Essays on C++ Concepts

Jyrki Katajainen (Editor)

Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
jyrki@diku.dk

Contributors

Kasper Egdø
Kean Pedersen

CPH STL Report 2008-4, July 2008.

Preface

The essays published in this volume were written in June 2006 as an assignment in our course "Generic programming and library development" (taught by Kenny Erleben, Knud Henriksen, and myself)[1]. Essays were written in groups of 1–3 people, and in total 15 essays were handed in. Of these, six essays were invited to this volume. The revision of the essays gave no extra study credits so at the end only two of the invited essays ended up in this volume.

The essays could be written on two topics: A) Generic programming and B) Concepts in C++0x. Both of the essays in this volume are on C++ concepts. Bjarne Stroustrup, the inventor of the C++ programming language, visited our department in March 2006, and in his talk he devoted some time for the C++ concepts. His slides[2] were given as a starting point for the work of the students. Today the topic is equally relevant as it was in 2006 due to the forthcoming ratification of the new C++ standard[3]. Concepts are supposed to solve many of the problems one encounters in C++ template programming. By reading these essays, you can decide yourself whether this is the case or not.

I would like to thank Kasper and Kean for permitting us to publish their essays.

July 2008 Jyrki Katajainen

[1] Course home page, Generic programming and library development, http://www.diku.dk/forskning/performance-engineering/Courses/Generic-programming-2006

[2] Bjarne Stroustrup, Future direction for C++0x, http://www.diku.dk/forskning/performance-engineering/Talks/2006-Announcements/2006.03.14-13%3A00%3A38.html

[3] Wikipedia, C++0x, http://en.wikipedia.org/wiki/C%2B%2B0x

Table of contents

Preface i

Table of contents ii

C++ koncepter og unødig kompleksitet 1
Kasper Egdø

Concepts in C++0x 6
Kean Pedersen

Author index 10

C++ koncepter og unødig kompleksitet

Kasper Egdø

Datalogisk Institut, Københavns Universitet
Universitetsparken 1, DK-2100 København Ø, Danmark
egdoe@diku.dk

Resumé. Jeg betragter i det følgende dele af nogle af konceptforslagene til C++, og argumenterer for at forslaget af Siek og andre er ramt af feature-creap.

1. Introduktion

Der findes efterhånden en del forskellige forslag til hvorledes C++ kan understøtte *koncepter*. I Boost finder vi Boost Concept Check Library (BCCL) [3], som på eksisterende compilere kan hjælpe både brugere og implementører af generiske algoritmer og klasser med at forstå hvorfor de ikke compiler hhv. om de bruger funktionalitet udover den de er dokumenteret til at anvende. Reis og Stroustrup foreslår en udvidelse til C++ baseret på "use patterns" [2], og Siek og andre foreslår en udvidelse baseret på pseudosignaturer og "name conformance" [4] (revideret i [1]), begge således at sproget egentlig understøtter koncepter.

Hvor BCCL blot er en hjælp nu (men ikke noget stærkt værktøj), lover begge forslag til sprogudvidelser væsentligt forbedrede muligheder for både bruger og implementører af generisk kode. De to tilgangsvinkler til udvidelsen er noget forskellige, men opnår omtrent den samme udtryksstyrke (dog lader Reis og Stroustrup's løsning til at være en smule stærkere idet man kan anvende `not` og `or` i ens `where`-clauses). Jeg vil i det følgende beskæftige mig primært med Siek og andres forslag.

2. Fordele ved koncepter

Ved begge forslag bliver det muligt at skrive template-funktioner/klasser der udnytter at de kan stille krav til de typer de anvendes med. Det kan se f.eks. således ud:

```
template<InputIterator IterIn, OutputIterator IterOut>
where EqualityComparable<IterIn::value_type>,
      Assignable<IterIn::value_type>
IterOut unique_copy(IterIn begin, IterIn end, IterOut out) {...}
```

Hermed kan man undersøge implementeringen af `unique_copy` på definitionstidspunktet for om den kun bruger de funktioner/typer der hører

Skrevet juni 2006.

til dens koncepter. Tilsvarende kan brugeren af funktionen få en tydelig compile-fejl hvis han forsøger at anvende funktionen med typer der ikke overholder kravene.

En af fordelene er at man herved kan have mere end én udgave af funktionen, således at man for "stærkere" iteratorer kan skrive mere effektiv kode — eksempelvis kunne man for RandomAccessIterators undgå en kopiering internt i funktionen; man kunne da have følgende overload af `unique_copy`:

```
template<RandomAccessIterator IterIn,
         MutableRandomAccessIterator IterOut>
where EqualityComparable<IterIn::value_type>,
      Assignable<IterIn::value_type>
IterOut unique_copy(IterIn begin, IterIn end, IterOut out) {...}
```

Compileren kan da på det sted funktionen kaldes vælge det "bedste" af de gyldige overloads — med bedste menes da den udgave af funktionen der bruges den mest *refine*'ede iteratortype. Her er der naturligvis potientialle for nogle tvetydighedesproblemer a la dem man kan opleve med "almindelige" funktionsoverloads, hvor flere funktioner kan være lige gode.

3. Kompleksitet

Siek og andre lader os erklære koncepter ved hjælp af en ny konstruktion, som kan udtrykke hvilke krav en type skal efterkomme for at leve op til konceptet. Eksempelvis kan man definere konceptet *Container*:

```
template<typename T>
concept Container
{
  bool X::empty() const;
  size_type size() const;
  ...
}
```

Siek og andre argumenterer for at en given type kun skal modelere et koncept hvis dette eksplicit angives i koden, via et statement i stil med:

```
template<>
concept Container<MyContainer>
{
}
```

Derved gøres det eksplicit at `MyContainer` er en *Container* (og compileren kan tjekke om alle kravene er opfyldt, og give en fejlmeddelelse hvis de ikke er).

I et forsøg på at tage livet af traits-klasser (med konkret fokus på `IteratorTraits`), indlemmes deres funktionalitet også i konceptfunktionaliteten, således at man ikke blot kan stille krav om en bestemt typedefinition skal eksistere i en type, men man kan endda definere den når man skriver sin template-koncept:

```
template<>
concept Container<MyContainer>
{
  //En container skal definere en value_type:
  typename value_type;
  //Findes ingen reference typedef, så
  //default til en reference til value_type:
  typename reference=value_type&;
  ...
}
```

Af hensyn til de indbyggede typer (`int`, `int*`, `char` osv.), der jo ikke kan have nestede typedefinitioner, kan det være praktisk at kunne anvende noget der ligner partiel template specialisering for at give disse nogle typedefinitioner svarende til dem der findes for `IteratorTraits`, og det er også muligt:

```
template<typename T>
concept Container<T*>
{
  typedef T value_type;
  typedef T& reference;
  ...
}
```

Ovenstående lader til at være nødvendig funktionalitet hvis man skal kunne anvende koncepter sammen med de indbyggede typer. Herefter er det mit indtryk at Siek et al er blevet lidt for inspirerede, og fortsætter med at tilføje funktionalitet til deres koncepter. Udover default-typedefinitoner tilføjer de mulighed for default-implementeringer af funktioner — dette illustreres bedst ved deres `LessThanComparable`-koncept:

```
template<typename T>
struct concept LessThanComparable
{
  bool operator<(const T&, const T&);
  bool operator<=(const T& x, const T& y)
   { return !(y < x); }
  bool operator> (const T& x, const T& y)
   { return y < x; }
  bool operator>=(const T& x, const T& y)
   { return !(x < y); }
};
```

For at en type skal opfylde dette koncept, skal den implementere `operator<`, og herefter kan al kode der *kræver* en *LessThanComparable*-type, anvende disse funktioner. Herved opnår man desværre at template-kode får adgang til noget funktionalitet som ikke-template-kode ikke har adgang til — det virker som et brud med den øvrige del af C++, at almindelig ikke-template-kode nu ikke længere skulle have mulighed for at tilgå samme funktionalitet

som template-kode. Det er til og med unødvendigt, idet der i forvejen findes velfungerende måder enkelt at tilføje de ekstra funktioner på (f.eks. via Boosts operators-bibliotek).

Et værre eksempel kunne være hvis Container-konceptet havde en default-implementering af `empty`-funktionen:

```
template<typename T>
concept Container
{
  size_type X::size() const;
  bool X::empty() const
   { return size()==0; }
}
```

Herved kunne en forfatter af `MyContainer` foranlediges til at tro at han havde implementeret en komplet container, men rent faktisk mangler en af de hyppigst anvendte funktioner på denne; det virker ikke rimeligt at en implementering med den manglende funktion af compileren erklæres for at overholde konceptet.

Ud over problemerne med default-implementeringerne introduceres også ny logik for opslag af funktioner og typenavne, idet funktioner og typer defineret i koncepter anvendt af en template, skal kunne anvendes direkte (altså der er ikke brug for eksplicit namespace brug). I forvejen er reglerne for namelookup bestemt ikke uden problemer eller trivielle at gennemskue (der findes adskillige WG21 foreslag til forbedringer til namespace funktionaliteten). Selvom det kan være rart at slippe for at skulle skrive nogle lokale typedefnitioner, så medfører tilstedeværelsen af de relevante koncepters typedefinitioner i det lokale namespace at man på lettere magisk vis får adgang til nogle navne, som det ikke ved læsning af en template-funktion umiddelbart er klart hvilket koncept kommer fra. I en del tilfælde vil disse navne også være tvetydige (hvis man f.eks. anvender to forskellige iteratorer i en funktion, og f.eks. deres `value_types` ikke er ens, og man kommer da herved ud i at alligevel skulle til at anvende eksplicitte namespaces.

Namespace-funktionaliteten lider (som med default-funktionsimplementeringerne) af at man ikke kan tage noget template-kode, og kopiere ud i en ikke-template-funktion (en teknik jeg hyppigt anvender til at debugge template-kode), og få det til at køre uden at skulle ændre i koden ved at tilføje eksplicitte namespaces til de anvendte typer og funktioner.

4. Konklusion

En del af de nye forslag til koncepter i C++ ser lovende ud, men der synes at være et behov for at få begrænset omfanget af den nye funktionalitet, således at den ikke går ud over hvad der er nødvendigt for at opnå de primære mål med den, nemlig bedre fejlbeskeder, forbedret typecheck, nye overloading muligheder, og simplere template-kode.

Litteratur

[1] D. Gregor, J. G. Siek, J. Willcock, J. Järvi, R. Garcia, and A. Lumsdaine, Concepts for C++0x, revision 1, Technical Report N1859=05-0109, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2005).

[2] G. Reis and B. Stroustrup, *Specifying C++ concepts*.

[3] J. Siek and A. Lumsdaine, *The Boost Concept Check Library (BCCL)* (2000). (`http://boost.org/libs/concept_check/concept_check.htm`.)

[4] J. G. Siek, D. Gregor, R. Garcia, J. Willcock, J. Järvi, and A. Lumsdaine, Concepts for C++0x, Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2005).

Concepts in C++0x

Kean Pedersen

Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
zakarun@diku.dk

Abstract. This essay presents an overview of the proposed addition to C++, called concepts.

1. Introduction

This presentation of the concepts for C++0x is based on the proposed syntax and semantics presented in Concepts for C++0x[1].

Concepts are a way of helping the users and creators of generic code in making sure that the types they're using actually can do what is expected by the generic code. In particular, it's a way to check that the types used in a template has the needed operators, fields and members needed to instantiate the template. The catch is that this check is now made both when the template is first encountered, and again when the template is to be instantiated.

1.1 Overview of Concepts

A concept is a set of requirements on a type. This overview starts from the viewpoint of a user of a concept-enabled template function.

You could for instance declare the concept of being comparable to an integer like this:

```
template<typeid T>
concept IntegerComparable {
    bool operator==(T a, int b);
    bool operator==(int a, T b) { return b == a; }
    bool operator!=(T a, int b) { return !(a == b); }
    bool operator!=(int a, T b) { return !(a == b); }
};
```

Now there are several things to notice. The first is the keyword `concept`, indicating that we are declaring a concept named `IntegerComparable`. The second is that we require four functions to be present in order for a type

[1] Document number: N1758=05-0018 at open-std.org.

Written June 2006.

to call itself IntegerComparable. The third thing to notice is that we can provide default implementations for the required functions.

Say that we have a class, A, which we would like to tag as being IntegerComparable. We say that A is a model of that particular concept. Then the syntax for doing this according to the proposal is:

```
class A {
    public:
        int GetIntegerRepresentation() const;
    private:
        ...
};

model IntegerComparable<A> {
    bool operator==(A a, int b) {
        return a.GetIntegerRepresentation() == b;
    }
};
```

This effectively declares our type A to be a model of the concept IntegerComparable. Notice that we only provide one function in the model, but still have the defaults declared in the concept. This greatly reduces the need to write trivial operators.

This is all good; we can now declare concepts and models of those concepts, but we still need to use them for something. A concept is a way of saying that a particular type can be used in particular ways. The IntegerComparable concept says that types modelling this concept can be compared to an integer. You could declare other concepts, and you don't need to restrict yourself to operators.

Moving over to using the concepts, let's say we are making a generic library for comparing types to other types. At the very heart of this library is a template for comparing two types to integers. Again using the syntax proposed, you would declare this template function like this:

```
template<typeid T1, typeid T2>
where {
    IntegerComparable<T1>,
    IntegerComparable<T2>
}
bool LibraryHeart(T1 t1, T2 t2, int n) {
    return n == t1 && n != t2;
}
```

First notice the reuse of the keyword `typeid` instead of `typename`. This is to ensure backwards compatibility, and also an indication to the compiler, that this template uses concepts. Second notice the new construct with the keyword `where`. This indicates what is expected of the types T1 and T2 — that they are models of `IntegerComparable`. The rest is a normal template. Third, notice how the default operators are used from the concept, without them being defined specifically for the class A.

This is the basic syntax of concepts. In the next section the big question of why to use concepts is answered. But first some more notes on what is possible with concepts.

A main feature is "inheritance" between concepts, or refinement as it is called. You can have a concept B that refines another concept A. A will then have a subset of the restrictions that B has on a type.

You can also say that a model of a particular concept must have a typedef named `value`, and you can give that typedef a default type. This does away with the current need for traits and the extra template parameters they impose on public templates.

1.2 Why concepts?

Taking the example from above, and writing it with normal C++, you would do something like this:

```
class A {
    public:
        int GetIntegerRepresentation() const;
        bool operator==(A a, int b)
            { return GetIntegerRepresentation() == a; }
        bool operator==(int a, A b) { return b == a; }
        bool operator!=(A a, int b) { return !(a == b); }
        bool operator!=(int a, A b) { return !(a == b); }

    private:
        ...
};

template<typeid T1, typeid T2>
bool LibraryHeart(T1 t1, T2 t2, int n) {
    return n == t1 && n != t2;
}
```

Imagine having to use one more class B in the template, and having to again implement all the extra operators. Of course the template function could promise just to use operator==, but that would quickly turn into ugly code. With concepts, all these extra operators can be added when you declare the model, and that is without having to fiddle around with non-intuitive class inheritance.

But that's not the strongest point of concepts. Imagine forgetting the operator!= in class A above, and then imagine the hours you need to spend reading the compiler error you get, which is even harder to understand if you didn't write the template function you're using.

With concepts, the compiler does an extra check to see if your model actually models the concept declared. If not, you will get an error along the lines of "class A does not fully model concept B, missing X". The compiler

can now also do a check for the template developer. When the template is first encountered doing compilation, it is checked that it can actually be instantiated using just what is declared in the concepts for the types. That way you can be sure that you don't use e.g. `last - first` when you're dealing with forward iterators. You might in your test cases only use random access iterators, and thus not discover the error. If you on your `first` and `last` said they where of the concept ForwardIterator, the compiler could tell you that you where using an operator not declared for that concept. This is done even before the first instantiation of the template.

An important notice regarding concepts is that they are purely used at compile time. No runtime checks are necessary, and it is therefore recommended that concepts are used in generic programming.

Another important note is that concepts can only impose syntax on a type, regarding traits, operators, members, fields etc., but it cannot impose semantics. The compiler can't tell whether an operator++ actually makes an iterator point to the next element, or does something completely different. That's what testing is for, concepts only help in type checking and syntax checking.

Author index

Egdø, Kasper 1
Katajainen, Jyrki i
Pedersen, Kean 6