Transparent functors for the C++ Standard Template Library*

Gábor Horváth
xazax.hun@gmail.com

Norbert Pataki
patakino@elte.hu

ELTE Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary

Abstract

The *C++ Standard Template Library (STL)* is the most popular library based on the *generic programming paradigm*. The STL is widely used, because the library is part of the C++ Standard. It consists of many useful generic data structures (like list, vector, map, etc.) and generic algorithms (such as for_each, sort, find, etc.) that are fairly irrespective of the used container. Iterators bridge the gap between containers and algorithms. As a result of this layout the complexity of the library is greatly reduced and we can extend the library with new containers and algorithms simultaneously. *Function objects* (also known as *functors*) make the library much more flexible without significant runtime overhead. They parametrize user-defined algorithms in the library, for example, they separate the comparison in the ordered containers or define a predicate to find. Programmers typically had to duplicate the type of contained objects before the C++14 standard when template functors have been used. Improper duplication may result in runtime errors. C++14 introduces the notion of transparent functors. These functors are able to avoid the duplication of referred types. When transparent functors are in use, template arguments can be deduced by the compiler. In this paper we argue for the usage of transparent functors. We developed a tool that can transform the source code to take advantage of transparent functors. Our tool is part of the latest (5.0) Clang release. We analyzed when this transformation is correct and what are the template parameters when the transformation cannot be used. The improper usage of transparent functors results in incorrect code snippets.

1 Introduction

The *C++ Standard Template Library* (STL) was developed with a *generic programming* approach [Aus98]. In this way containers are defined as class templates and many algorithms can be implemented as function templates.

*Supported by the NKP-17-4 New National Excellence Program of the Ministry of Human Capacities

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

Furthermore, algorithms are implemented in a container-independent way [Mus88], so one can use them with different containers [Str00]. C++ STL is widely used because it is a very handy, standard C++ library that contains beneficial containers (like list, vector, map, etc.) and large number 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 [Bec01]. The expression problem [Tor04] is solved with this approach. The STL also includes adaptor types which transform standard elements of the library for a different functionality [Ale01].

However, the usage of C++ STL does not mean bugless or error-free code [Dev09]. Contrarily, incorrect application of the library may introduce new types of problems [Por07].

One of the problems is that the error diagnostics are usually complex, and it is very hard to figure out the cause of a program error [Zol01, Zol04]. The violation of special precondition requirements (e.g. sorted ranges) is not tested, but results in runtime bugs [Gre06, Pat11b]. 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* [Dev07]. 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 physically. Since, for example the `remove` algorithm does not actually remove any element from a container [Mey03].

Most of the properties are checked at compilation time. For example, the code does not compile if one a uses sort algorithm with the standard list container, inasmuch as list's iterators do not offer random accessibility [Jar06]. 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 [Pat06].

Unfortunately, there is still a large number of properties that are tested neither at compilation-time nor at run-time. Observance of these properties is in the charge of the programmers.

Functor objects make the STL more flexible as they enable the execution of user-defined code parts inside the library [Pat11a]. Basically, functors are usually simple classes with an `operator()`. Inside the library `operator()`s are called to execute user-defined code snippets. This can call a function via a pointer to a function or an actual `operator()` in a class. Functors are widely used in the STL because they can be inlined by the compiler and they cause no runtime overhead in contrast to function pointers. Moreover, in the case of functors extra parameters can be passed to the code snippets via constructor calls.

Functors can be used in various roles: they can define predicates when searching or counting elements, they can define comparison for sorting elements and properly searching, they can define operations to be executed on elements.

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.

C++14 introduces the so-called transparent functors. They enable to remove unnecessary code duplication in template arguments. These functors are superior to work with the associative containers.

The rest of this paper is organized as follows. In section 2 we show what are the transparent functors and what is the motivation for their usage.

2 Transparent functors

In this section we present the notation of transparent functors. We show the motivation behind them, and the problems that these functors overcome.

C++14 introduces the so-called *transparent functors*. These are specializations of the existing functors and they take advantage of the C++'s parameter deduction to avoid code duplication. Let us consider the following code snippet: `std::multiset<std::string, std::greater<std::string>> s;`. If the `multiset` contains `std::string` objects, it is code duplication that the container compares the objects as strings. However, `std::greater` is a class template, so the template argument has to be specified.

Code duplication is typically a bad pattern in programming. Duplicated code snippets are prone to become inconsistent. Let us consider the following code snippet:

```
std::map<double, std::string, std::greater<int>> s;
```

```
s[ 1.4 ] = "Hello";
s[ 1.9 ] = "World";

std::cout << s.size(); // 1
```

This code snippet contains a use-case when the template arguments have become inconsistent because of maintenance problems. Probably the keys of the map were `int`s but later changed to `double`s, while the comparison has not been touched. This compiles and the behaviour is strange. With the usage of transparent functors it can be fixed:

```
std::map<double, std::string, std::greater<>> s;
s[ 1.4 ] = "Hello";
s[ 1.9 ] = "World";

std::cout << s.size(); // 2
```

This phenomenon indicates the appearance of C++14's transparent functors which are the specializations of the template functor classes. They are specialized to `void`. These specializations consist of an `operator()` that is a template member function and it is able to use parameter deduction.

It can be used with algorithms as well:

```
std::vector<int> v;
// ...
std::sort( v.begin(), v.end(), std::greater<>() );
```

Since the C++14 standard the transparent functors are the preferred ones.

3 The Clang compiler infrastructure

Clang is an open source compiler for C/C++/Objective-C that is built on the top of the Low Level Virtual Machine (LLVM) compiler infrastructure. It is a rapidly evolving project supported by Apple, Microsoft and Google. Clang is widely used for the static analysis of C/C++ source code [Bab17]. The refactoring of existing code is important aspect of static analysis [Maj14, Sza14]. Refactoring tools are widely-used nowadays because agile methods (like DevOps) emphasize the importance of refactoring.

One of the advantages of Clang is its modular architecture. One can use the parser as a library to build tools. The popularity of this compiler also implies that it is well tested. The parser module is a hand-written recursive descendant parser that does the parsing and the typing at the same time. So in Clang the result of the parsing is a typed Abstract Syntax Tree (AST).

Clang has an embedded domain specific language (EDSL) to match the AST for simpler checks that do not require symbolic execution. This EDSL is written in a declarative functional style. The EDSL is called ASTMatchers [Hor15]. The language contains primitive matcher expressions that can be composed to build more complex patterns. Those primitive matchers can be classified into three categories: node matchers, narrowing matchers, and traversal matchers. The node matchers only match a certain type of AST node (e.g. a function call.) The narrowing matchers can match on properties of the AST nodes, to make the pattern more strict (e.g. the number of arguments provided for a function call.) The traversal matchers can specify the relationship of two AST nodes in the pattern (e.g. one node to be the child of the other.)

There is an open source tool that is mainly used and developed by Google called Clang Tidy. This tool consists of many useful ASTMatcher patterns that can detect code smells. In some cases Clang Tidy also offers automated source code transformation to fix them.

The Clang Tooling library also provides users with an infrastructure to handle the compilation database. That database consists of the compilation parameters for each file. Such database can be generated using external tools (that are not part of the compiler). Using this library one can easily load a compilation database and look up the corresponding compilation arguments for a file. This is a useful facility because the semantics of a software might depend on those arguments.

4 Our tool

The tool we developed is an extension of Clang Tidy which was eventually included in the official set of checks.

In the C++ source code there can be several source locations where a type is mentioned. In the Clang AST one type is only created once, and its memory address can be used as identity. During the parsing, every mention of a certain type will create a *TypeLoc* AST node that refers to the type and contains the source location where it was mentioned.

In order to find the uses of non-transparent functors that could be refactored first we need to define a pattern that describes how such functors look like.

```
const auto NonTransparentFunctors = classTemplateSpecializationDecl(
      unless(hasAnyTemplateArgument(refersToType(voidType()))),
      hasAnyName("::std::plus", ... , "::std::bit_not"));
```

Those functors are class template specializations that a non-void template argument and their name matches one of the predefined functors from the STL. The next step is to find all source locations that mention these types.

```
loc(qualType(unless(elaboratedType()),
             hasDeclaration(NonTransparentFunctors)))
```

We do not permit to match the locations that are elaborated types. We would match the same location twice (both the *std::greater* and *greater*) had we permitted elaborated types since the declaration of an elaborated type is the same as its inner type.

It would be straightforward to replace every mention of a non-transparent functor with the corresponding transparent one. This solution however can alter the meaning of the code in some cases.

```
std::map<const char *, std::string, std::greater<std::string>> s;
```

In the example above removing the template parameter of `greater` would modify the semantics of the code. The original version uses lexicographical ordering while the modified one would compare the memory addresses of the strings instead of their value. Even though this pattern is rare, we would like to preserve the semantics of the code in a source to source translation.

```
loc(qualType(
      unless(elaboratedType()),
      hasDeclaration(classTemplateSpecializationDecl(
          unless(hasAnyTemplateArgument(templateArgument(refersToType(
              qualType(pointsTo(qualType(isAnyCharacter()))))))),
          hasAnyTemplateArgument(
              templateArgument(refersToType(qualType(hasDeclaration(
                                  NonTransparentFunctors))))))))
```

We want to exclude those cases when one of the sibling template parameter of the non-transparent functor is a pointer to a character type.

There are some other rules when the transformation can not be done. When some part of the AST that we matched is the result of macro expansions we can not rewrite the source code. The reason is that rewriting a macro definition might break other uses of the macro.

The source transformation code is straightforward, the source range that represents the template argument of the non-transparent functor needs to be removed.

We ran the tool on the LLVM codebase and found five usages of non-transparent functors that could be replaced by transparent ones. The tool was able to do the transformation in each case automatically and the resulting code compiled correctly. LLVM has an extensive test suite, our changes did not cause any regressions.

In the future we might extend this check to also check for user written functors that could be transformed into a transparent functor.

5 Conclusion

The C++ Standard Template Library is a popular library based on the generic programming paradigm. Functors are able to execute user-defined code snippets in the library without significant overhead. C++14 introduces transparent functors to avoid unnecessary code duplication.

In this paper we presented the advantages of transparent functors. We have developed a tool that is able to refactor code to use transparent functors. The tool is part of the Clang compiler infrastructure. It works reliably on industrial scale projects. We presented the background and the approach of this tool.

References

[Ale01] A. Alexandrescu. *Modern C++ Design.* Addison-Wesley, 2001.

[Aus98] M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Addison-Wesley, 1998.

[Aus96] M. H. Austern, R. A. Towle, A. A. Stepanov. Range partition adaptors: a mechanism for parallelizing STL. *ACM SIGAPP Applied Computing Review*, 4(1):5–6, 1996.

[Bab17] B. Babati, N. Pataki. Analysis of Include Dependencies in C++ Source Code. *Annals of Computer Science and Information Systems*, 13:149–156, 2017.

[Bec01] T. Becker. STL & generic programming: writing your own iterators. *C/C++ Users Journal*, 19(8):51–57, 2001.

[Dev09] G. Dévai, N. Pataki. A tool for formally specifying the C++ Standard Template Library. *Ann. Univ. Sci. Budapest., Comput.*, 31:147–166, 2009.

[Dev07] G. Dévai, N. Pataki. Towards verified usage of the C++ Standard Template Library. *Proc. of The 10th Symposium on Programming Languages and Software Tools (SPLST)*, 360–371, 2007.

[Eng00] D. Engler, B. Chelf, A. Chou, S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. *Proc. of the 4th conference on Symposium on Operating System Design & Implementation* 4:1, 2000.

[Gre06] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine. Concepts: linguistic support for generic programming in C++. *Proc. of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA 2006)*, 291–310, 2006.

[Hal02] S. Hallem, B. Chelf, Y. Xie, D. Engler. A system and language for building system-specific, static analyses. *Proc. of the ACM SIGPLAN 2002 conference on Programming language design and implementation (PLDI '02)*, 69–82, 2002.

[Hor15] G. Horváth, N. Pataki. Clang matchers for verified usage of the C++ Standard Template Library, *Annales Mathematicae et Informaticae*, 44:99–109, 2015.

[Jar06] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek. Algorithm specialization in generic programming: challenges of constrained generics in C++. *Proc. of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2006)*, 272–282.

[Maj14] V. Májer, N. Pataki. Concurrent object construction in modern object-oriented programming languages. *Proc. of the 9th International Conference on Applied Informatics*, 2:293–300, 2014.

[Mey03] S. Meyers. *Effective STL.* Addison-Wesley, 2003.

[Mus88] D. R. Musser, A. A. Stepanov. Generic Programming, *Proc. of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation, Lecture Notes in Comput. Sci.*, 358:13–25, 1988.

[Pat11a] N. Pataki. C++ Standard Template Library by Safe Functors. *Proc. of The 8-th Joint Conference on Mathematics and Computer Science, Selected Papers*, 357–368, 2011.

[Pat06] N. Pataki, Z. Porkoláb, Z. Istenes. Towards Soundness Examination of the C++ Standard Template Library. *Proc. of Electronic Computers and Informatics, ECI 2006* 186–191, 2006.

[Pat11b] N. Pataki, Z. Szűgyi, G. Dévai. Measuring the Overhead of C++ Standard Template Library Safe Variants. *Electronic Notes in Theoretical Computer Science*, 264(5):71–83, 2011.

[Pir08] P. Pirkelbauer, S. Parent, M. Marcus, B. Stroustrup. Runtime Concepts for the C++ Standard Template Library. *Proc. of the 2008 ACM Symposium on Applied Computing*, 171–177, 2008.

[Por07] Z. Porkoláb, Á. Sipos, N. Pataki. Inconsistencies of Metrics in C++ Standard Template Library. *Proc. of 11th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering QAOOSE Workshop, ECOOP 2007, Berlin*, 2–6, 2007.

[Rep95] T. Reps, S. Horwitz, M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. *Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 49–61, 1995.

[Str00] B. Stroustrup. *The C++ Programming Language - Special Edition.* Addison-Wesley, 2000.

[Sza14] Cs. Szabó, M. Kotul, R. Petruš. A closer look at software refactoring using symbolic execution. *Proc. of the 9th International Conference on Applied Informatics*, 2:309–316, 2014.

[Tor04] M. Torgersen. The Expression Problem Revisited – Four New Solutions Using Generics. *Proc. of European Conference on Object-Oriented Programming(ECOOP) 2004, Lecture Notes in Comput. Sci.*, 3086:123–143, 2004.

[Zol01] L. Zolman. An STL message decryptor for visual C++. *C/C++ Users Journal*, 19(7):24–30, 2001.

[Zol04] 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, 2004.

[Xu10] Z. Xu, T. Kremenek, J. Zhang. A Memory Model for Static Analysis of C. Programs. *Proc. of ISoLA'10 4th iternational conference on Leveraging applications of formal methods, verification, and validation*, 1:535–548, 2010.