C++1z: Concepts-Lite

Meeting C++ 2013

slides: http://wiki.hsr.ch/PeterSommerlad/

Prof. Peter Sommerlad
Director IFS Institute for Software
Credo:

- **Work Areas**
  - Refactoring Tools (C++, Scala, ...) for Eclipse
  - Decremental Development (make SW 10% its size!)
  - C++ (ISO C++ committee member)

- **Pattern Books (co-author)**
  - Pattern-oriented Software Architecture Vol. 1
  - Security Patterns

- **Background**
  - Diplom-Informatiker / MSc CS (Univ. Frankfurt/M)
  - Siemens Corporate Research - Munich
  - itopia corporate information technology, Zurich
  - Professor for Software Engineering, HSR Rapperswil, Director Institute for Software

**People create Software**
- communication
- feedback
- courage

**Experience through Practice**
- programming is a trade
- Patterns encapsulate practical experience

**Pragmatic Programming**
- test-driven development
- automated development
- Simplicity: fight complexity
What is a concept?

- The STL defined many "concepts" for template parameters since the 1990s, e.g.,
  - ForwardIterator
  - SequenceContainer
  - UnaryFunction
- The language standard defines concepts as type categories:
  - integer types, arithmetic types, ...
Template Argument's Concepts

- Requirements on a template's typename parameter are given implicitly by its usage.
- Such implicit requirements are called the "concept" of the template parameter
- fulfillment is only checked when template is used, i.e., min() is called with a type

```cpp
template <typename T>
T const& min(T const& a, T const& b){
    return (a < b)? a : b ;
}
```
Concept for min()'s T

- operator<(T const&, T const&) must be defined
- or operator<(T, T)
- and it must return a value convertible to bool
- void operator<(T, T) would be impossible
What is the concept for max()'s T?

- operator<(T const&, T const&) must be defined
- or operator<(T,T)
- and it must return a value convertible to bool
- What else?

```cpp
template <typename T>
T max(T a, T b){
    return a < b ? b : a;
}
```
Concept LessThanComparable

```cpp
template <typename LTC>
concept bool LessThanComparable() {
   return requires(LTC a, LTC b) {
      { a < b } -> bool;
   };
}

template <typename T>
requires LessThanComparable<T>()
T const &max(T const &a, T const &b) {
   return (a < b)?b:a;
}

template <LessThanComparable T>
T const &min(T const &a, T const &b) {
   return (a<b)?a:b;
}
```
What does a concept provide?

• A concept for a type denotes the allowed operations on values of that type or usages of that type in specific contexts

• Those are rules in the standard to be enforced by compilers and the library

• If not followed, often "undefined behavior"

• However, C++ language support for concepts of the library is only available through clumsy workarounds with traits, i.e.,

  • `iterator_traits<iterator_type>::iterator_category`
Why do we want concepts?

• In contrast to regular function parameters that have a type
  • `void foo(int i)`

• template typename parameters are unconstrained
  • `template<typename T> void foo(T)`

• calling foo with the wrong type argument can result in compile errors, depending on foo's implementation
Benefits of unconstrained templates

• Better reuse
  • can use templates in unforeseen ways
• fostered by partial instantiation
  • only what is actually used is instantiated
• enabling "conditional compilation" through SFINAE
  • e.g., `std::enable_if<cond,type>` return types or parameter types for function overloads to select implementations in a generic way
Drawbacks of unconstrained templates

• potentially late and unintelligible error messages
  • not where the mistake is made by the user, but deeply nested in a library

• enable_if is hard to teach and apply
  • and might get overused
    • for functions overloading might be better

• Debugging compilation problems of template programs that make use of SFINAE and partial specialization is tricky (minimal mistake can lead to enormous error message)

• Many more...
Dreams of template library authors

• Specify that a template argument must follow some rules explicitly
  • not implicitly by its usage in the template
• Early error messages when misapplied
  • not in deeply nested template instantiations
• Have a type system for template (template) parameters
  • constrain usages, guide template users
Why concepts are hard?

Getting the granularity right is problematic:

- too coarse: templates become less reusable
- too stringent on individual template argument, i.e., when only partial functionality of a class is actually needed
- too fine: concept explosion leads to chaos
- combining the right concepts and knowing them is hard (-> let to abandonment of concepts for C++0x)
New approach: concepts-lite

- Stage introduction of concepts into C++:
  
  1. introduce syntax to establish syntactic constraints on template arguments (concepts-lite)
  
  2. adapt or rewrite standard library to gain experience
  
  3. establish "full-fledged" concepts with semantic constraints and template definition checking as language feature and adapt library
How will it look like?

- Provide implicit constraints on template typename or template parameters through using a concept name instead of "typename"

  - `template<Integral INT> void foo(INT i)`
  - `template<SameType ...Args> auto make_vector_from_values(Args... values)`
  - `template<Sortable CONTAINER> void sort(CONTAINER &c)`
Details unfolded

- Specifying a concept
- Defining concepts using constraints
- Constraining code through requirements

- Deeper details: ordering of requirements, overloading viability, specialization order, member requirements
Specifying a concept

- A concept looks like a constexpr bool function
  - constexpr function syntax was first proposed, currently concept is a separate thing
  - still some discussion on syntax is going on
- requires(){}; -> specifies constraints
- all elements in list must be fulfilled
Specifying a requirement

• requires clauses specify requirements on template parameters
• works for function templates, template classes
• member functions of template classes
• special syntax before body

```
template < typename ADD, typename ADD2=ADD>
requires Addable<ADD, ADD2>()
ADD add(ADD a, ADD2 b)
{
    return a+b;
}
```
requirement for member of template

template <typename T>
struct Vec : std::vector<T> {
using std::vector<T>::vector;
T sum() const requires Addable<T>() {
    T res{};
    for(auto x:*this) { res = res + x; }
    return res;
}
"current syntax requires () here to refer to concept"

• member functions of template classes
  • special syntax just before body
  • after a trailing return type (->T)
template <typename ADD>
concept bool Addable1()
{
    return requires(ADD a, ADD b){
        a+b;
    };
}
template <Addable1 ADD>
ADD add1(ADD a, ADD b)
{
    return a+b;
}

- You can use a concept's name instead of typename when defining templates
- For functions/lambdas you can even use it as pseudo parameter type (not implemented yet)
requires clause: Allowed checks

- concept checks ("call concept function")
- requires expressions
- "atomic propositions"
  - any other constant expression -> bool
  - e.g. from type_traits
  - constexpr function calls
- conjunction && and disjunction ||
requires clause: combined concepts

- clause is a constant expression evaluating to bool, but only some operators are allowed:
  - Conjunction: `&&` - both satisfied
    - no overloaded operator `&&()`
  - Disjunction: `||` - at least one satisfied
    - no overloaded operator `||()`
  - NO logical negation allowed
    - would make ordering impossible
Overloading with concepts

• Like with SFINAE you can select overloads with concept requirements

• a viable function without satisfied concept is not chosen

• Unlike SFINAE constrained templates are ordered by how "specialized" a constraint is

• "more specialized" match is chosen
# Example overload selection

```cpp
#include <iostream>

template <typename T>
bool concept is_red() {
    return requires (T a) {
        a.red();
    };
}

template <typename T>
bool concept is_green() {
    return requires (T a) {
        ++a;
    };
}

// template with requires for green

template <typename T>
requires is_green<T>()
auto doit(T t) {
    std::cout << "doit with green\n";
    return ++t;
}

// template with requires for red

template <typename T>
requires is_red<T>()
auto doit(T t) {
    std::cout << "doit with red\n";
    return t;
}

struct Green { void operator++() {} };
struct Red { void red() const {} };
struct Blue {};

int main() {
    doit(5); // green
    doit(Red{}); // red
    doit(Green{}); // green
    doit(Blue{}); // without green
}
```
Template Specialization: concepts

- Class template specializations are ordered towards "more specialized"
- But only if constraints are satisfied
- Non-satisfied constraint hides specialization, like with SFINAE
- Logic determined through ordering of pseudo-function template definitions using constraints

```
template<typename T> class S { };
template<Integer T> class S<T> { }; // #1 -> A
template<Unsigned_integer T> class S<T> { }; // #2->B-> more special
// compiler internal rewrite to determine ordering:
template<Integer T> void f(S<T>); // A
template<Unsigned_integer T> void f(S<T>); // B more specialized
```
requires expressions: what?

• requires expression can provide a list of syntactical requirements or nested requires expressions.

• simple: expression requiring syntax:
  • a++;

• compound: { expression } -> result_type
  • { *it } -> Value_type<ITER> const &

• type: type existence guarantee
  • typename Value_type<ITER>;

• nested: requires( params ) { ... }
#include <iterator>
#include <algorithm>

template <typename ITER>
using Value_type=typename
std::iterator_traits<ITER>::value_type;

template <typename ITER>
using IterCategory=typename
std::iterator_traits<ITER>::iterator_category;

template <typename ITER>
concept bool Iterator() {
    return requires(ITER it, ITER end){
        IterCategory<ITER>();
        { *it++ } -> Value_type<ITER>;
        { it != end } -> bool;
    };
}

template <typename OITER>
concept bool OutputIterator() {
    return requires(OITER out) {
        {IterCategory<OITER>()} -> std::output_iterator_tag;
        *out = Value_type<OITER>();
        ++out;
    };
}

template <typename ITER>
concept bool ForwardIterator() {
    return requires(ITER it, ITER end) {
        { std::iterator_traits<ITER>::iterator_category() } -> std::forward_iterator_tag;
    };
}

template <ForwardIterator ITER>
requires OutputIterator<ITER>()
concept bool WritableRandomAccessIterator() {
    return requires(ITER it) {
        { it[0] } -> Value_type<ITER>;
    };
}

template <WritableRandomAccessIterator ITER>
void mysort(ITER b, ITER e) {
    std::sort(b,e); 
}

#include "iterator.h"
#include <vector>
#include <list>

int main(){
    std::vector<int> v{3,1,4,1,5,9,2,6};
    mysort(v.begin(),v.end());
    std::list<int> l{3,1,4,1,5,9,2,6};
    // mysort(l.begin(),l.end());
}
More experiments...

• What do you want to know?
Round up

• Concepts will come for C++
  • even though there are "bike-shed" discussions on syntax
• Early implementation not yet complete
  • stay tuned -> http://concepts.axiomatics.org/
• "Conceptifying" the standard library will come as a second step and may result in a new STL with ranges etc.
Questions?


http://linticator.com http://includator.com

http://sconsolidator.com

peter.sommerlad@hsr.ch http://ifs.hsr.ch

Have Fun with C++
Try TDD, Mockator
and Refactoring!