

Electronic Notes in Theoretical Computer Science 238 (2009) 71–82

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Towards the Modularization of C++ Concept Maps

Zal´an Szu˝gyi[1](#_bookmark0) A´d´am Sipos[2](#_bookmark0) Zolta´n Porkola´b[3](#_bookmark0)

*Department of Programming Languages and Compilers E¨otv¨os Lora´nd University*

*Budapest, Hungary*

**Abstract**

*Concept* is a new and powerful language element being introduced in the next C++ standard. With the help of concepts we can define the properties a template requires from its type arguments. If a type does not fulfill the requirements syntactically, but semantically, the connection must be declared with the use of a concept map. Often the description of the semantic matching results in long codes that need to be modularized. In this paper we present an extension to the concept map language constructs that enables this transformation. We introduce the well-known public, protected, private class arrangement scheme into concept maps. We present our preprocessor, that transforms the modularized code into regular code to be compiled by ConceptGCC, the experimental Concept C++ compiler.

*Keywords:* C++, ConceptC++

# Introduction

*Generative programming* [[2](#_bookmark5)] is a new programming paradigm, focusing on program transformation, code generation, and the development of highly reusable libraries. The C++ programming language supports generative programming by the use of its generic construct, the *template*[[11](#_bookmark10)]. Templates in C++ are *unconstrained*, so it is possible to instantiate a class template or function template with any type argument. The instantiation process is carried out irrespective of whether or not the argument satisfies the requirements implicitly set by the template. Thus the relevant compiler errors are triggered too late in the instantiation process causing complicated and difficult to understand error messages. Writing a good documen- tation and giving informative names to our template’s parameters is not a sufficient solution for informing the user of the template’s requirements.

1 Email: [lupin@inf.elte.hu](mailto:lupin@inf.elte.hu)

2 Email: [shp@inf.elte.hu](mailto:shp@inf.elte.hu)

3 Email: [gsd@inf.elte.hu](mailto:gsd@inf.elte.hu)

1571-0661/© 2009 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2009.05.008

The research area *concept checking* identified this problem and provided library- level solutions [[7](#_bookmark11),[8](#_bookmark12)]. A cleverly designed C++ code is capable of deciding at compile- time if two classes are in a parent-child relationshup, or if a type is a pointer type, and other properties. The most notable concept checking library containing such constructs is boost::concept check [[12](#_bookmark13)]. The boost::concept check library included and unified many previous approaches, and attracted attention to this area. However, the library still lacks a number of functionalities, and proved that a language-based solution is required rather than a library-based one.

A new C++ language construct, the *concept* [[3](#_bookmark6)] solves the problem of compile-

time type checking at the language level by adding constrained templates to the language, and still keeping all of the advantages of the C++ template system. A concept defines a set of requirements on one or more types. These requirements

describe the *interface* a type needs to offer to meet the template’s precondition.

The interface is described with *function signatures* and *associated types*. If a type satisfies all the requirements of a concept we say that type *models* this concept.

In many cases a class does not syntactically match the requirements of a concept. For example the concept may require that the *plus* (+) operator be defined on the argument type[[13](#_bookmark15)]. In the case of a Color type, however, the *plus* operation may be modelled with the method mix() mixing two colors. This *seman- tic* matching can be described with a concept map.

Originally concept maps were intended to be short code parts with function signatures and short function bodies. However, nowadays concept maps are being used for a number of new and complex purposes, like serving as *adaptors* between a class and an interface (see the std::vector – Stack example in Section [3](#_bookmark2)). Thus the description of a semantic match may result in a longer and more complicated program code. Currently this whole code must be written either in the function body or with the help of other function calls. Avoiding repetition of common code fragmens in concept maps also requires the implementation of helper functions. However, currently such functions have to be declared and implemented outside of the concept map. Thus, helper functions regularly declared as free functions or placed in ad-hoc implementational namespaces, leading to scattering helper func- tions in the source and seriously decrease the quality of the code.

This phenomenon is in sharp contrast with other constructs of the C++ lan- guage. Classes, for instance, support better modularization: private and public visibility distinguish helpers from interface functions, in the same time avoid code scattering grouping them into one code unit.

In this paper we present an approach for modularizing concept maps thus cre- ating code that is easier to understand, and maintain. We present our preprocessor implementing the approach by transforming the modularized code into C++ code accepted by the experimental ConceptGCC compiler.

The rest of the paper is organized as follows. In Section [2](#_bookmark1) we discuss the concept and concept map language constructs. In Section [3](#_bookmark2) we describe the concept map modularization problem to be addressed in the paper. Section [4](#_bookmark3) discusses our solution for the problem, and the preprocessor implementing our approach. In

Section [6](#_bookmark4) we conclude our paper and discuss future work.

# Concepts

The language construct concept is essentially a listing of function signatures and associated types that we intend to use in our template. For example, the following concept requires that the equality operator== is defined for types T and U.

concept EqualityComparable<typename T, typename U = T>

{

bool operator==(T a, U b);

};

Every type whose equality operator is defined models this concept. The equality operator can be defined as a member or a non member function. We can refine an existing concept by adding new requirements for it. For example we add a requirement for the less than (<) operator to the previous concept.

concept Comparable<typename T, typename U = T>

: EqualityComparable<T,U>

{

bool operator<(T a, U b);

};

There is another way to create a concept, using the keyword auto before keyword

concept. Consider the following example:

auto concept EqualityComparable<typename T, typename U = T>

{

bool operator==(T a, U b);

};

In the first two cases it must be explicitly declared how the type satisfies the re- quirements, and in the third case this is not necessary. The concept map is the language construct used for describing the semantical connection between the type and the concept. A type can satisfy the requirements *syntactically* if it contains the functions and types that the concept requires. The type satisfies the requirements *semantically* if even though the type does not have all of these functions, the usage of these functions is semantically rational. Let us consider the following example.

struct Person

{

int id; string name;

};

There is no equality operator for Person, but we can consider two persons equal when their id-s are equal. Using concept map it is possible to explicitly declare this semantic equality.

concept\_map EqualityComparable<Person>

{

bool operator==(Person a, Person b)

{

return a.id == b.id;

}

};

If the type syntactically satisfies the requirements (for example int has an equality operator) we can just write the following:

concept\_map EqualityComparable<int>

{

}

Note that in case the concept is declared as an auto concept we do not need to make this mapping.

In the following example we present how to write a constrained function tem- plate. The following function utilizes the equality operator of type T and declares this with the requires keyword.

template<typename T>

requires EqualityComparable<T> void f(const T& t1, const T& t2)

{

... if(t1 == t2) ...

}

Function f can be instantiated only with types satisfying the EqualityComparable

concept.

1. **Transforming the** concept map

A concept map declares explicitly how a type satisfies the requirements of a con- cept. In the context of one concept, with concept map we can redefine some member functions of a type, or add some new ones without doing any modification inside the type. For example, in the previous chapter the Person struct has no equality operator, but with a concept map we can define one. Thus Person can be passed to any class template or function template requiring an EqualityComparable argu- ment. The concept map makes it easier to solve certain problems. Let us suppose we need an adaptor for the container std::vector. An obvious though tedious approach is to write a wrapper class Stack for the container. However, we can utilize a concept map and essentially rename the container member functions. In the following example, we demonstrate how to create a stack using std::vector[[4](#_bookmark7)].

concept Stack<typename X>

{

typename value\_type;

void push(X&, const value type&); void pop(X&);

value type top(const X&);

};

template<typename T>

concept\_map Stack<std::vector<T> >

{

typedef T value type;

void push(std::vector<T>& v, const T& x) { v.push back(x); } void pop(std::vector<T>& v) { v. pop back(); }

T top(const std::vector<T>& v) { return v. back(); }

};

With the concept map we have described how the std::vector can be regarded as a Stack. Now we can instantiate every class template or function template with std::vector, even if the template argument is required to fulfill the Stack con- cept. In these examples the code of the concept maps is only two or three lines long. However, many times with complex concept mappings the code describing the semantical matching may be more complicated and longer. In our next exam- ple we present how a to simulate an aspect-oriented feature [[6](#_bookmark9)] with the help of concept map. Let us suppose we want to log data before and after calling the f() and g() functions of type MyType. Our logging procedure consists of opening a network connection, carrying out file I/O operations, and ending the connection.

concept LogConcept<typename C>

{

void f(C&); void g(C&);

}

concept\_map LogConcept<MyType>

{

void f(MyType& c)

{

// open network connection

// open logfile

// log into file

// close logfile

// close network connection

// error handling c.f();

// open network connection

// open logfile

// log into file

// close logfile

// close network connection

// error handling

}

void g(MyType& c)

{

// open network connection

// open logfile

// log into file

// close logfile

// close network connection

// error handling c.g();

// open network connection

// open logfile

// log into file

// close logfile

// close network connection

// error handling

}

};

The logging part of this concept map contains several redundant lines of code. On one hand this leads to long function bodies in the concept map. On the other hand the actual original function calls (c.f() and c.g()) will be lost in the long code. It may be useful to trigger a separate helper function call at these points, however, there is no well situated place to put these functions. It is possible to put the helper functions into the global namespace, but if they are related only to the concept map this solution will be confusing and makes our code harder to understand.

In the following we describe our remedy for these problems.

# Modularization

Our approach is to introduce helper functions into concept maps. These functions do not express some requirement for the template argument, but rather are called from bodies of the various functions of the concept map, just as if they were or- dinary C++ functions. The concept map structure is extended with private and protected and public parts as done in class or struct constructs. Functions defined as private or protected are hidden outside the concept map. The public keyword is optional, as all requirements (function signatures) are public by default. Consequently, our approach is backward compatible with the current concept mech- anism: if none of the previous visibility keywords are used, the concept map has its regular form, and can be compiled. At this stage of the research the protected and private keywords provide the same functionality. One of our most important

future plans is to iontroduce some refined meaning for protected.

Let us consider the following example. We modified the example about logging in the previous chapter using the public and private keywords.

concept LogConcept<typename C>

{

void f(C&); void g(C&);

}

concept\_map LogConcept<MyType>

{

public:

void f(MyType& c)

{

log();

c.f();

log();

}

void g(MyType& c)

{

log();

c.g();

log();

}

private:

void log()

{

// open network connection

// open logfile

// log into file

// close logfile

// close network connection

// error handling

}

};

In the function bodies of f() and g() we have substituted the redundant logging procedure with simple function calls. Consequently our concept map is shorter, more readable, avoids code duplication, and is easier to maintain. The log() func- tion declared as private will be unreachable from the outside of the concept map, thus only member functions will be able to use it. This extended notation of concept maps is not a valid ConceptC++ construct, thus a compiler extension is required to be able to handle the notation.

The only compiler currently implementing concepts for C++ is *Concept- GCC* [[13](#_bookmark15)]. We are not intending to create a new concept compiler, rather to write a preprocessor transforming our code into valid ConceptC++ code and still keeping our new structural advantages. The idea is to put the private and protected parts of the concept map to a unique namespace and put the namespace scope before the function calls in the public part. We have implemented a precompiler- like program to carry out this transformation. After the transformation we get the following code:

namespace unique\_ns\_name

{

void log()

{

...

}

}

concept\_map LogConcept<MyType>

{

void f(MyType& c)

{

unique\_ns\_name::log(); c.f(); unique\_ns\_name::log();

}

void g(MyType& c)

{

unique\_ns\_name::log(); c.g(); unique\_ns\_name::log();

}

};

As seen in example 9 the concept map itself can be template. In this case the template argument may appear inside the helper functions. Transforming this kind of concept maps is not so simple as it was in the previous example. When the helper function definition is moved from the body of the concept map to the unique namespace, the template arguments of the concept map became unknown types. Thus it is necessary to transform these helper functions to template functions with the same template arguments as the concept map. It is necessary to instantiate these helper functions explicitly in the public part of the concept map, because the template arguments may not be part of the function parameter list. In the following example we create a new concept map (which contains helper functions) to concept Stack defined in example 9.

template<typename T>

concept\_map Stack<std::vector<T> >

{

public:

typedef T value type;

void push(std::vector<T>& v, const T& x)

{

f();

v.push back(x); g();

}

void pop(std::vector<T>& v) { v. pop back(); }

T top(const std::vector<T>& v) { return v. back(); } private:

void f() { ... T t; ... do something before push ... }

void g() { ... T t; ... do something after push ... }

};

After the transformation we get the following code:

namespace unique\_ns\_name

{

template<typename T>

void f() { ... T t; ... do something before push ... }

template<typename T>

void g() { ... T t; ... do something after push ... }

}

template<typename T>

concept\_map Stack<std::vector<T> >

{

typedef T value type;

void push(std::vector<T>& v, const T& x)

{

unique\_ns\_name::f<T>(); v.push back(x); unique\_ns\_name::g<T>();

}

void pop(std::vector<T>& v) { v.pop\_back(); }

T top(const std::vector<T>& v) { return v.back(); }

};

In some cases the member functions are so long that it is not practical to store them within the body of the concept map (similarly to the case of C++ structs and classes and their member functions). Our approach provides a solution for these cases, as it is possible to define the functions outside the type definition. When

the concept map is not a template the helper function can be defined in a separate file. It is necessary to put the outside definition into the same namespace as the concept map. In the following example we implement the logger functions outside the concept map.

concept\_map LogConcept<MyType>

{

public:

void f(MyType& c)

{

log();

c.f();

log();

}

void f(MyType& c)

{

log();

c.g();

log();

}

private:

void log();

}

// function can be placed into a separate file void concept\_map LogConcept<MyType>::log()

{

...

}

After the transformation we get the following code:

// Forward declaration is needed because

// the definition of the function can be in a separate file namespace unique\_ns\_name

{

void log();

}

concept\_map LogConcept<MyType>

{

void f(MyType& c)

{

unique\_ns\_name::log(); c.f(); unique\_ns\_name::log();

}

}

// can be placed into a separate file namespace unique\_ns\_name

{

void log()

{

...

}

}

# Related work

A strong restriction on concept maps is that it must be in the same namespace as the original concept. If we intend to use a third-party library (as in [[5](#_bookmark8)]), we must put our own concept maps into the library’s namespace. This is not a clean approach, and may obviously cause problems. In [[9](#_bookmark14)] the authors present a detailed description of the problem and also a solution. Our approach can fully cooperate with their proposed solution, as these two problems are independent of each other.

# Conclusion

In this paper we have presented a new approach for modularizing the new C++ language construct concept map. The keywords private, public, and protected are now available for use in concept maps, resulting in more maintainable code. Our preprocessor is capable of recognizing concept maps, and transforming the contained code into simple ConceptGCC code. Our solution is backward compatible with the proposed C++ Standard, i.e. omitting keywords we get semantically equivalent code with the Concepts proposal.

# References

1. Bourdev, L. and H. Jin: *Generic Image Library*, <http://adobe.com/gil/> 2006. opensource.
2. Czarnecki, K. and U. W. Eisenecker: *Generative Programming: Methods, Tools and Applications*, Addison-Wesley 2000.
3. Douglas, G., J. J¨arvi, J. G. Siek, G. D. Reis, B. Stroustrup and A. Lumsdaine: *Concepts: Linguistic Support for Generic Programming in C++*, ”Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA’06)” (2006)
4. Douglas, G. and B. Stroustrup: *Concepts (Revision 1)*, 2006.
5. J¨arvi, J., M. A. Marcus and J. N. Smith: *Library Composition and Adaptation using C++ Concepts*, In Proc. of the 6th international Conference on Generative Programming and Component Engineering,

pp. 73-82, 2007.

1. Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin: *Aspect-Oriented Programming*, Ak¸sit et al: ”Proceedings European Conference on Object-Oriented Programming **1241**” (1997), Springer-Verlag, pp. 220-242, 1997.
2. McNamara, B., Smaragdakis, Y.: *Static Interfaces in C++*, In First Workshop on C++ Template Metaprogramming, October 2000, Erfurt.
3. Siek, J., Lumsdaine, A.: *Concept Checking: Binding Parametric Polymorphism in C++*, In First Workshop on C++ Template Metaprogramming, October 2000, Erfurt.
4. Siek, J.: *Scoped Concept Maps*, C++ Standards Committee Paper.
5. Siek, J., L.-Q. Lee and A. Lumsdaine: *The Boost Graph Library: User Guide and Reference Manual*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
6. Stroustrup, B.: *The C++ Programming Language Special Edition*, Addison-Wesley 2000.
7. *The Boost Concept Checking Library.* [http://www.boost.org/libs/concept\_check/concept\_check.](http://www.boost.org/libs/concept_check/concept_check.htm) [htm](http://www.boost.org/libs/concept_check/concept_check.htm)
8. *ConceptGCC BoostCon Edition.* <http://www.generic-programming.org/software/ConceptGCC/>