Concepts: Linguistic Support for Generic Programming in C++

Douglas Gregor
Indiana University
dgregor@osl.iu.edu

Jaakko Järvi
Texas A&M University
jarvi@cs.tamu.edu

Jeremy Siek
Rice University
Jeremy.G.Siek@rice.edu

Bjarne Stroustrup
Texas A&M University
bs@cs.tamu.edu

Gabriel Dos Reis
Texas A&M University
gdr@cs.tamu.edu

Andrew Lumsdaine
Indiana University
lums@osl.iu.edu

Abstract

Generic programming has emerged as an important technique for the development of highly reusable and efficient software libraries. In C++, generic programming is enabled by the flexibility of templates, the C++ type parametrization mechanism. However, the power of templates comes with a price: generic (template) libraries can be more difficult to use and develop than non-template libraries and their misuse results in notoriously confusing error messages. As currently defined in C++98, templates are unconstrained, and type-checking of templates is performed late in the compilation process, i.e., after the use of a template has been combined with its definition. To improve the support for generic programming in C++, we introduce *concepts* to express the syntactic and semantic behavior of types and to constrain the type parameters in a C++ template. Using concepts, type-checking of template definitions is separated from their uses, thereby making templates easier to use and easier to compile. These improvements are achieved without limiting the flexibility of templates or decreasing their performance—in fact their expressive power is increased. This paper describes the language extensions supporting concepts, their use in the expression of the C++ Standard Template Library, and their implementation in the ConceptGCC compiler. Concepts are candidates for inclusion in the upcoming revision of the ISO C++ standard, C++0x.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

General Terms Design, Languages

Keywords Generic programming, constrained generics, parametric polymorphism, C++ templates, C++0x, concepts

1. Introduction

The C++ language [25, 62] supports parametrized types and functions in the form of *templates*. Templates provide a unique combination of features that have allowed them to be used for many different programming paradigms, including Generic Programming [3,44], Generative Programming [11], and Template Metaprogramming [1, 66]. Much of the flexibility of C++ templates comes from their unconstrained nature: a template can perform any operation on its template parameters, including compile-time type computations, allowing the emulation of the basic features required for diverse programming paradigms. Another essential part of templates is their ability to provide abstraction without performance degradation: templates provide sufficient information to a compiler's optimizers (especially the inliner) to generate code that is optimal in both time and space.

Consequently, templates have become the preferred implementation style for a vast array of reusable, efficient C++ libraries [2, 6, 14, 20, 32, 54, 55, 65], many of which are built upon the Generic Programming methodology exemplified by the C++ Standard Template Library (STL) [42, 60]. Aided by the discovery of numerous *ad hoc* template techniques [28, 46, 56, 66, 67], C++ libraries are becoming more powerful, more flexible, and more expressive.

However, these improvements come at the cost of implementation complexity [61, 63]: authors of C++ libraries typically rely on a grab-bag of template tricks, many of which are complex and poorly documented. Where library interfaces are rigorously separated from library implementation, the complexity of implementation of a library is not a problem for its users. However, templates rely on the absence of modular (separate) type-checking for flexibility and performance. Therefore, the complexities of library implementation leak through to library users. This problem manifests itself most visibly in spectacularly poor error messages for simple mistakes. Consider:

```
list<int> lst;
sort(lst.begin(), lst.end());
```

Attempting to compile this code with a recent version of the GNU C++ compiler [17] produces more than two kilobytes of output, containing six different error messages. Worse, the errors reported provide line numbers and file names that point to the implementation of the STL sort() function and its helper functions. The only clue provided to users that this error was triggered by their own code (rather than by a bug in the STL implementation) is the following innocuous line of output:

```
sort_list.cpp:8:   instantiated from here
```

The actual error, in this case, is that the STL sort() requires a pair of Random Access Iterators, i.e., iterators that can move any number of steps forward or backward in constant time. The STL

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.