Generic Programming with Concepts Lite

Andrew Sutton
University of Akron

Overview

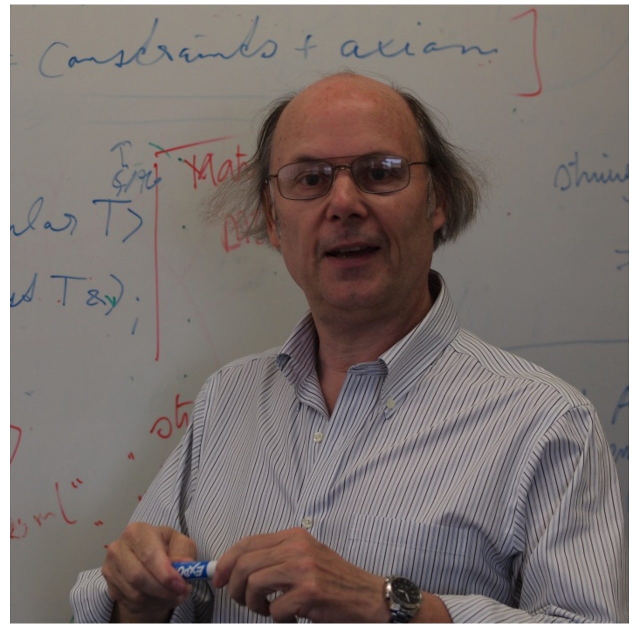
This talk is about the problems that concepts address

Improved compiler diagnostics: avoid template spew

Direct expression of intent: avoid clever idioms

Improved expressivity: better tools for extending definitions

Concepts design



Concepts

On its way to being an ISO Technical Specification (TS)

https://github.com/cplusplus/concepts-ts

Last publication was n4040

Will be a newer (and better) version for November

Not a good overview of concepts

Hopefully, will be approved as a PDTS in November

Prior documents

Stroustrup, Sutton, *A Concept Design for the STL*, n3551

Overview of approach taken to formulate concepts needed by STL algorithms

Sutton, Stroupstrup, Dos Reis, *Concepts Lite*, n3701

Overview of language features in Concepts TS

Implementation

Implementation based on GCC 4.10

http://gcc.gnu.org/svn/gcc/branches/c++-concepts/

Feature status

Implements almost every feature described in this talk

Two exceptions related to extended use of placeholder types

Implementation contributors

Braden Obrzut

Jason Merrill

Ville Voutilainen

Library support

Origin C++ Libraries – experimental library support

https://github.com/asutton

Library includes:

Concepts for all standard

Base set of concepts from N3351 (or similar)

Initial, experimental, changing concepts for ranges

Overview

First hour

Motivating constrained templates

Concepts Litest

Second hour

Constraints

Concepts

Libraries

Generic programming

"… a style of computer programming in which algorithms are written in terms of *to-be-specified-later* types that are then *instantiated* when needed for specific types provided as [arguments]"

Wikipedia

An algorithm in Python

Returns true if, for each element **x** in **seq**, **pred(x)** evaluates to **True**

```
def all(seq, pred):
  for x in seq:
    if not pred(x):
      return False;
  return True;
```

Is this generic?

Using the algorithm

```
def test(seq, fn): print all(seq, fn)

def is_even(n): return n % 2 == 0
def is_lower(c): return ‘a’ <= c and c <= ‘z’

test([0, 2], even)     # prints True
test((0, 1), even)     # prints False
test(“abc”, is_lower) # prints False
test([0, 2], True)     # Exception
```

Dynamic typing

Dynamically typed languages, type is part of the object

The algorithm is "instantiated" when it runs

- Names and operations resolved against the argument's dynamic type and content

"Duck" typing

Exceptions happen

```
Traceback (most recent call last):
 File "test.py", line 15, in <module>
    print test([0, 2], True)    # Exception!
  File "test.py", line 7, in test
    def test(seq, fn): return all(seq, fn)
  File "test.py", line 3, in all
    if not pred(x):
TypeError: 'bool' object is not callable
```

Type errors

In a generic program, a type error occurs when an argument is used in a way that is not supported

Errors are diagnosed at the point at which they occur in the program

- Always entails some kind of a stack

Translating to C++

Can we achieve the same level of simplicity?

At the same time improving

- Diagnostics
- Expressivity

Same algorithm in C++

C++ is statically typed, **seq**, **pred** and **x**, need types

```
bool all(const _____& seq, _____ pred) {
  for(const _____& x : seq)
    if (!pred(x)) return false;
  return true;
}
```

Fill in the blanks

Same algorithm in C++

C++ is statically typed, **seq**, **pred** and **x**, need types

```
bool all(const _____& seq, _____ pred) {
  for(const auto& x : seq)
    if (!pred(x)) return false;
  return true;
}
```

Fill in the blanks

Same algorithm in C++

C++ is statically typed, **seq**, **pred** and **x**, need types

```
bool all(const auto& seq, auto pred) {
  for(const auto& x : seq)
    if (!pred(x)) return false;
  return true;
}
```

Fill in the blanks

Concepts feature

The **auto** type-specifier is allowed in types of function parameters

```
void f1(auto x);

void f2(vector<auto>& v);

void f3(auto (auto::*mfp)(auto));
```

Concepts feature

The **auto** type-specifier is allowed in types of function parameters

```
template<typename T>
void f1(T x);
```

```
void f2(vector<auto>& v);
```

```
void f3(auto (auto::*mfp)(auto));
```

Concepts feature

The **auto** type-specifier is allowed in types of function parameters

```
template<typename T>
void f1(T x);

template<typename T>
void f2(vector<T>& v);


void f3(auto (auto::*mfp)(auto));
```

Concepts feature

The **auto** type-specifier is allowed in types of function parameters

```
template<typename T>
void f1(T x);

template<typename T>
void f2(vector<T>& v);

template<typename T1, typename T2, typename T3>
void f3(T1 (T2::*mfp)(T3));
```

Generic algorithm

Here is the version declared with **auto**

```
bool all(const auto& seq, auto pred) {
  for(const auto& x : seq)
    if (!pred(x)) return false;
  return true;
}
```

Generic algorithm

This is the template version of our algorithm

```
template<typename Seq, typename Fn>
bool all(const Seq& seq, Fn fn) {
  for(const auto& x : seq)
    if (!pred(x)) return false;
  return true;
}
```

Usage

```
void test(const auto& seq, auto fn) {
  cout << all(seq, fn) << '\n';
}

bool is_even(int);
bool is_lower(char);

test(vector<int>{0, 2}, is_even); // prints true
test(list<int>{0, 1}, is_even);   // prints true
test("abc", is_lower)             // prints false
test(vector<int>{0, 2}, true)     // error
```

Statically typed languages

In statically typed languages, instantiation happens at compiled time

Deduce the template arguments from the function arguments

Replace occurrences of template parameters with deduced template arguments

Create a new declaration (specialization) of the function from the substituted code

Type errors

```
test.cpp: In instantiation of
  ‘bool all(const auto:1&, auto:2)
  [with auto:1 = std::list<int>; auto:2 = bool]’:
test.cpp:14:48:   required from
  ‘void test(const auto:3&, auto:4)
  [with auto:3 = std::list<int>; auto:4 = bool]’
test.cpp:24:34:   required from here
test.cpp:9:17: error: ‘fn’ cannot be used as a
  function
     if (not fn(x)) return false;
                  ^
```

Type errors

```
test.cpp: In instantiation of
  'bool all(const auto:1&, auto:2)
  [with auto:1 = std::list<int>; auto:2 = bool]':
test.cpp:14:48:   required from
  'void test(const auto:3&, auto:4)
  [with auto:3 = std::list<int>; auto:4 = bool]'
test.cpp:24:34:   required from here
test.cpp:9:17: error: 'fn' cannot be used as a
  function
     if (not fn(x)) return false;
               ^
```

Type errors

```
test.cpp: In instantiation of
  ‘bool all(const auto:1&, auto:2)
  [with auto:1 = std::list<int>; auto:2 = bool]’:
test.cpp:14:48:   required from
  ‘void test(const auto:3&, auto:4)
  [with auto:3 = std::list<int>; auto:4 = bool]’
test.cpp:24:34:   required from here
test.cpp:9:17: error: ‘fn’ cannot be used as a
  function
     if (not fn(x)) return false;
                 ^
```

Type errors

```
test.cpp: In instantiation of
  ‘bool all(const auto:1&, auto:2)
  [with auto:1 = std::list<int>; auto:2 = bool]’:
test.cpp:14:48:   required from
  ‘void test(const auto:3&, auto:4)
  [with auto:3 = std::list<int>; auto:4 = bool]’
test.cpp:24:34:   required from here
test.cpp:9:17: error: ‘fn’ cannot be used as a
  function
     if (not fn(x)) return false;
                  ^
```

Type errors

```
test.cpp: In instantiation of
  ‘bool all(const auto:1&, auto:2)
  [with auto:1 = std::list<int>; auto:2 = bool]’:
test.cpp:14:48:   required from
  ‘void test(const auto:3&, auto:4)
  [with auto:3 = std::list<int>; auto:4 = bool]’
test.cpp:24:34:   required from here
test.cpp:9:17: error: ‘fn’ cannot be used as a
  function
     if (not fn(x)) return false;
                 ^
```

Type errors

Same problem as with Python program

Instantiation stack instead of runtime stack

Type errors

Make writing, maintaining programs difficult

Can sometimes lead to subtle bugs

Preventing type errors

We can explicitly check types

```
def all(seq, pred):
  assert isinstance(seq, types.ListType)
  assert isinstance(pred, types.FunctionType)
  ...

all_of([0, 2], is_even) # OK
all_of([0, 2], True)    # Exception
```

This is good, right?

Not so much…

A whole lotta problems:

- Point of assertion is inside the function definition, so you don't know there's an error until its too late
- Interface is part of the implementation
- Can't effectively extend the definition to new types
- We're not checking the right properties of argument types

Type assertions

```
Traceback (most recent call last):
  File "check1.py", line 16, in <module>
    test([0, 2], True)    # Exception!
  File "check1.py", line 8, in test
    def test(seq, fn): all(seq, fn)
  File "check1.py", line 6, in all
    assert isinstance(pred, types.FunctionType)
AssertionError
```

Static assertions

In C++, **`static_assert`** is used to similar effect

```
template<typename Seq, typename Fn>
bool all(const auto& seq, auto fn) {
  static_assert(Sequence<Seq>{}, “bad sequence”);
  static_assert(Predicate<Fn>{}, “bad predicate”);
  for(const auto& x : seq)
    if (!pred(x)) return false;
  return true;
}
```

Is this any better?

For reference

What are **Sequence** and **Predicate**?

```
template<typename T>
struct Sequence : std::false_type { };

template<typename T>
struct Predicate : std::false_type { };
```

Static assertions

Can actually end up being a little worse

- Emit diagnostics for static assertion
- Emit diagnostics for every other type error in the body of that function

Static assertions

How bad can things get?

```
all(0, true);
```

Pretty bad

```
test.cpp: In instantiation of 'bool all(const auto:1&, auto:2) [with auto:1 = int; auto:2 = bool]':
test.cpp:25:14:   required from here
test.cpp:16:3: error: static assertion failed: not a sequence
   static_assert(is_sequence(seq), "not a sequence");
   ^
test.cpp:17:3: error: static assertion failed: not a predicate
   static_assert(is_predicate(fn), "not a predicate");
   ^
test.cpp:18:3: error: 'begin' was not declared in this scope
   for(const auto& x : seq)
   ^
test.cpp:18:3: note: suggested alternatives:
In file included from .../c++/4.10.0/bits/basic_string.h:42:0,
                 from .../c++/4.10.0/string:52,
                 from .../c++/4.10.0/bits/locale_classes.h:40,
                 from .../c++/4.10.0/bits/ios_base.h:41,
                 from .../c++/4.10.0/ios:42,
                 from .../c++/4.10.0/ostream:38,
                 from .../c++/4.10.0/iostream:39,
                 from test.cpp:2:
.../c++/4.10.0/initializer_list:89:5: note:   'std::begin'
     begin(initializer_list<_Tp> __ils) noexcept
     ^
.../c++/4.10.0/initializer_list:89:5: note:   'std::begin'
test.cpp:18:3: error: 'end' was not declared in this scope
   for(const auto& x : seq)
   ^
test.cpp:18:3: note: suggested alternatives:
In file included from .../c++/4.10.0/bits/basic_string.h:42:0,
                 from .../c++/4.10.0/string:52,
                 from .../c++/4.10.0/bits/locale_classes.h:40,
                 from .../c++/4.10.0/bits/ios_base.h:41,
                 from .../c++/4.10.0/ios:42,
                 from .../c++/4.10.0/ostream:38,
                 from .../c++/4.10.0/iostream:39,
                 from test.cpp:2:
.../c++/4.10.0/initializer_list:99:5: note:   'std::end'
     end(initializer_list<_Tp> __ils) noexcept
     ^
.../c++/4.10.0/initializer_list:99:5: note:   'std::end'
test.cpp:19:17: error: 'fn' cannot be used as a function
     if (not fn(x))
                 ^
```

Not so bad...

```
static_assert.cpp: In instantiation of
  ‘bool all(const auto:1&, auto:2)
  [with auto:1 = int; auto:2 = bool]’:
static_assert.cpp:25:14:   required from here
static_assert.cpp:16:3: error:
   static assertion failed: bad sequence
   static_assert(Sequence<Seq>{}, “...”);
   ^
static_assert.cpp:17:3: error:
   static assertion failed: bad predicate
   static_assert(Predicate<Fn>{}, “...”);
   ^
```

Integers are not ranges

```
test.cpp:18:3: error: ‘begin’ was not declared in
  this scope
   for(const auto& x : seq)
   ^
test.cpp:18:3: note: suggested alternatives:
In file included from .../c++/4.10.0/bits/
  basic_string.h:42:0,
                 from .../c++/4.10.0/ostream:38,
                 ...
                 from test.cpp:2:
.../c++/4.10.0/initializer_list:89:5: note:
  ‘std::begin’
     begin(initializer_list<_Tp> __ils) noexcept
     ^
```

Integers are not ranges

```
test.cpp:18:3: error: 'end' was not declared in this
   scope
    for(const auto& x : seq)
    ^
test.cpp:18:3: note: suggested alternatives:
In file included from .../c++/4.10.0/bits/
   basic_string.h:42:0,
                 from .../c++/4.10.0/ostream:38,
                 ...
                 from test.cpp:2:
.../c++/4.10.0/initializer_list:99:5: note:
   'std::end'
     end(initializer_list<_Tp> __ils) noexcept
     ^
```

Boolean values aren't callable

```
test.cpp:19:17: error: ‘fn’ cannot be used as a
   function
      if (not fn(x))
```

Just in case you didn't know…

Preventing type errors

Asserting type properties still causes the entire stack to be printed as a diagnostic

How do you avoid this?

- Specify type requirements as constraints on a declaration
- Check constraints at the point of use

Lifting constraints

Before: assertion of type constraints

```
template<typename Seq, typename Fn>
bool all(const Seq& seq, Fn fn) {
  static_assert(Sequence<Seq>{}, “”);
  static_assert(Predicate<Fn>{}, “”);
  for(const auto& x : seq)
    if (!pred(x)) return false;
  return true;
}
```

Lifting constraints

Next: rewrite constraints as conjunctions

```
template<typename Seq, typename Fn>
bool all(const Seq& seq, Fn fn) {
  static_assert(Sequence<Seq>{}
             && Predicate<Fn>{}, “”);
  for(const auto& x : seq)
    if (!pred(x)) return false;
  return true;
}
```

Lifting constraints

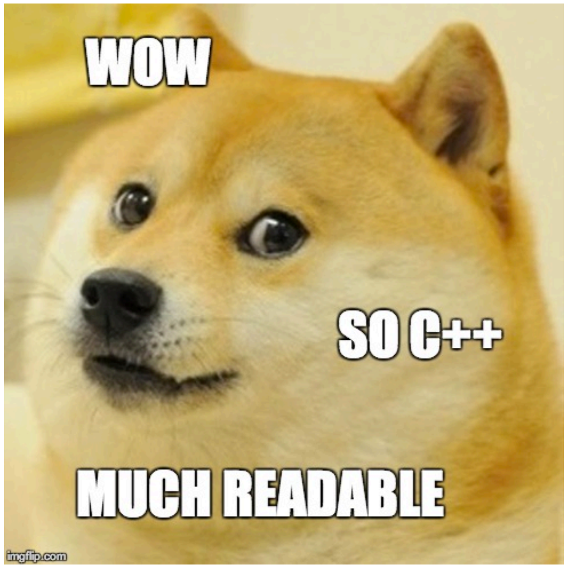
Next: lift constraints into the declaration

```
template<typename Seq, typename Fn>
typename std::enable_if<
  Sequence<Seq>{} && Predicate<Fn>{}, bool
>::type
all(const Seq& seq, Fn fn) {
for(const auto& x : seq)
    if (!pred(x)) return false;
  return true;
}
```

Lifting constraints

Next: lift constraints into the declaration

```
template<typename Seq, typename Fn>
typename std::enable_if<
  Sequence<Seq>{} && Predicate<Fn>{}, bool
>::type
all(const Seq& seq, Fn fn) {
for(const auto& x : seq)
    if (!pred(x)) return false;
  return true;
}
```



It works…

… but leaves much to be desired

Requires deep knowledge of the template system

Hard to use with class templates

Doesn’t work well with member functions

Overloading limited to true/false conditions

Non-intuitive, hard to read and write

Does not significantly improve diagnostics

Slower compile times

We can do better

We want to directly state state requirements as part of the declaration

Simplified and direct expression of intent

We want to overload declarations based on those requirements and support open extension

We want improved diagnostics

We can do better

Adds 0 runtime overhead

Do not significantly impact compile times

Make it faster if we can!

Do not increase the size of binaries

Template constraints

Declaration before concepts:

```
template<typename Seq, typename Fn>
typename std::enable_if<
  Sequence<Seq>{} && Predicate<Fn>{}, bool
>::type
all(const Seq& seq, Fn fn) {
for(const auto& x : seq);
```

Template constraints

Declaration with concepts

```
template<typename Seq, typename Fn>
  requires Sequence<Seq>{} && Predicate<Fn>{}
bool all_of(const Seq& seq, Fn fn) {
for(const auto& x : seq);
```

Template constraints

Declaration with concepts

```
template<typename Seq, typename Fn>
  requires Sequence<Seq>{} && Predicate<Fn>{}
bool all_of(const Seq& seq, Fn fn) {
for(const auto& x : seq);
```

We'll make this even more concise later

Requires clause

Allows the specification of constraints on template arguments

Allowed on any template or function declaration

Followed by a *constraint-expression*

Constraints are checked at the point of use, during lookup

Constraint satisfaction

A constraint is *satisfied* if and only if it evaluates to **`true`**

If a substitution failure occurs when processing a constraint, it the same as evaluating to **`false`**

Usage

```
all(vector<int>{0, 2}, true); // error
```

Error occurs at the point of use: no instantiation stack

```
req1.cpp: In function ‘int main()’:
req1.cpp:24:30: error: cannot call function
  ‘bool all(const Seq&, Fn)
  [with Seq = std::vector<int>; Fn = bool]’
   all(vector<int>{0, 2}, true);
                              ^
```

Constraint diagnostics

```
req1.cpp:16:10: note:   constraints not satisfied
  [with Seq = std::vector<int>; Fn = bool]
     bool all(const Seq& seq, Fn fn) {
          ^
req1.cpp:16:10: note:   ‘Sequence<Seq>{}’ evaluated
  to false
req1.cpp:16:10: note:   ‘Predicate<Fn>{}’ evaluated
  to false
```

Constraint diagnostics

Based on the *contents* of the *constraint-expression*

Can provides extremely accurate diagnostics

```
req1.cpp:16:10: note:   'Sequence<Seq>{}' evaluated
  to false
req1.cpp:16:10: note:   'Predicate<Fn>{}' evaluated
  to false
```

Requirements

Surely, we're not limited to constraining function templates?

Where can else can we write requirements?

Note: the following examples use type traits from the standard library

Requirements go here

Class templates!

```
template<typename T>
  requires std::is_object<T>{}
class vector;
```

Requirements go here

Partial specializations of class templates!

```
template<typename T>
  requires std::is_integral<T>{}
class complex<T>; // Note: specialization arguments
```

Requirements go here

Alias templates!

```
template<typename T>
  requires std::integral_type<T>{}
using Unsigned = typename std::make_unsigned_t<T>;
```

Requirements go here

Variable templates!

```
template<typename T>
  requires std::is_arithmetic<T>{}
constexpr T min = numeric_limits<T>::min();
```

Requirements go here

Member functions!

```
template<typename T, int N>
struct Array {
  void fill(const T&)
    requires std::is_copy_assignable<T>{};
};
```

Requirements go here

Member functions definitions!

```
template<typename T, int N>
void array<T, N>::fill(const T& value)
  requires std::is_copy_assignable<T>{}
{
  std::fill(begin(), end(), value);
}
```

Requirements go here

Normal functions!

```
void f() requires sizeof(int) == 4;
```

Requirements go here

Normal functions!

```
void f() requires sizeof(int) == 4;
```

Note: This is not “static if”

Checking constraints

Checking constraints always happens on lookup

- During lookup for classes, variables, aliases
- During function overload resolution for functions

Diagnostics emitted when lookup fails due constraint failures

Checking constraints

Possible gotcha for people expecting this to work

```
template<typename T>
  requires std::is_arithmetic<T>{}
struct Limits;

template<>
struct Limits<Big_int> { ... }; // error
```

Can’t avoid checks in this way; keeps you honest

Checking constraints

```
limits.cpp:11:22: error: template constraint failure
 struct Limits<Big_int> { ... };
                     ^
limits.cpp:11:22: note:   constraints not satisfied
  [with T = Big_int]
limits.cpp:11:22: note:
  ‘std::is_arithmetic<_Tp>{}’ evaluated to false
```

Checking constraints

Another possible gotcha:

```
template<typename T>
  requires std::is_object<T>{}
struct Vector;

Vector<int&>* v; // error
```

Checks are made even when complete types are not required

Checking constraints

```
compl.cpp:8:12: error: template constraint failure
 Vector<int&>* v;
           ^
compl.cpp:8:12: note:   constraints not satisfied
  [with T = int&]
compl.cpp:8:12: note:
  ‘std::is_object<_Tp>{}’ evaluated to false
```

Summary

Deferred type checking for templates results in

Bad diagnostics
Buried interfaces
Closed for extension

Concepts

Writing constraints as part of a declaration
Checking those constraints at the point of use

Summary

Everything presented up to this point is essentially the first iteration of Concepts Lite (or Concepts Litest)

No support for defining concepts

No support for overloading based on constraints

An all around replacement for **`enable_if`**

Next hour

More about constraints

Concept definitions

Systems of concepts

- Overloading and specialization

Questions