INTRODUCTION TO CONCEPTS

By Avi Lachmish

BACKGROUND

- In about 1987, Stroustrup tried to design templates with proper interfaces [Str94]. He failed. He wanted three properties for templates:
  - Full generality/expressiveness
  - Zero overhead compared to hand coding
  - Well-specified interfaces
- Then, nobody could figure out how to get all three, so we got
  - Turing completeness
  - Better than hand-coding performance
  - Lousy interfaces (basically compile-time duck typing)

GENERIC PROGRAMING

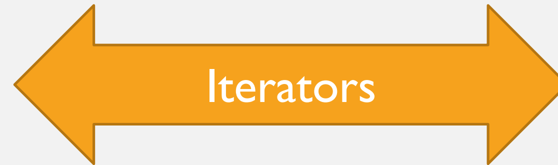
- A methodology for the development of reuse software

- Characteristics:
  - Reusable
  - Composable
  - Efficient

THE STL - TODAY

- The C++ STL embodies generic programming
  - Tree different kind of components

Containers



Algorithms

- C++ template enable the application of GP
  - Overloading permits natural abstractions
  - Instantiation emits the cost of abstraction
- Significant problems:
  - Inability to directly express ideas of GP
  - GP is fragile

FRAGILE C++ TEMPLATES

```
template <typename InputIterator, typename T>
InputIterator find(InputIterator first, InputIterator last, const T& value) {
    while( first < last && !(*first == value) ) {
        ++first;
    }
   return first;
}
```

- Using it

```
std::vector<int> v = {1,2,3};

find(v.begin(), v.end(), 17); // okay


std::list<int> l = {1,2,3};

find(l.begin(), l.end(), 17); // error!
```

FRAGILE C++ TEMPLATES

```
template <typename InputIterator, typename T>
InputIterator find(InputIterator first, InputIterator last, const T& value) {
    while( first < last && !(*first == value) ) {
        ++first;
    }
    return first;
}
```

- The error was in the definition of the template:
  - But it was found by an unlucky user
  - < is not part of the InputIterator **concepts**

Who is accountable for the error the using app or the GP class

FRAGILE C++ TEMPLATES

```
template <typename InputIterator, typename T>
InputIterator find(InputIterator first, InputIterator last, const T& value) {
    while( first < last && !(*first == value) ) {
        ++first;
    }
    return firs
}
```

Can we design better support for generic programming in C++ without sacrificing the performance and flexibility of templates

- The error was in the definition of the template:
  - But it was found by an unlucky user
  - < is not part of the InputIterator **concepts**

Who is accountable for the error the using app or the GP class

GENERIC PROGRAMMING

```
// Traditional code:
double sqrt(double d); // accept any d that is a double

double d = 7;
double d2 = sqrt(d); // fine: d is a double

vector<string> vs = { "Good", "old", "templates" };
double d3 = sqrt(vs); // error: vs is not a double
```

GENERIC PROGRAMMING - IN CONTRAST

```
template<class T> void sort(T& c) { // C++98: accept a c of any type T
// code for sorting (depending on various properties of T,
// such as having [] and a value type with <
}

vector<string> vs = { "Good", "old", “templates" };
sort(vs); // fine: vs happens to have all the syntactic properties required by sort

double d = 7;
sort(d); // error: d doesn’t have a [] operator
```

GENERIC PROGRAMMING - WE HAVE PROBLEMS

- As you probably know, the error message we get from **sort(d)** is verbose and nowhere near as precise and helpful as my comment might suggest.
- To use **sort**, we need to provide its definition, rather than just its declaration, this differs from ordinary code and changes the model of how we organize code.
- The requirements of **sort** on its argument type are implicit ("hidden") in its function body.
- The error message for **sort(d)** will appear only when the template is instantiated, and that may be long after the point of call.
- The **template<typename T>** notation is unique, verbose, repetitive, and widely disliked.

USING CONCEPTS

- A concept is a compile-time predicate (that is, something that yields a Boolean value). For example, a template type argument, **T**, could be required to be
  - an iterator: **Iterator<T>**
  - a random access iterator: **RandomAccessIterator<T>**
  - a number: **Number<T>**
- The notation **C<T>** where **C** is a concept and **T** is a type is an expression meaning "**true** if **T** meets all the requirements of **C** and **false** otherwise."
- Concept are very expressive and nicely cheap to compile (cheaper than template metaprogramming workarounds)
- Concepts enable overloading
- Significantly simplifying metaprogramming as well as generic programming

USING CONCEPTS - SPECIFYING TEMPLATE INTERFACES

- Consider a variant of **std::find()**

```
template <typename S, typename T>
        requires Sequence<S> && EqualityComparable<S::value_type ,T>
S::iterator find(S& seq, const T& value);
```

- This is a template that takes two template type arguments (nothing new here).
- The first template argument must be a sequence (**Sequence<S>**) and we have to be able to compare elements of the sequence to **value** using the **==** operator (**EqualityComparable<Value_type<S>,T>**).
- This **find()** takes its sequence by reference and the value to be found as a **const** reference. It returns an iterator (nothing new here).

USING CONCEPTS - SPECIFYING TEMPLATE INTERFACES

- try to use this **find()**

```
void use(vector<string>& vs, list<double>& lstd)
{
        auto p0 = find(vs,"Waldo");         // OK
        auto p1 = find(vs,0.5772);          // error: can’t compare a string and a double
        auto p2 = find(lstd,0.5772);        // OK
        auto p3 = find(lstd, "Waldo");      // error: can’t compare a double and a string
}
```

USING CONCEPTS - SHORTHAND NOTATION

```
template<typename Seq>
        requires Sequence<Seq>
void algo(Seq& s);
```

- That is, we need an argument of type **Seq** that must be a **Sequence**. I did that by saying "The template takes a type argument; that type argument must be a **Sequence**." That's a bit verbose.

USING CONCEPTS - SHORTHAND NOTATION

```
template<typename Seq>
        requires Sequence<Seq>
void algo(Seq& s);
```

- That is, we need an argument of type **Seq** that must be a **Sequence**. I did that by saying "The template takes a type argument; that type argument must be a **Sequence**." That's a bit verbose.

- We say "The template takes a **Sequence** argument" and write this

```
template<Sequence Seq>
void algo(Seq& s);
```

USING CONCEPTS - SHORTHAND NOTATION

```
template<C T>
```

- means

```
template<typename T> requires C<T>
```

- So our find function

```
template <typename S, typename T>
        requires Sequence<S> && Equality_comparable<S::value_type ,T>
S::iterator find(S& seq, const T& value);
```

- means

```
template <Sequence S, typename T>
        Equality_comparable<S::value_type ,T>
S::iterator find(S& seq, const T& value);
```

DEFINING CONCEPTS

- Often, you'll find useful concepts, such as **EqualityComparable** in libraries (e.g., the Ranges TS [Nie15])

```
template<typename T>
concept bool EqualityComparable =
        requires (T a, T b) {
                { a == b } -> bool;  // compare Ts with ==
                { a != b } -> bool;  // compare Ts with !=
};
```

- The **EqualityComparable** concept is defined as a variable template. To be **EqualityComparable**, a type **T** must provide **==** and **!=** operations that each must return a **bool** (technically, "something convertible to **bool**"). The **requires** expression allows us to directly express how a type can be used:
  - **{ a == b }** says that two **T**s should be comparable using **==**.
  - **{ a == b } -> bool** says that the result of such a comparison must be a **bool** (technically, "something convertible to **bool**").

DEFINING CONCEPTS

- A **requires** expression is never actually executed. Instead, the compiler looks at the requirements listed and returns **true** if they would compile and **false** if not. This is obviously a very powerful facility. To learn the details, I recommend Andrew Sutton's paper [Sut16]. Here, I'll just show examples

```
template<typename T> concept bool Sequence =
        requires(T t) {
        typename T::value_type;              // must have a value type
        typename T::iterator;                // must have an iterator type
        { begin(t) } -> T::iterator;         // must have begin()
        { end(t) } -> T::iterator;           // must have end()
        requires Input_iterator<T::iterator>;
        requires Same_type<T::value_type, T::iterator:: value_type>;
};
```

DEFINING CONCEPTS

- Now, finally, we can do **Sortable,** To be sortable, a type must be a sequence offering random access and with a value type that supports **<** comparisons

```
template<typename T> concept bool Sortable =
        Sequence<T> &&
        Random_access_iterator<T::iterator> &&
        Less_than_comparable<T::value_type>;
```

- There are many concepts already declared as part of Ranges TS

DEFINING CONCEPTS

- Often, we want to make requirements on the relationship among concepts. For example

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

- This allows comparing **int**s to **double**s and **string**s to **char***s, but not **int**s to **string**s

DESIGNING WITH CONCEPTS

- What makes a good concept?

```
template<typename T>
concept bool Drawable = requires(T t) { t.draw(); } ;

class Shape { void draw(); };

class Cowboy {void draw(); };

template<Drawable D> void draw_all(vector<D*>& v) {
        for (auto x : v) v->draw();
}
```

- Ask yourself: What fundamental concept does “has a draw member function taking no argument” represent?
  - There is no good answer (A cowboy might make a good concept in a games context and a drawable shape a good concept in a graphics context)
- A shape has several more essential properties than just “can be drawn” (e.g. “has location”, “can be moved” and “can be hidden”)
- a cowboy (e.g., “can ride a horse”, “likes booze”, and “can die”)
- A rule of thumb is to avoid “single property concepts.” For that reason, **Drawable** is instantly suspicious.

DESIGNING WITH CONCEPTS

- What makes a good concept?

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

- Have we meant to include std::string - but that **+** concatenates, rather than adding
- **Addable** is not a suitable concept for general use, it does not represent a fundamental user-level concept
- If **Addable**, why not **Subtractable**? Instead, define something like **Number**

```
template<typename T>
concept bool Number = requires(T a,T b) {
        { a+b } -> T;
        { a-b } -> T;
        { a*b } -> T;
         { a/b } -> T;
        { -a } -> T;
        … };
```

DESIGNING WITH CONCEPTS

- A good useful concept supports a fundamental concept by supplying the set of properties – such as operations and member types – that a domain expert would expect.

- The mistake made for **Drawable** and **Addable** was to use the language features naively without regard to design principles.

DESIGNING WITH CONCEPTS - SEMANTICS

- How do we find such a useful set of properties to design a useful concept? Most application areas already have them. Examples are
  - C++ built-in type concepts: arithmetic, integral, and floating
  - STL concepts like iterators and containers
  - Mathematical concepts like monoid, group, ring, and field
  - Graph concepts like edges and vertices; graph, DAG, etc.
- Alex Stepanov once said "concepts are all about semantics"
- We have found that trying to specify semantics for a new concept is an invaluable help in designing useful concepts and stable interfaces
- Often asking "can we state an axiom?" leads to significant improvements of a draft concept

DESIGNING WITH CONCEPTS - SEMANTICS

- The first step to design a good concept is to consider what is a complete (necessary and sufficient) set of properties (operations, types, etc.) to match the domain concept, taking into account the semantics of that domain concept.

DESIGNING WITH CONCEPTS - IDEALS FOR CONCEPT DESIGN

- What makes one concept better than another?
- Fundamentally, concepts are there to allow us to state fairly abstract ideas in code, so that we can write better generic code
- One way to look at this is that concepts help us to make algorithms and types "plug compatible"
  - we want to write algorithms that can be used for a wide variety of types, and
  - we want to define types that can be used with a wide variety of algorithms.
- For example: we want to define vector<number> types that can be used for our numeric algorithms and algorithms that can be used with all our containers.

DESIGNING WITH CONCEPTS - IDEALS FOR CONCEPT DESIGN

- Consider a simplified version of **std::accumulate**

```
template<Forward_iterator Iter, typename Val>
        requires Incrementable<Val, Iter::value_type>
Val sum(Iter first, Iter last, Val acc) {
        while (first!=last) {
                acc += *first;
                ++first;
        }
        return acc;
};
```

- **Incrementable** would be a concept that simply required the **+=** operator to be present.

DESIGNING WITH CONCEPTS - IDEALS FOR CONCEPT DESIGN

- Minimize the work needed by someone designing types that might be used as **sum** arguments and maximize the usefulness of the **sum** algorithm. However
  - We "forgot" to say that **Val** had to be copyable and/or movable
  - We cannot use this **sum** for a **Val** that provides **+** and **=**, but no **+=**
  - We cannot modify this **sum** to use **+** and **=** instead of **+=** without changing the requirements (part of the functions interface)

- This is not very "plug compatible" and rather ad hoc. That kind of design leads to programs where
  - Every algorithm has its own requirements (a variety that we cannot easily remember).
  - Every type must be designed to match an unspecified and changing set of requirements.
  - When we improve the implementation of an algorithm, we must change its requirements (part of its interface), potentially breaking code.

DESIGNING WITH CONCEPTS - IDEALS FOR CONCEPT DESIGN

- In this direction lies chaos. Thus, the ideal is not "minimal requirements" but "requirements expressed in terms of fundamental and complete concepts."
- This puts a burden on the designers of types (to match concepts), but that leads to better types and to more flexible code.

```
template<Forward_iterator Iter, Number<Iter::value_type> Val>

Val sum(Iter first, Iter last, Val acc) {
        while (first!=last) {

                acc += *first;

                ++first;

        }

        return acc;

}
```

DESIGNING WITH CONCEPTS - IDEALS FOR CONCEPT DESIGN

- Note that by requiring a **Number** we gained flexibility. We also "lost" the accidental ability to use **sum** to concatenate **std::string**s and to sum a **vector<int>** into a **char***
- **Good!** Strings and pointers are not numbers. If we really wanted that functionality, we could easily write it deliberately

- To design good concepts and to use concepts well, we must remember that an implementation isn't a specification – someday, someone is likely to want to improve the implementation and ideally that is done without affecting the interface. Often, we cannot change an interface because doing so would break user code. To write maintainable and widely usable code we aim for semantic coherence, rather than minimalism for each concept and algorithm in isolation.

DESIGNING WITH CONCEPTS - CONSTRAINTS

- So should we use only complete concepts?
- incomplete concepts can be very useful, especially during earlier stages of development in a new application domain.
- For example, the **Number** concept above is incomplete because I "forgot" to require **Number**s to be copyable and/or movable.
- Even as it is, the use of **Number** saves us from many errors. It catches all errors related to missing arithmetic operations.
- However, we still get an error: we just get one of the traditional late and messy error messages we have been used to for decades. The system is still type safe.

- "incomplete concepts" as an important aid to development and to gradual introduction of concepts – Stroustrup
- Sometimes, we call such overly simple or incomplete concepts "constraints" to distinguish them from the "real concepts"

DESIGNING WITH CONCEPTS - MATCHING TYPES TO CONCEPTS

- How can a writer of a new type be sure it matches a concept?

```
class My_number { /* … */ };
static_assert(Number<My_number>);
static_assert(Group<My_number>);
static_assert(Someone_elses_number<My_number>);
class My_container { /* … */ };
static_assert(Random_access_iterator<My_container::iterator>);
```

- After all, concepts are simply predicates, so we can test them. They are *compile-time* predicates, so we can test them at compile time.
- A user might like to add such tests to catch mismatches early and in specific places in the code.

CONCEPT OVERLOADING

- Generic programming relies on using the same name for operations that can be use equivalently for different types. Thus, overloading is essential.
- Where we cannot overload, we need to use workarounds (e.g., traits, **enable_if**, or helper functions).
- Concepts allows us to select among functions based on properties of the arguments given. Consider a simplified version of the standard-library **advance** algorithm

```
template<typename Iter> void advance(Iter p, int n);
```

- But we need a different impl for Forward_iterator and Random_access_iterator such compile-time selection is essential for performance of generic code. Traditionally, we have implemented that using helper functions and tag dispatch

CONCEPT OVERLOADING

- With concepts the solution is simple and obvious

```
template<Forward_iterator Iter>  void advance(Iter p, int n) {  while(n--) ++p;  }

template< Random_access_iterator Iter> void advance(Iter p, int n) {  p+=n; }


void use(vector<string>& vs, list<string>& ls)  {
        auto pvs = find(vs, "foo");

        advance(pvs, 2); // use fast advance


        auto pls = find(ls, "foo");

        advance(pls, 2); // use slow advance

}
```

CONCEPT OVERLOADING

- How does the compiler figure out how to invoke the right **advance?**

- Overload resolution based on concepts is fundamentally simple:
  - If a function matches the requirements of one concept only, call it
  - If a function matches the requirements of no concept, the call is an error
  - If the function matches the requirements of two concepts, see if the requirements of one of those concepts is a subset of the requirements of the other.
    - If so, call the function with the most requirements (the strictest requirements).
    - If not, the call is an error (ambiguous).

CONCEPT THE SHORT-FORM NOTATIONS

- Consider:

```
template<typename Seq> requires Sortable<Seq>
void sort(Seq& s);
```

- We can shorten that to:

```
template<Sortable Seq>
void sort(Seq& s);
```

- However, that still doesn’t get us to the ideal equivalence to “ordinary non-generic”

```
void sort(Sortable& s);
```

CONCEPT THE SHORT-FORM NOTATIONS - AUTO ARGUMENTS

```
void f(auto x); // take argument of any type

concept bool Any = true;   // every type is an Any
void g(Any x);             // take argument of any type

Void ff(auto x, auto y);   //x and y can be of different type

void gg(Any x, Any y);     // x and y must take the same type

void user(vector<string>& vs, list<double>ld)  {
        ff(&vs,&ld);                   // keep a list of containers (of arbitrary types)
        gg(vs.begin(), vs.end());      // OK: two iterators of the same type
        gg(vs.begin(), ld.end());      // error: two iterators to different types
}
```

CONCEPT THE SHORT-FORM NOTATIONS - READABILITY

- expected that because fundamentally concepts enable better interface specification and good interfaces simplify understanding.
- Replacing **auto** with a concept
  - **if (auto x = foobar(z))**
  - **if (InputChannel x = foobar(z))**
- Concepts eliminate unreadable workarounds and complicated boilerplate. (e.g. enable_if)

REFERENCES

- https://en.cppreference.com/w/cpp/language/constraints
- https://accu.org/index.php/journals/2316
- https://gcc.gnu.org/gcc-6/changes.html
- http://www.stroustrup.com/good_concepts.pdf
- http://www.stroustrup.com/popl06.pdf
- C++ Templates book second edition