roi.barkan@gmail.com

Concepts, Type Traits and Specialization

Italian C++ Conference, June 2021

roi.barkan@gmail.com

Ciao, sono Roi

- Roi Barkan - רועי ברקן
- I live in Tel Aviv, Israel
- C++ developer since 2000
- VP Technologies @ Istra Research
  - Finance, Low Latency, in Israel
  - careers@istraresearch.com

roi.barkan@gmail.com

C++20 - Concepts are Here

https://youtu.be/Bcu9ymklfM8
Early 2014

roi.barkan@gmail.com

C++20 - Concepts are Here

https://youtu.be/0avh39CI_ls

Early 2013

roi.barkan@gmail.com

Templates and Overload Resolution

- Templates let us write the same algorithm for multiple types (or parameters)

```
template <class T> constexpr const T &min(const T &a, const T &b)
{
  return (b < a) ? b : a;
}
```

- Metaprogramming lets us implement different overloaded algorithms (and more)

```
template< class T >
constexpr void swap( T& a, T& b ) noexcept;
template< class T2, std::size_t N >
constexpr void swap( T2 (&a)[N], T2 (&b)[N]) noexcept;
```

roi.barkan@gmail.com

Templates and Overload Resolution

- Sometimes multiple overloads are legitimate, but one is preferable

```
template <typename _Tp>
inline typename __enable_if<__is_byte<_Tp>::__value,
void>::__type
__fill_a(_Tp *__first, _Tp *__last, const _Tp &__c) {
  const _Tp __tmp = __c;
  if (const size_t __len = __last - __first)
    memset(__first, static_cast<unsigned char>(__tmp), __len);
}
```

roi.barkan@gmail.com

What is this Talk About ?

- Putting constraints on our templates
- C++20 Concepts - and their alternatives/ancestors
- Many opinions, some facts.
- Tips and ideas - when should we use various mechanisms
- Suggestions for changes to the language
  - Opinions - not facts
- Snippets from the STL
- Clips from YouTube

roi.barkan@gmail.com

What are Concepts

https://youtu.be/0avh39CI_ls
Early 2013

roi.barkan@gmail.com

What are Concepts Good For

https://youtu.be/0avh39CI_ls
Early 2013

roi.barkan@gmail.com

What do they Mean

Concepts library (C++20)

The concepts library provides definitions of fundamental library concepts that can be used to perform compile-time validation of template arguments and perform function dispatch based on properties of types. These concepts provide a foundation for equational reasoning in programs.

Most concepts in the standard library impose both syntactic and semantic requirements. It is said that a standard concept is *satisfied* if its syntactic requirements are met, and is *modeled* if it is satisfied and its semantic requirements (if any) are also met.

In general, only the syntactic requirements can be checked by the compiler. If the validity or meaning of a program depends whether a sequenced of template arguments models a concept, and the concept is satisfied but not modeled, or if a semantic requirement is not met at the point of use, the program is ill-formed, no diagnostic required.

Concepts Seem Simple

- A Bunch of Boolean Expressions
- Take the Overload that Meets the Largest Number of Predicates
- Syntactic and Semantic

roi.barkan@gmail.com

A Bunch of Boolean Expressions

- ```
  template < class T >
  concept integral = std::is_integral_v<T>;
  ```
- ```
  template < class T >
  concept signed_integral = std::integral<T> &&
  std::is_signed_v<T>;
  ```
- ```
  template< class T >
  concept swappable = requires(T& a, T& b) {
      ranges::swap(a, b);
    };
  ```
  ```

roi.barkan@gmail.com

Boolean Expressions aren't New

- Class templates (with '::value' member)

```
template <bool B>
using bool_constant = integral_constant<bool, B>;
typedef bool_constant<true> true_type;
```

- Variable templates

```
template<class R>
inline constexpr bool enable_borrowed_range = false;
```

- Function templates

```
template<class S, class M>
constexpr bool is_pointer_interconvertible_with_class (M S::* mp) noexcept;
```

roi.barkan@gmail.com

Full Expressiveness is Possible

- Boolean operators are allowed

```
template< class T >
struct is_scalar : integral_constant<bool,
    is_arithmetic<T>::value || is_enum<T>::value           ||
    is_pointer<T>::value    || is_member_pointer<T>::value ||
    is_null_pointer<T>::value> {};
```

- SFINAE, void_t, the detection idiom instead of `requires` expressions

```
template <typename, typename = void>
struct has_meow : std::false_type { };
template <typename T>
struct has_meow<T, void_t<decltype(std::declval<T>().meow())>>
    : std::true_type { };
```

roi.barkan@gmail.com

Predicates on Types - More Approaches

- Specialization (common for STL type traits)

```
template<class T> struct is_const          : std::false_type {};
template<class T> struct is_const<const T> : std::true_type {};
```

- Opt-In/Out specialization

```
template<class R>
inline constexpr bool enable_borrowed_range = false;
```

- Concepts CANNOT be specialized
    - But they can be composed of booleans that specialize
- Traits:

```
namespace std {
    template<> struct numeric_limits<Temperature> {
        static constexpr bool has_infinity = false;
        // implement other traits
    };}
```

roi.barkan@gmail.com

Concepts Seem Simple

- A Bunch of Boolean Expressions
- Take the Overload that Meets the Largest Number of Predicates
- Syntactic and Semantic

Controlling Library-Application Interaction

- When Applications use Libraries there’s risk of errors due incorrect expectations.
    - Different components developed by different people on separate occasions.
- Overload-resolution is a way to try and verify that expectations are matched.
- This can be an ‘on/off’ constraint to (dis)allow certain interaction, or more advanced mechanism to choose/tailor specifics of an interaction
- Some resolution mechanisms can be easily bypassed, while others are less negotiable.

Overload Resolution with Concepts

- requires clause
    - Two more syntax alternatives for good measure
- The most specialized version wins (see standard for details)
- SFINAE friendly
- Clear error messages
- Faster compilation speed
- Library defines requirements - Application must conform.

roi.barkan@gmail.com

Alternatives to `requires` Clause

https://youtu.be/ImLFlLjSveM
September 2020

roi.barkan@gmail.com

More Library Guided Approaches

- enable_if
  - Library defines requirements.
  - Compiler doesn't rank - error on multiple matches.
- "Partial Specialization" - function more specialized than another

```
template< class Ptr >
constexpr auto to_address(const Ptr& p) noexcept;
template< class T >
constexpr T* to_address(T* p) noexcept;
```

  - Library defines requirements. Compiler ranks most-specialized.

roi.barkan@gmail.com

Tag Dispatch - The STL Classic

- ```
  template <class _InputIter>
  inline void advance(_InputIter &__i,
          typename iterator_traits<_InputIter>::difference_type __n)
  {
    __advance(__i, __n, typename
                iterator_traits<_InputIter>::iterator_category());
  }
  ```
- Iterators opt-in to the their category/concept
- In STL this dispatch is hidden (implementation detail)
  - Technically libraries could allow call-site override
  ```

roi.barkan@gmail.com

Application Guided Approaches

- Policy-Based Design

```
template <class ForwardIt, class Compare = std::less<>>
constexpr ForwardIt max_element(ForwardIt first, ForwardIt last,
                                Compare comp = Compare{});
```

  - This way callers can override at the call-site
- Customization Points (and CPOs, tag_invoke)
  - Algorithms that can be specialized
  - Examples: std::swap, ranges::ssize, ranges::empty, …
  - CPOs are objects with operator() that deal with overload resolution intricacies
  - Niebloids - similar mechanism for ADL avoidance
- Behavioral Properties (P1393, C++23 executors?, Hollman & Niebler)

```
std::require(executor, execution::blocking.always);
```

  - Library defines properties and Application can use them.

roi.barkan@gmail.com

Overload Resolution / Customizations

	On/Off	Choose from Few	User Code	Simplicity
requires	Library	Library	No	Yes
enable_if	Library	Library	No	No
“Specialization”	Library	Library	No	Yes
Tag Dispatch	Application	Application	No	Medium
Policy Based Design	N/A	No	Caller	Yes*
CPOs	Application	No	Application	Medium
std::require	No	Caller	No	Yes

roi.barkan@gmail.com

Advanced Overload Resolution Schemes

	On/Off	Choose from Few	User Code	How?
requires	Library	Library Application	No	Warrants
enable_if	Library	Library Application	No	Warrants
“Specialization”	Library	Library	No	
Tag Dispatch	Application	Application Caller	No Application	Expose/Add
Policy Based Design	N/A	No Caller	Caller	Policy Tags
CPOs	Application	No (runtime)	Application	
std::require	No	Caller	No	

roi.barkan@gmail.com

Concepts Seem Simple

- A Bunch of Boolean Expressions
- Take the Overload that Meets the Largest Number of Predicates
- Syntactic and Semantic

roi.barkan@gmail.com

Semantics are Tricky

- `std::is_trivially_copyable_v<std::pair<int,int>>`
- Complexity of `std::list::size()`

Complexity

Constant or linear.	(until C++11)
Constant.	(since C++11)

3 years ago

+ Standard - Future

...

The assignment operators ne
elements. Both pair and tuple
conditionally trivial when there
so is an ABI break.

--

```
template< class T >
concept sized_range = ranges::range<T> &&
  requires(T& t) {
    ranges::size(t);
  };
```

roi.barkan@gmail.com

Concepts with Escape Hatches (Warrants)

```
template< class T >
concept sized_range = ranges::range<T> &&
  requires(T& t) {
    ranges::size(t);
  };                                                    (1)

template<class>
inline constexpr bool disable_sized_range = false;      (2)
```

```
template<class R>
concept borrowed_range =
    ranges::range<R> &&
    (std::is_lvalue_reference_v<R> || ranges::enable_borrowed_range<std::remove_cvref_t<R>>);   (1)
```

Defined in header <ranges>
Defined in header <span>
Defined in header <string_view>

```
template<class R>
inline constexpr bool enable_borrowed_range = false;    (2)
```

roi.barkan@gmail.com

cheaply_copiable_t

https://youtu.be/6lurOCdaj0Y
September 2020

roi.barkan@gmail.com

Special Concept Cases

std::equivalence_relation

Defined in header <concepts>

```
template < class R, class T, class U >
concept equivalence_relation = std::relation<R, T, U>;
```

(since C++20)

The concept `equivalence_relation<R, T, U>` specifies that the `relation` R imposes an equivalence relation on its arguments.

Semantic requirements

A relation `r` is an equivalence relation if

- it is reflexive: for all x, `r(x, x)` is true;
- it is symmetric: for all a and b, `r(a, b)` is true if and only if `r(b, a)` is true;
- it is transitive: `r(a, b) && r(b, c)` implies `r(a, c)`.

Notes

The distinction between `relation` and `equivalence_relation` is purely semantic.

roi.barkan@gmail.com

Semantic Sugar - Attaching Semantics

```
template <typename T> concept critical_code = /* ... */;


template <critical_code Operation>
void run_with_priviliges(const std::string &input, Operation operation) {
  operation(input);
}

int main() {
  run_with_priviliges("hello world", mark_critical([](const auto &msg) {
                                      std::cout << msg << std::endl;
                                    }));
  return 0;
}
```

roi.barkan@gmail.com

Semantic Sugar

```
template <typename T> constexpr bool
critical_code_v() { return false; };

struct critical_code_tag {};
template <typename T> requires
std::is_base_of_v<critical_code_tag, T>
constexpr bool critical_code_v() {
  return true;
};
```

```
template <typename T> concept
critical_code = critical_code_v<T>();

template <typename T>
struct mark_critical : T,
critical_code_tag {
  using T::operator();
};
// deduction guide
template <typename T> mark_critical(T)
-> mark_critical<T>;
```

https://godbolt.org/z/4q7fxn

Points to Take Home

- Concepts are great
- `requires` doesn't require `concept`s
- Library writers - give your users power
  - Build escape hatches / warrants
  - Consider call-site customizations
- C++ Standard
  - Consider concept specialization
  - Consider type-trait specialization

roi.barkan@gmail.com

Thank You Questions are Welcome