20]. It is made by David R. Musser and others. Algebraic specification is used by the Tecton. From the formal specification Tecton makes (not necessarily constructive way) Prolog programs (*Interface Engine*). This cannot be used directly. Between the two parts there is the *Run-Time Analysis Oracle – RAO*. This can bind IEs with C++ generic algorithm being verified. The RAO works with C++ classes. The specification class must be implemented with 3 member functions: precond(), that checks the precondition, post_update(), that simulize the application of the function, and postcond(), that checks the postcondition. The implementation of this class is not trivial.

MELAS is the MEta-Level integrated Analysis System, which was developed for the analysis of C++ template-based generic algorithms. It allows the user to verify and test C++ generic programs at a meta-level directly. MELAS is implemented on top of the GNU C/C++ debugging system (gdb). This can link IE and RAO. The verification process can work automatically or manually. The debugger executes this program, that use specification object, in a loop. When the loop ends, the verification ends.

This is an interesting solution, but this cannot prove soundness of a program or a library that use STL. And this approach is not easy.

4. Extension of Hoare–method

Before specificate the STL, we show our formalism, that based on the pre- and post-conditions: $\{P\}S\{Q\}$, where P and Q is a first-order logic expression, and S is a program, in this case S will be syntactical good C++ instruction or instructions. P is called precondition, Q is called postcondition. We won't write out the `std` namespace.

We had to extend the Hoare's formal tools to be able to specificate the STL and important C++ constructions.

In C++ we can use an object (for example, an iterator or an int variable) before it get value. This state is represented by a ? symbol, for example $\{i = ?\}$.

The $Undef$ symbol means that program effect is undefined. It means there will be a problem, maybe a runtime error, or something else.

An order can throw an exception, so $Exc(a)$ means is the program throws an exception of type a.

Algorithms can substitute loops, so algorithms often mean an sequential orders. $SEQ(f(a), g(b))$ means the sequential execution of $f(a)$ and $g(b)$.

Types have general invariants, so do the STL containers. $\mathcal{I}$ means the class or typeinvariant.

The assignment in C++ is dissimilar from other languages. When we write `=` in C++, it can be two different functions depend on the context. The copy constructor runs in the case, when we create an object and immediate give it value, (for example: `int i=0;`). The assignment operator runs, when we use `operator=` to a defined object, for example: `it=v.begin();`, where it is an iterator. For an arbitrary class this functions can work improperly. So, we cannot assume, this functions are correct. In the formalism, we will use 2 symbol, one for the copy constructor (*cpctor*), and one for the assignment operator.

The STL containers are abstract sequences according to their traverses, so a container named v can describe this way: $v =< v_1, v_2, \ldots, v_n >$. An empty container is formalized:$v =<>$. The vector index starts with 0, rather a vector write this way:
$v =< v_0, v_1, \ldots, v_n >$. The container objects can be constant and non-constant. The const member functions can be applied to a constant and a non-constant too, but a non-const member function do not allow to apply to a const object. C++ supports overloading on const. We will describe x as a constant container: $const(x)$, and y as a non-const container: $\neg const(y)$.

Let us show some examples: from the vector class:

$$\{v =< v_0, v_1, \ldots, v_n >\}$$

$$s = v.size();$$

$$\{s = n + 1 \wedge v =< v_0, v_1, \ldots, v_n >\}$$

$$\{v =< v_0, v_1, \ldots, v_n > \wedge i \leq n \wedge \neg const(v)\}$$

$$T\ t = v[i];$$

$$\{t = t.vpctor(v_i) \wedge v =< v_0, v_1, \ldots, v_n > \wedge \neg const(v)\}$$

$$\{v =< v_0, v_1, \ldots, v_n > \wedge i \leq n \wedge const(v)\}$$

$$T\ t = v[i];$$

$$\{t = t.vpctor(v_i) \wedge v =< v_0, v_1, \ldots, v_n > \wedge const(v)\}$$

This two last specification shows that `operator[]` is overloaded with const, and this two functions are not the same:

$$\{v =< v_0, v_1, \ldots, v_n > \wedge (*it = v.end \vee *it = ?) \wedge$$

$$type(it) = vector < T > \wedge \neg const(v) \wedge \neg rev\_it(it)\}$$

$$v.erase(it);$$

$$\{Undef\}$$

This is a part of the set (associative container) specification:

$$\mathcal{I} = \{LTC(T, operator <)\}$$

$$\{s =< x_1, x_2, \ldots, x_n > \wedge$$

$$\wedge \neg const(s) \wedge \exists j : 1 \geq j \geq n : a = x_j$$

$$pair < set < T >:: iterator, bool >$$

$$p = s.insert(a);$$

$$s =< x_1, x_2, \ldots, x_n > \wedge \neg const(s) \wedge$$

$$\wedge p.second = false \wedge$$

$$\wedge * (p.first) = x_j \wedge \neg rev\_it(p.first)$$

$$\wedge const\_it(p.first) \wedge type(p.first) = set < T >$$

Iterators have more difficult formalism. The most important attribute is where points an iterator: $*it = v_i$ means that, an iterator named it, points to v_i. There are const iterators, and iterators: $const\_it(it)$ describes a const_iterator, and $\neg const\_it(it)$ describes it as an iterator. A reverse iterator can be described this way: $rev\_it(it)$ and a not reverse iterator implicitly $\neg rev\_it(it)$. Iterators have a hierarchy. If an iterator is RandomAccessIterator, that will be formalized: $cat(it) = Ran$. Implicitly $cat(it) = Bi$ means it is a BidirectionalIterator, $cat(it) = For$ means it is a ForwardIterator, $cat(it) = In$ means it is an InputIterator, $cat(it) = Out$ means it is an OutputIterator. Of course, we have to describe the iterator operations, too.

The iterator hierarchy could be described this way:

$$cat(it) = Ran \Rightarrow cat(it) = Bi$$

$$cat(it) = Bi \Rightarrow cat(it) = For$$

$$cat(it) = For \Rightarrow cat(it) = In$$

$$cat(it) = In \Rightarrow cat(it) = Out$$

Containers determine their iterator category:

$$type(it) = vector < T > \Rightarrow cat(it) = Ran$$

$$type(it) = list < T > \Rightarrow cat(it) = Bi$$

etc.

Let assume, we define a new `list<T>::iterator`. This can be described:

$$\{true\}$$

$$list < T >:: iterator\ i;$$

$$\{type(i) = list < T > \wedge \neg rev\_it(i) \wedge$$

$$\neg const\_it(i) \wedge *i = ?\}$$

Algorithms do not require new formal tools.

$$\{x =< x_1, x_2, \ldots, x_n > \wedge$$

$$cat(it1) = In \wedge cat(it2) = In \wedge$$

$$*it1 = x_k \wedge *it2 = x_l \wedge$$

$$\neg rev\_it(it1) \wedge \neg rev\_it(it2)\}$$

$$for\_each(it1, it2, f);$$

$$\{SEQ(f(x_k), f(x_{k+1}), \ldots, f(x_{l-1})) \wedge$$

$$x =< x_1, x_2, \ldots, x_n > \wedge$$

$$cat(it1) = In \wedge cat(it2) = In \wedge$$

$$*it1 = x_k \wedge *it2 = x_l \wedge$$

$$\neg rev\_it(it1) \wedge \neg rev\_it(it2)\}$$

After the specification of the STL, the method is able to prove correctness of a program, library or an implementation.

5. Experiments, Conclusions and Future works

Meanwhile we created the formal specification, find some interesting observations. Notwithstanding the standard describes the for_each, the find_if, etc. algorithms as non-modifying algorithms, we can change the container with these algorithms with a proper function. However, this problem cannot be caught by the STLlint.

In this paper, we showed a potential formalism for the C++ STL. This formalism supports soundness proving STL implementations, libraries over the STL, and programs, that use the STL. We specificated a part of the STL and showed some observations. The method can be extend for multithreaded systems.

There are a lot of work to make this method work properly: it would be nice to make a program that made this check automatically. The STLlint works, but it cannot prove the soundness, and there are some scantiness. Our method able to get more problem, than STLlint.

AspectC [22] is an extension to the aspect-oriented programming (AOP). Pre- and post-conditions can be checked with this paradigm. Unfortunately, AspectC does not support the weaving in template instances, yet.

References

[1] Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)

[2] Sergio Antoy, Dick Hamlet: Automatically Checking an Implementation against Its Formal Specification (1999)

[3] Matthew H. Austern: Generic Programming and the STL. Addison-Wesley (1999)

[4] Ulrich Breymann: Designing Components with C++ STL: A New Approach to Programming, Addison-Wesley (1998)

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: Introduction to Algorithms, McGraw Hill (1990)

[6] Krzysztof Czarnecki, Ulrich W. Eisenecker: Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)

[7] David L. Deflefs: An Overview of the Extended Static Checking System, Proc. of The First Workshop on Formal Methods in Software Practice (1996)

[8] Jeremy Gibbons: Patterns in Datatype-Generic Programming, In J. Striegnitz, editor, DPCOOL (2003).

[9] C. A. R. Hoare: An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576-580, 1969.

[10] Douglas Gregor: STLlint `http://www.cs.rpi.edu/~gregod/STLlint/`

[11] Jaakko Jarvi, Jeremiah Willcock, Andrew Lumsdaine: Concept-Controlled Polymorphism. In proceedings of GPCE 2003, LNCS 2830, pp. 228-244.

[12] D. Kapur, D. R. Musser, X. Nie: An Overview of the Tecton Proof System, Theoretical Computer Science, Vol. 133, pp. 307-339 (1994)

[13] Scott Meyers: Effective STL. Addison-Wesley (2001)

[14] S. Owicki, D.Gries: An axiomatic proof technique for parallel programs, Acta Informatica 6, pp. 319-340, (1976)

[15] Musser and Stepanov: Generic Programming Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation 1989

[16] Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)

[17] Bjarne Stroustrup: The Design and Evolution of C++. Addison-Wesley (1994)

[18] The Boost concept checking library. `http://www.boost.org/libs/concept_check/concept_check.htm`

[19] David Vandevoorde, Nicolai M. Josuttis: C++ Templates: The Complete Guide. Addison-Wesley (2002)

[20] Changqing Wang and David R. Musser: Dynamic Verification of C++ Generic Algorithms, Software Engineering Vol. 23. (5) pp. 314-323 (1997)

[21] István Zólyomi, Zoltán Porkoláb: Towards a general template introspection library Generative Programming and Component Engineering LNCS Vol.3286(2004) pp. 266-282.

[22] AspectC, `http://aspectc.org/`