Concepts Lite

Constraining Template Arguments with Predicates

Andrew Sutton,
Bjarne Stroustrup, Gabriel Dos Reis
Texas A&M University

Quick Update

All concept-related information presented here will be included in an ISO TS (Technical Specification)

- A TS is an extension of the standard
  - http://isocpp.org/std/iso-iec-jtc1-procedures

Based on WG21 document n3580

Aim to deliver TS at the same time as C++14

Concepts Lite Resources

Information about compilers, libraries, and concepts related to Concepts-Lite work (under construction)

http://concepts.axiomatics.org/

Implementation:

GCC-4.9 Compiler

Overview

Introduction

Constraining templates

Defining constraints

Language mechanics

Notation

Implementation

Programming

Templates: An Ideal

Abstract expression of algorithms, data structures

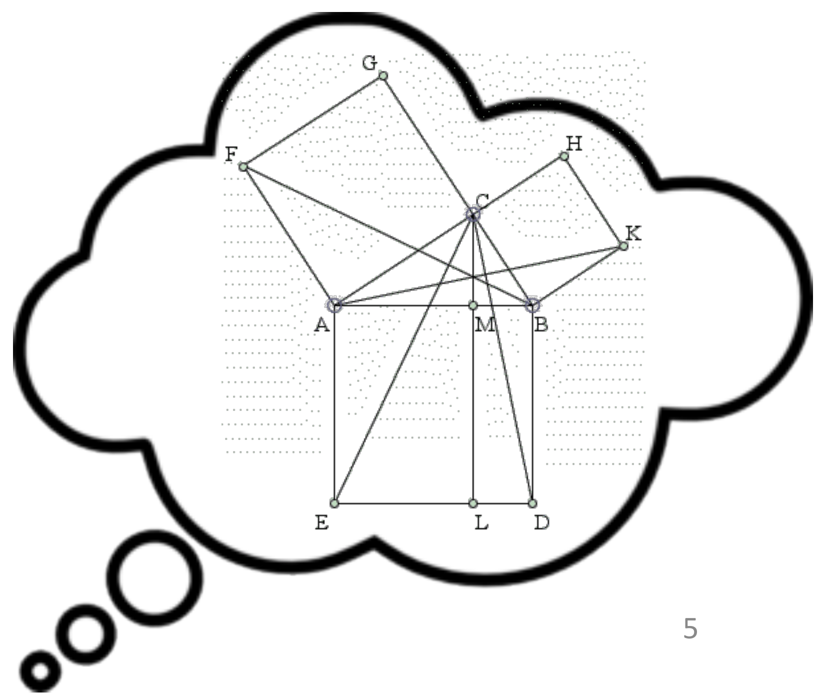
- Integers, Reals, Sequences, Sets, Graphs, etc.

Generality

- Not limited to a single model

Fast code

- No abstraction penalty
- Type-based optimizations



Templates: The Reality

```
template<typename T>
typename enable_if<is_integral<T>::value, T>::type
gcd(T a, T b)
{
  return do_gcd(a, b,
                typename is_unsigned<T>::type{});
}
```

Templates: Reality Bites

```
gcd(16.0, 2.0); // Error!

error: In the instantiation of ‘gcd(T, T)’
  where T = double
error: In the instantiation of ‘do_gcd(T, T, X)’
  where T = double, X = integral_constant<bool, false>
error: In the instantiation of ‘euclid_gcd(T, T)’
  where T = double
error: no match for operator ‘%’ in ‘a % b’
note: candidates are:
note:   operator%(int, int)
note:   operator%(long, long)
note:   ...
```

Concepts Lite: Template Constraints

Improve language support for generic programming

- Directly state requirements on template arguments
- Check requirements at the point of use
- Support overloading and specialization based on constraints
- Improve interfaces and enhance diagnostics

Without runtime overhead or long compilation times

Almost completely implemented (twice)

- Handles the Standard Library algorithms and their uses

Constraints Are Not Concepts

Only check requirements at the point of use

- Does not check template definitions

No dramatic changes to lookup rules

Approach allows incremental adoption/use of concepts in generic libraries

There is a (language) migration path to concepts

Constraining Template Arguments

Constrain template arguments with predicates

```
template<Sortable_container C>
void sort(C& container);
```

Equivalently:

```
template<typename C>
  requires Sortable_container<C>()
void sort(C& container);
```

Constraints

Are just `constexpr` function templates

```
template<typename T>
concept bool Sortable()
{
  return ...; // Returns true when T is a
              // permutable container whose
              // elements can be totally ordered
}
```

Constraint Checking

Constraints are checked at the point of use

```
forward_list<int> lst { ... };
sort(lst);
```

See program output

Constraints on Class Templates

Just like function templates

```
template<Object T, Allocator A>
class vector;
```

Equivalently:

```
template<typename T, typename A>
  requires Object<T>() && Allocator<A>()
class vector;
```

Constrained Members

Member functions and constructors can be constrained

```
template<Object T, Allocator A>
class vector {
  vector(const vector& x)
    requires Copyable<T>();

  void push_back(T&& x)
    requires Movable<T>();
};
```

Constrained Member Definitions

Out-of-class member definitions are matched to their declarations by requirements

```
template<Object T, Allocator A>
void vector<T, A>::push_back(T&& x)
  requires Movable<T>()
{
  ...
}
```

Multi-type Constraints

Constraints can be applied to multiple types

```
template<Sequence S,
         Equality_comparable<Value_type<S>> T>
Iterator_type<S> find(S&& s, const T& value);
```

Equivalently with a `requires`:

```
template<typename S, typename T>
  requires Sequence<S>()
        && Equality_comparable<T, Value_type<S>>()
Iterator_type<S> find(S&& s, const T& value);
```

Overloading

Function overloading is extended to include constraints

```
template<Input_iterator I>
void advance(I& iter);

template<Bidirectional_iterator I>
void advance(I& iter);

template<Random_access_iterator I>
void advance(I& iter);
```

Overloading

Compiler selects the *most constrained* overload

```
istream_iterator<int> iter(cin);
advance(iter); // Input overload

list<T>::iterator first = lst.begin();
advance(first); // Bidirectional overload
```

The most constrained is automatically determined by comparing template constraints

Class Template Specialization

Also extended to support constraints

```
template<typename T>
  class complex; // Undefined primary template

template<Real T>
  class complex<T> { ... }; // Complex number

template<Integer T>
  class complex<T> { ... }; // Gaussian integer
```

Specialization arguments

More About Constraints

Discussed in n3580:

- Constrained alias templates
- Constrained template template parameters
- Variadic constraints

Defining Constraints

Writing requirements

Requires expressions

Type requirements

Defining Constraints

A *constraint* is effectively a **`constexpr`** template

- Has **`concept`** as a decl-specifier instead of **`constexpr`**
- Can use type traits, call other **`constexpr`** functions
- Cannot be specialized (by constraints)

Constraints check *syntactic requirements*

- Is this expression valid for objects of type **T**?
- Is the result type of an expression convertible to **U**?

The meaning of **`concept`**

The **`concept`** declaration specifier has the following meaning:

- The declaration is **`constexpr`**
- The declaration may not be specialized or refined
- The declaration must have a definition
- The declaration name can be used as a type specifier

Constraints: First Pass

Use type traits

```
template<typename T>
concept bool Equality_comparable()
{
  return has_eq<T>::value // a == b
      && is_convertible<eq_type<T>, bool>::value
      && has_ne<T>::value // a != b
      && is_convertible<ne_type<T>, bool>::value;
}
```

Many, many downsides

Constraints: Current Design

Invent new syntax for requirements

```
template<typename T>
concept bool Equality_comparable()
{
  return requires (T a, T b) {
    {a == b} -> bool;
    {a != b} -> bool;
  };
}
```

Constraints: Longhand

Can be equivalently written as

```
template<typename T>
concept bool Equality_comparable()
{
  return requires (T a, T b) {
    a == b; // Means a == b is valid syntax
    requires Convertible<decltype(a == b), bool>();
    a != b;
    requires Convertible<decltype(a != b), bool>();
  };
}
```

Constraints: Type Requirements

We can also write type requirements

```
template<typename I>
concept bool User_defined_iterator()
{
  return requires (I i) {
    typename I::iterator_category;
    {*i} -> const Value_type<I>&;
  };
}
```

Constraints: The Language

Constraints: how do they work?

- Language primitives
- Reduction
- Decomposition
- Overload resolution

Constraint Language

Formally, constraints are defined over a set *of atomic propositions*, connected by **&&** and **||**

```
is_lvalue_reference<T>::value && is_const<T>::value

is_integral<T>::value || is_floating_point<T>::value
```

Atomic Propositions

For the most part, any C++ expression that is not an **&&** or **||** expression

```
is_integral<T>::value
!is_void<T>::value
N == 2
0 < M
is_prime(N)
true
false
```

Calls to constraints are not atomic

Constraint Reduction

Function calls to constraints are *reduced* by inlining them into a `requires` clause

```
template<typename T>
concept bool Arithmetic()
{
  return is_integral<T>::value
      || is_floating_point<T>::value;
}
```

Constraint Reduction

Before:

```
template<typename T>
  requires Arithmetic<T>()
T do_math(T a, T b);
```

After:

```
template<typename T>
  requires is_integral<T>::value
        || is_floating_point<T>::value
T do_math(T a, T b);
```

Overload Resolution

Find candidates, instantiate templates

- Deduce template arguments
- ***Instantiate and check the constraints***
- Instantiate the declaration

Choose the best candidate

- Most specialized
- ***Most constrained***

Constraint Satisfaction

How do we determine if constraints are satisfied

- Constraints are just constant expressions
- Evaluate them!

Most specialized

Compare the types of function arguments of candidate functions, f1 and f2

Try to substitute argument types of f1 into f2 and vice versa

- If either succeeds than one is more specialized
- If neither succeeds, the overload is ambiguous
- What if both succeed?

Most Constrained

Given two constraints Γ and Δ, Γ **subsumes** Δ iff Γ contains all of Δ's propositions

- Solved as an application of first order logic
- Easily thought of as a subset problem

Given two declarations A and B *with equivalent type*, A is **more constrained** than B iff A's requirements (Γ) subsume B's (Δ)

Unconstrained templates are the ***least constrained***

Constraint Decomposition

Decomposed into sets of propositions through the application of sequent calculus for first order logic

Lists of atoms

Expressions

$$\frac{\Gamma, P, Q \vdash \Delta}{\Gamma, P \wedge Q \vdash \Delta} \qquad \frac{\Gamma, P \vdash \Delta \qquad \Gamma, Q \vdash \Delta}{P \vee Q \vdash \Delta}$$

Derivable

Conjunction (&&)

Disjunction (||)

Subsumption

Given a previously decomposed list of atomic propositions, Γ, determine if an expression *e* is valid (can be derived)

$$\frac{\Gamma \vdash P, \Delta \qquad \Gamma \vdash Q, \Delta}{\Gamma \vdash P \wedge Q, \Delta} \qquad \frac{\Gamma \vdash P, Q, \Delta}{\Gamma \vdash P \vee Q, \Delta}$$

Basically, search Γ for atoms in *e*

Comparing Constraints

```
template<typename T>
concept bool Advanceable()
{ return requires (T i) { ++i; }; }

template<typename T>
concept bool Incrementable()
{ return requires (T i) { ++i; i++; }; }
```

Does Advanceable subsume Incrementable?
Does Incrementable subsume Advanceable?
Proofs left to the viewer as an exercise

Notation

Variable Templates

Constraining Generic Lambdas

Constrained Auto

Terse Templates

Variable Templates

New in C++14, allows the variable templates:

```
template<Number T>
constexpr T min = numeric_limits<T>::min();

cout << min<int> << '\n';
cout << min<unsigned> << '\n';
```

Variable Templates and Constraints

Can use variable templates to define constraints

```
template<typename T>
concept bool Equality_comparable =
  requires(T a, T b) {
    {a == b} -> bool;
    {a != b} -> bool;
  };

template<typename T>
  requires Equality_comparable<T>
void f(T a, T b);
```

Generic Lambdas

New in C++14, generic lambdas

```
template<Container C>
void f(C& c)
{
  sort(c, [](auto x, auto y) { return x < y });
}
```

Types of **x**, **y** depend on arguments to, instantiation of **sort**

Lambda/Concepts Interaction

Eventually, we'd like separate checking of template definitions

Generic lambdas (as proposed for C++14) are unconstrained

There is some concern that widespread use of generic lambdas will cause code breakage when we eventually enable separate checking

Hopefully not a big deal

Constraining Generic Lambdas

We'd like to notation for adding constraints to generic lambdas

Lambda notation is ***terse***

- Constrained lambda notation should also be terse

That notation should be ***general*** and ***consistent***

- Lambdas are functions. What works for lambdas should also work for functions

Notation

But template syntax can be ***verbose***

From day #1 some (but not all) people have complained that the template syntax is verbose

- Novices seem to want "loud syntax", then feel comfortable having "the new" stand out
- Experts tire of repetitive syntax and find it distracting

Notation matters

- Optimized for the common case

Absurdly Verbose Constraints

```
template<typename F1, typename F2, typename O>
  requires Forward_iterator<F1>()
        && Forward_iterator<F2>()
        && Output_iterator<O>()
        && Assignable<Value_type<F1>, Value_type<O>>()
        && Assignable<Value_type<F2>, Value_type<O>>()
        && Comparable<Value_type<F1>, Value_type<F2>>()
void merge(F1 f1, F1 l1, F2 f2, F2 l2, O o);
```

Too verbose for templates, utterly absurd for lambdas

Making the Verbose Terse

Predicate abstraction to the rescue

```
template<typename F1, typename F2, typename O>
  requires Mergeable<F1, F2, O>
void merge(F1 f1, F1 l1, F2 f2, F2 l2, O o);
```

Still too verbose for lambdas.

```
[]<typename F1, typename F2, typename O>
  requires Mergeable<F1, F2, O>
(F1 f1, F1 l1, F2 f2, F2 l2, O o)
```

Plus it doesn't work – parsing ambiguity

Introducing Template Parameters

Allow template parameters to be ***introduced*** from a concept definition

```
template<Mergeable{F1, F2, O}>
void merge(F1 f1, F1 l1, F2 f2, F2 l2, O o);
```

Probably the best we can do for lambdas.

```
[]<Mergeable{F1, F2, O}>
(F1 f1, F1 l1, F2 f2, F2 l2, O o)
```

If your lambdas really look like this

Introduction Syntax

This:

```
template<Mergeable{F1, F2, O}>
void merge(F1 f1, F1 l1, F2 f2, F2 l2, O o);
```

Is equivalent to writing:

```
template<typename F1, typename F2, typename O>
  requires Mergeable<F1, F2, O>
void merge(F1 f1, F1 l1, F2 f2, F2 l2, O o);
```

Declarations with Type Concepts

Single-argument concepts (***type concepts***) are special. For example:

```
template<Sortable_container C>
void sort(C& cont);
```

We can make this even more terse:

```
void sort(Sortable_container& cont);
```

`Sortable_container` is a concept that introduces a ***named placeholder type***

Type Concepts and Lambdas

We can write a lambda like this:

```
[x]<Regular T>(T y) { return x == y; }
```

Or we can equivalently write:

```
[x](Regular y) { return x == y; }
```

Et Voila! Tersely constrained lambdas!

The Same-Type Problem

```
void sort(Random_access_iterator p,
          Random_access_iterator q);
```

Obviously **p** and **q** are the same type

- Their types have the same spelling

How do we guarantee that **p** and **q** are of the same type?

Same-type Substitution

When a concept is used as a type specifier for a parameter, all other uses are replaced by an implementation-defined type name

```
template<Random_access_iterator __R>
  void sort(__R p, __R q);
```

Don't want this behavior?

Use verbose notation and declare 2 parameters

Implementation

Two implementations:

- Initial prototype (GCC-4.8, from September)
- GCC Branch (based on 4.9)

Library support (Origin)

- https://github.com/asutton/origin
- Built against the GCC branch

Compiler Performance

Small test of constraints vs. type traits (emulation) for similar programs

- Tested using initial prototype (GCC-4.8)
- Performance gains increase with number of requirements checked
- Observed gains of 13-25% for even small numbers of requirements

Defining and instantiating type traits is expensive!

Library Support

All constraints for all concepts in Palo Alto TR (n3351)

- **`Equality_comparable`**, **`Totally_ordered`**, **`Regular`**, **`Function`**, **`Predicate`**, **`Relation`**
- **`Input_iterator`**, **`Forward_iterator`**, **`Bidirectional_iterator`**, **`Sortable`**

Some variations

Programming

Concept design

Fun with language features

Library Design

Concepts arise from common implementation patterns in concrete, and later abstract algorithms

Libraries should have relatively few concepts

- When compared to algorithms + data structures

Why?

- Easier to learn and remember
- Easier to write concise requirements

Concept Design

Concepts should describe an expressive computational basis [EoP]

- Require semantically related operators (e.g., **==** ***and*** **!=** for `Equality_comparable`)

Why?

- A concept establishes notation for a (mathematical?) domain
- Prefer to write in terms of that notation
- Fewer constraints on implementations
- Leads to fewer concepts

Generating Default Definitions

```
// In global namespace?
template<typename T>
  requires (T a, T b) { {a == b} -> Boolean; }
auto operator!=(T a, T b)
{
  return !(a == b);
}

class Date { ... };
bool operator==(Date, Date) { ... };

static_assert(Equality_comparable<Date>(), “”);
```

An Evolution Problem

Constraining templates can quietly change the results of overload resolution

```
void f(double); // #1

template<typename T>
void f(T x); // #2

f(0); // calls #2
```

An Evolution Problem

Constraining templates can quietly change the results of overload resolution

```
void f(double); // #1

template<Character T> // char, wchar_t, etc.
void f(T x); // #2

f(0); // calls #1 – not good!
```

Can we modify the library to ensure that overloads don't change unexpectedly?

Unconstrained Templates May Go

Delete the unconstrained template.

```
void f(double); // #1

template<typename T>
void f(T x) = delete; // #2

template<Character T> // char, wchar_t, etc.
void f(T x); // #3

f(0); // Error!
```

Conclusions

Concepts Lite

- **`enable_if`** on steroids
- Relies on **`constexpr`**, builds on existing features, practice
- Rooted in established theories of formal logic, languages

Future Work

Implement terse templates, constrained generic lambdas

More work on Origin, other libraries

Write the TS

Questions