C++14 Concepts

Bjarne Stroustrup

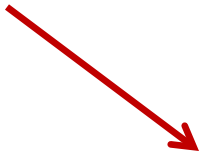
Texas A&M University

www.stroustrup.com

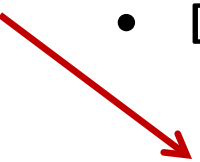
Templates

- Primary aim: to support efficient generic programming
  - Uncompromised generality
    - Should do far more than I could imagine
  - Uncompromised performance
    - User-defined vector should compete with built-in array
  - Good interfaces
    - Well, two out of three ain't bad ☹
- Provides compile-time duck typing

Duck Typing is Insufficient



- There are no proper interfaces
- Leaves error detection far too late
  - Compile- and link-time in C++
- Encourages a focus on implementation details
  - Entangles users with implementation
- Leads to over-general interfaces and data structures
  - As programmers rely on exposed implementation “details”
- Does not integrate well with other parts of the language
  - Teaching and maintenance problems



- We must think of generic code in ways similar to other code
  - Relying on well-specified interfaces (like OO, etc.)

Generic Programming is “just” Programming

- *Traditional code*

```
double sqrt(double d);      // C++84: accept any d that is a double
double d = 7;
double d2 = sqrt(d);        // fine: d is a double
double d3 = sqrt(&d);       // error: &d is not a double
```

- Generic code

```
void sort(Container& c);    // C++14: accept any c that is a Container
vector<string> vs { “Hello”, “new”, “World” };
sort(vs);                   // fine: vs is a Container
sort(&vs);                  // error: &vs is not a Container
```

Remember C++0x Concepts?

- Could express requirements of all standard library algorithms
  - Could check calls
  - Could check definitions
  - Could map names in calls
  - Was object-oriented in nature
    - Somewhat similar to Haskell type classes (but more general)
  - Is dead
- A debacle of complexity
  - 120 “concepts” in the standard library
  - 73 pages of specification (more than C++85)
  - Compilation required heroic efforts
    - To re-gain run-time performance (done)
    - To re-gain compilation speed (not done)
  - Not as general/flexible as I would like
  - Parts, I couldn’t understand

Back to square #1

- First
  - What are concepts?
  - What concepts are there?
  - How do we use concepts?
- Finally
  - what language support do we need?
  - What language support can we afford
    - No runtime overhead
      - done
    - Max 20% compile-time overhead
      - We do much better than that: faster than workarounds

C++14: Constraints aka “Concepts lite”

- How do we specify requirements on template arguments?
  - state intent
    - Explicitly states requirements on argument types
  - provide point-of-use checking
    - No checking of template definitions
  - use constexpr functions
- Voted as C++14 Technical Specification
- Design by Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Sutton
- Implemented by Andrew Sutton in GCC
- There are no C++0x concept complexities
  - No concept maps
  - No new syntax for defining concepts
  - No new scope and lookup issues

What is a Concept?

- Concepts are fundamental
  - They represent fundamental concepts of an application area
  - Concepts are come in “clusters” describing an application area
- A concept has semantics (meaning)
  - Not just syntax
  - Operations are related (e.g., +, -, *, and %)
  - “**Subtractable**” is not a concept
- We have always had concepts
  - C++: Integral, arithmetic
  - STL: forward iterator, predicate
  - Informally: Container, Sequence

What is a Concept?

- A concept is ***not*** the minimal requirements for an implementation
  - An implementation does not define the requirements
  - Requirements should be stable
- Concepts support interoperability
  - There are relatively few concepts
  - We can remember a concept

C++14 Concepts (Constraints)

- A concept is a predicate on one or more arguments
  - E.g. **Sequence<T>()** **//** *is T a Sequence?*
- Template declaration

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

- Template use

```
void use(vector<string>& vs)
{
        auto p = find(vs,"Jabberwocky");
        // …
}
```

C++14 Concepts: “Shorthand Notation”

- Shorthand notation

```
template <Sequence S, Equality_comparable<Value_type<S>> T>
        Iterator_of<C> find(S& seq, const T& value);
```

- We can handle essentially all of the Palo Alto TR
  - (STL algorithms) and more
    - Except for the axiom parts
  - We see no problems checking template definitions in isolation
    - But proposing that would be premature (needs work, experience)
  - We don’t need explicit **requires** much (the shorthand is usually fine)

C++14 Concepts: Error handling

- Error handling is simple (and fast)

```
template<Sortable Cont>
        void sort(Cont& container);

vector<double> vec {1.2, 4.5, 0.5, -1.2};
list<int> lst {1, 3, 5, 4, 6, 8,2};

sort(vec);      // OK: a vector is Sortable
sort(lst);      // Error at (this) point of use: Sortable requires random access
```

- Actual error message

  error: ‘list<int>’ does not satisfy the constraint ‘Sortable’

C++14 Concepts: Overloading

- Overloading is easy

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

template<Associative_container C>
        Iterator_type<C> find(C& assoc, const Key_type<C>& key);

vector<int> v { /* ... */ };
multiset<int> s { /* ... */ };
auto vi = find(v, 42);                  // calls 1st overload:
                                        // a vector is a Sequence
auto si = find(s, 12-12-12);            // calls 2nd overload:
                                        // a multiset is an Associative_container
```

C++14 Concepts: Overloading

- Overloading based on predicates
  - specialization based on subset
  - Far easier than writing lots of tests

```
template<Input_iterator I>
        void advance(I& i, Difference_type<I> n) { while (n--) ++i; }

template<Bidirectional_iterator I>
        void advance(I& i, Difference_type<I> n)
        { if (n > 0) while (n--) ++i; if (n < 0) while (n++) --i; }

template<Random_access_iterator I>
        void advance(I& i, Difference_type<I> n) { i += n; }
```

- We don't say

  Input_iterator < Bidirectional_iterator < Random_access_iterator

  we compute it

C++14 Concepts: Definition

- How do you write constraints?
  - Any **bool** expression
    - Including type traits and constexpr function
  - a **requires(expr)** compile time intrinsic function
    - **true** if **expr** is a valid expression
  - To recognize a concept syntactically, we can declare it **concept**
    - Rather than just **constexpr**

Generic (Polymorphic) Lambdas

- Lambdas are closely related t templates
  - You can think of a generic lambda as a template
- Check
  - Unconstrained lambda + unconstrained template argument
    - => late checking
    - you're on your own
  - Unconstrained lambda + constrained template argument
    - => Use constraint from template
  - Constrained lambda + unconstrained template argument
    - => Use constraint from lambda
  - Constrained lambda + constrained template argument
    - => use (constraint from lambda && constraint from template)

C++14 Concepts: “Terse Notation”

- We can use a concept name as the name of a type than satisfy the concept

```
void sort(Container& c);            // terse notation
```

  - means

```
template<Container __Cont>          // shorthand notation
    void sort(__Cont& c);
```

  - means

```
template<typename __Cont>           // explicit use of predicate
    requires Container<__Cont>()
        void sort(__Cont)& c;
```

  - Accepts any type that is a Container
    - **vector<string> vs;**
    - **sort(vs);**

C+14 Concepts: “Terse Notation”

- We have reached the conventional notation
  - with the conventional meaning

```
void sort(Container& c);     // accept any c that is a Container
vector<string> vs { “Hello”, “new”, “World” };
sort(vs);                    // fine: vs is a Container
sort(&vs);                   // error &vs is not a Container

double sqrt(double d);       // accept any d that is a double
double d = 7;
double d2 = sqrt(d);         // fine: d is a double
double d3 = sqrt(&d);        // error: &d is not a double
```

C+14 Concepts: “Terse Notation”

- Consider **std::merge**:

```
template<typename For,
         typename For2,
         typename Out>
   requires Forward_iterator<For>()
      && Forward_iterator<For2>()
      && Output_iterator<Out>()
      && Assignable<Value_type<For>,Value_type<Out>>()
      && Assignable<Value_type<For2,Value_type<Out>>()
      && Comparable<Value_type<For>,Value_type<For2>>()
void merge(For p, For  q, For2 p2, For2 q2, Out p);
```

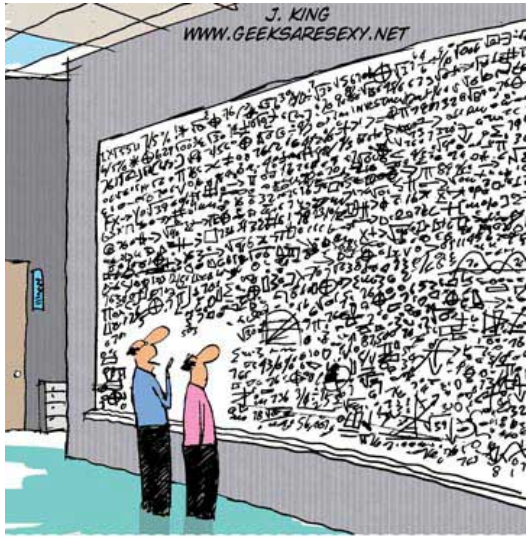
- Headache inducing, and **accumulate()** is worse

C+14 Concepts: “Terse Notation”

- Better:

```
template<Forward_iterator For,
         Forward_iterator For2,
         Output_iterator Out>
              requires Mergeable<For,For2,Out>()
void merge(For p, For q, For2 p2, For2 q2, Out p);
```

- Quite readable



“...And that, in simple terms, is what’s wrong with your software design.”

C+14 Concepts: “Terse Notation”

- Better still:

```
Mergeable{For,For2,Out}
void merge(For p, For  q, For2 p2, For2 q2, Out p);
```

- The

  concept-name { identifier-list }

  notation introduces constrained names

C+14 Concepts: “Terse Notation”

- Now we just need to define **Mergeable**:

```
template<typename T1,T2,T3>
concept bool Mergeable()
{
        return Forward_iterator<For>()
          && Forward_iterator<For2>()
          && Output_iterator<Out>()
          && Assignable<Value_type<For>,Value_type<Out>>()
          && Assignable<Value_type<For2,Value_type<Out>>()
          && Comparable<Value_type<For>,Value_type<For2>>();
 }
```

- It’s just a predicate

“Paradigms”

- Much of the distinction between object-oriented programming, generic programming, and “conventional programming” is an illusion
  - based on a focus on language features
  - incomplete support for a synthesis of techniques
  - The distinction does harm
    - by limiting programmers, forcing workarounds

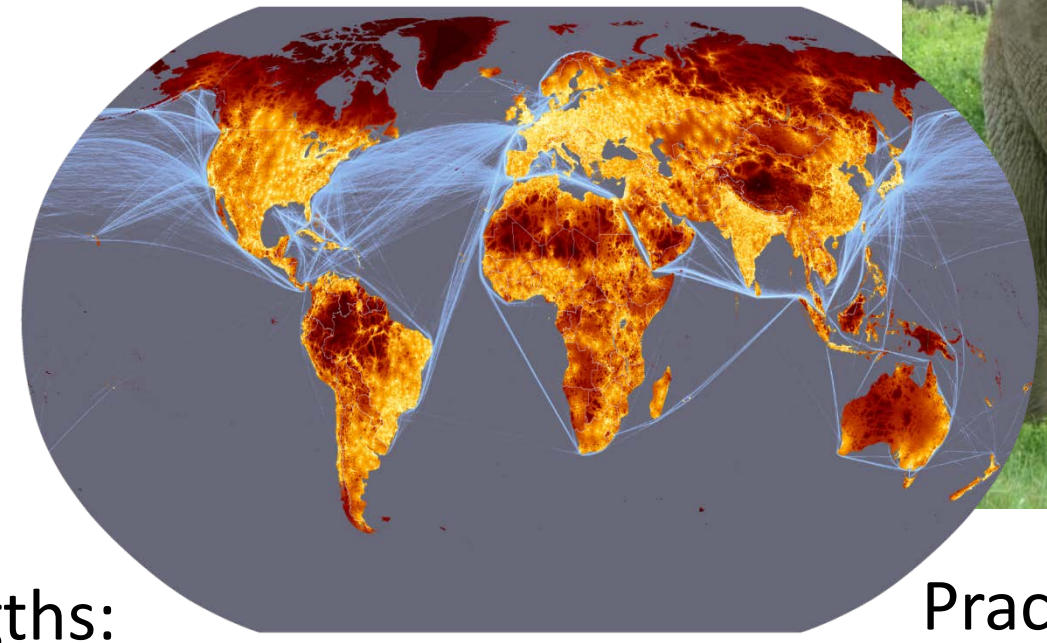
```
void draw_all(Container& c)     // is this OOP, GP, or conventional?
{
    for_each(c, [](Shape* p) { p->draw(); } );
}
```

Reading

- A. Sutton, B. Stroustrup, G. Dos Reis: Concepts Lite: Constraining Templates with Predicates. N3580. (current draft)
- B. Stroustrup and A. Sutton: A Concept Design for the STL. N3351==12-0041. (“Palo Alto TR”)
- Andrew Sutton and Bjarne Stroustrup: Design of Concept Libraries for C++. Proc. SLE 2011 (International Conference on Software Language Engineering). July 2011.

Questions?

C++: A light-weight abstraction programming language



Key strengths:

- software infrastructure
- resource-constrained applications

Practice type-rich programming