BUILDING BLOCKS

Generic function and overloading sets. Basic concepts.

K. Vladimirov, Syntacore, 2025
mail-to: konstantin.vladimirov@gmail.com

Raising a number to a power

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov
- Let's start with the first one.
- `unsigned nth_power(unsigned x, unsigned n); // returns` x^n
- How to write the body of this function?

Getting the algorithm right

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
unsigned nth_power(unsigned x, unsigned n) {
  unsigned acc = 1;
  if ((x < 2) || (n == 1)) return x;
  while (n > 0) {
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    x *= x; n /= 2;
  }
  return acc;
}
```


https://godbolt.org/z/eG7855scf

Figuring out possible generalizations

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
unsigned nth_power(unsigned x, unsigned n) {
  unsigned acc = 1;
  if ((x < 2) || (n == 1)) return x;
  while (n > 0) {
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    x *= x; n /= 2;
  }
  return acc;
}
```

Naive generalization

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
template <typename T> T nth_power(T x, unsigned n) {
  T acc = 1;
  if ((x < 2) || (n == 1)) return x;
  while (n > 0) {
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    x *= x; n /= 2;
  }
  return acc;
}
```

Naive generalization issues

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
template <typename T> T nth_power(T x, unsigned n) {
  T acc = 1;
  if ((x < 2) || (n == 1)) return x;
  while (n > 0) {
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    x *= x; n /= 2;
  }
  return acc;
}
```


https://godbolt.org/z/rx8aqfsqM

Less naive generalization

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
template <typename T> T nth_power(T x, unsigned n) {
  T acc = id<T>();
  if ((x == acc) || (n == 1)) return x;
  while (n > 0) {
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    x *= x; n /= 2;
  }
  return acc;
}
```

Introducing traits

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
template <typename T, typename Trait = default_id_trait<T>>
T nth_power(T x, unsigned n) {
  T acc = Trait::id();
  if ((x == acc) || (n == 1)) return x;
  while (n > 0) {
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    x *= x; n /= 2;
  }
  return acc;
}
```


https://godbolt.org/z/d8xM6Exj8

Separating the clean part

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
template <typename T> T nth_power(T x, T acc, unsigned n) {
  while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { x *= x; n /= 2; }
  return acc;
}

unsigned nth_power(unsigned x, unsigned n) {
  if (x < 2u || n == 1u) return x;
  return nth_power<unsigned>(x, 1u, n);
}
```


https://godbolt.org/z/soPExTrqh

Building blocks

- **The building block of generic programming is an overload set.**

```
template <typename T> T nth_power(T x, T acc, unsigned n);
unsigned nth_power(unsigned x, unsigned n);
double nth_power(double x, unsigned n);
```

- We will encounter this repeatedly: classes are designed with sets of overloaded constructors, operators, methods, etc.

Back to Strings

- What if we try to write `operator==` для `basic_string`?
- One simple solution

```
template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs,
                const basic_string<CharT, Traits, Alloc>& rhs) {
  return lhs.compare(rhs) == 0;
}
```

- What's wrong with it?

Back to Strings

- What if we try to write `operator==` для `basic_string`?
- One simple solution

```
template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs,
                const basic_string<CharT, Traits, Alloc>& rhs) {
  return lhs.compare(rhs) == 0;
}
```

- It is inefficient. Think about (`"hello" == str`), an extra copy is clearly created here.
- We would prefer to overload it as a regular function.

A Better Comparison Approach

- The adopted solution (including in libstdc++) uses overloads.

```
template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs,
                const basic_string<CharT, Traits, Alloc>& rhs) {
  return lhs.compare(rhs) == 0;
}

template<typename CharT, typename Traits, typename Alloc>
bool operator==(const CharT* lhs, const basic_string<CharT, Traits, Alloc>& rhs) {
  return rhs.compare(lhs) == 0;
}

template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs, const CharT* rhs) {
  return lhs.compare(rhs) == 0;
}
```

Discussion

- Are there any general principles for designing an overload set?

Examples of Good Design

- Different but related types.

```
void Foo(const char* s);
void Foo(std::string s) { Foo(s.c_str()); }
```

- Different number of parameters.

```
auto s1 = twine("Hello", name).str();
auto s2 = twine("Hello", name, " ", surname).str();
```

- Optimizations.

```
void vector<T>::push_back(const T&);
void vector<T>::push_back(T&&);
```

Winter's Rules

- A person should not be forced to mentally perform overload resolution.
- A single comment can describe the entire set.
- **Each element in the overload set does roughly the same thing.**
- Below is an example of very poor design.

```
// For an integer argument n, returns the smallest coprime
number greater than 1. For a string argument (a comma-
separated list of coprimes), deduces and returns the smallest
possible n that could have generated that list
int least_coprime(int n);
int least_coprime(const std::string& x);
```

Overload Set: Transform

- Is this a good overload set?

```
template <class InputIt, class OutputIt, class UnaryOp>
OutputIt transform(InputIt first1, InputIt last1,
                   OutputIt d_first, UnaryOp unary_op);

template <class ExecutionPolicy,
          class ForwardIt1, class ForwardIt2, class UnaryOp>
ForwardIt2 transform(ExecutionPolicy&& policy,
                     ForwardIt1 first1, ForwardIt1 last1,
                     ForwardIt2 d_first, UnaryOp unary_op);

template <class InputIt1, class InputIt2, class OutputIt, class BinaryOp >
OutputIt transform(InputIt1 first1, InputIt1 last1, InputIt2 first2,
                   OutputIt d_first, BinaryOp binary_op);
```

Overload Set: String Constructors

- Is this a good overload set?

```
basic_string();
basic_string(size_type count, CharT ch);
basic_string(const basic_string& other, size_type pos);
basic_string(const basic_string& other, size_type pos, size_type count);
basic_string(basic_string&& other, size_type pos, size_type count);
basic_string(const CharT* s, size_type count);
basic_string(const CharT* s);
template <class InputIt> basic_string(InputIt first, InputIt last);
basic_string(const basic_string& other);
basic_string(basic_string&& other);
basic_string(std::initializer_list<CharT> ilist);
template <class StringViewLike> explicit basic_string(const StringViewLike& t);
template <class StringViewLike> explicit basic_string(const StringViewLike& t,
                                                      size_type pos, size_type count);
```


[basic_string ctors]

We Must Be Nice

- We must somehow encode the requirements for the interface of generic functions.

```
template <typename T, typename U>
bool check_eq(T lhs, U rhs) { return (lhs == rhs); }
```

- The simplest way to document this is with a requires constraint.

```
template <typename T, typename U>
requires is_equality_comparable<T, U>::value
bool check_eq (T lhs, U rhs) { return (lhs == rhs); }
```

- Now, instead of an error inside the function, we simply exclude it from the overload set.

https://godbolt.org/z/Ez3Mna48E

Combining Constraints

- Constraints are easy to combine

```
template <typename Iter>
 requires(is_forward_iterator<Iter>::value &&
          is_totally_ordered<typename Iter::value_type>::value)
Iter my_min_element(Iter first, Iter last) {
```

- Here, both requirements must be met.
- Furthermore, different error messages are shown depending on what went wrong.

note: 'is_forward_iterator::value' evaluated to false

note: 'is_totally_ordered<typename Iter::value_type, void>::value' evaluated to false

https://godbolt.org/z/Tfnbb9109

Overloading by Constraints

- You can overload based on constraints.

```
struct Foo {
  template <typename Int>
    requires std::is_integral<Int>::value
  Foo (Int x) { std::cout << "Creating int-like object\n";  }

  template <typename Float>
    requires std::is_floating_point<Float>::value
  Foo (Float x) { std::cout << "Creating float-like object\n" }
};
```

- If both failed, what to expect?

Improving Diagnostics

- If no overload is suitable, the failed constraints from each are displayed.

```
struct S{};

Foo fs(S{});
```

- The error messages will be clear and informative.

```
note:   constraints not satisfied
note: 'std::is_integral::value' evaluated to false

note:   constraints not satisfied
note: 'std::is_floating_point::value' evaluated to false
```

Complete coverage

- We can use explicit constraints to discriminate between functions.

```
template <typename T> requires (sizeof(T) > 4)
void foo(T x) { do smth with x }

template <typename T> requires (sizeof(T) <= 4)
void foo(T x) { do smth else with x }
```

- The special status of constraints makes them part of overload resolution.

[over.dcl] two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations and equivalent trailing requires-clauses, if any.

https://godbolt.org/z/66neY5sWb

Sometimes This Goes Too Far

- Expressions inside requires don't even require CT evaluation.

```
consteval bool C() { return true; }

template<typename T> struct A {
  int f() requires (C()) { return 1; }

  // this is not a redeclaration
  int f() requires true { return 2; }
};
```

- This means the analysis of requires clauses happens very early.

https://godbolt.org/z/7Esqvb4jr

Limitations of Simple Constraints

- Alas, simple type traits are not ordered by how restrictive they are.

```
template <typename It>
struct is_input_iterator: std::is_base_of<
  std::input_iterator_tag,
  typename std::iterator_traits<It>::iterator_category>{};

template <typename It>
struct is_random_iterator: std::is_base_of<
  std::random_access_iterator_tag,
  typename std::iterator_traits<It>::iterator_category>{};
```

- These are just two different templates. And this leads to problems.

Limitations of Simple Constraints

- Alas, simple type traits are not ordered by how restrictive they are.

```cpp
template <typename Iter>
  requires is_input_iterator<Iter>::value
int my_distance(Iter first, Iter last) {
  int n = 0;
  while (first != last) { ++first; ++n; }
  return n;
}

template <typename Iter>
  requires is_random_iterator<Iter>::value
int my_distance(Iter first, Iter last) { return last - first; }
```

- In practice, this would cause ambiguity for `std::vector::iterator`.

https://godbolt.org/z/shPTPsdjK

Complex Constraints

- Let's return to a simple example.

```
template <typename T, typename U> bool
  requires is_equality_comparable<T, U>::value
check_eq(T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- The same can be written using a requires-expression.

```
template <typename T, typename U> bool
  requires requires(T t, U u) { t == u; }
check_eq(T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- Yes, the requires-requires might look confusing. But recall noexcept-clause and noexcept-expression.

https://godbolt.org/z/q1sevPWMT

Even Better Diagnostics

```
template <typename T, typename U> bool
  requires requires(T t, U u) { t == u; }
check_eq(T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- The expression

```
check_eq(std::string{"1"}, 1);
```

- Yields

```
note: because 't == u' would be invalid
```

- This not only states the constraint name, but also the specific ill-formed expression within it.

https://godbolt.org/z/q1sevPWMT

The Key Difference of Complex Constraints

- Simple constraints are evaluated at compile time.

```
template <typename T> consteval int somepred() { return 14; }

template <typename T>
  requires(somepred<T>() == 42)
bool foo(T&& lhs, U&& rhs);
```

- Complex constraints check **the validity of an expression**.

```
template <typename T>
  requires requires(T t) { somepred<T>() == 42; }
bool bar(T&& lhs, U&& rhs);
```


https://godbolt.org/z/MoEYTjGcz

Syntax of Complex Constraints

- Think of a complex constraint as a compile-time function returning true or false.

```
requires(T t, U u) {
  u + v; // true если u + v синтаксически возможно [simple]
  typename T::inner; // true если T::inner есть [type]
}
```

- Think of each requirement inside it as a conjunct.
- Simple requirements and type requirements are basic variants of complex constraints.
- There are two more: compound and nested.

Concepts: convertible_to

- To define constraint systems, C++20 introduced the special keyword concept.

```
template<class From, class To>
concept convertible_to = std::is_convertible_v<From, To> &&
  requires {
    static_cast<To>(std::declval<From>());
  };
```

- Think of a concept as an abbreviation for a requires-expression.
- And of course, many useful concepts are already in your standard library.

convertible_to.html

You can check a concept

- Any concept works as a compile-time predicate.
- But it doesn't need to be **called**. A concept is already a value.

```
struct S {};
static_assert(std::move_constructible<S>); // ok
bool a = std::convertible_to<int, double>; // true
bool b = std::convertible_to<int, S>; // false
```


https://godbolt.org/z/9v6jrWsn8

Compound Requirements

- Compound requirements check type compatibility with expressions.

```
requires requires(T x) { {*x} -> typename T::inner; }
```

- A compound requirement can use concepts.

```
requires requires(T x) {
  {*x} -> std::convertible_to<typename T::inner>; // concept
}
```

- There is also a special noexcept syntax.

```
requires requires(T t) {
  { ++t } noexcept;
}
```

Nested constraints

- Inside the requires-expression it can be repeated. This is nested requirement.

```
requires(T t) {
  requires sizeof(T) == 4; // calculated [nested]
  requires somepred<T>() == 42; // consteval predicate [nested]
  requires noexcept(++t); // noexcept expression [nested]
}
```

- Task: simplify this nested requires-clause.

```
template <typename T> int foo(T)
requires requires(T t) { requires noexcept(++t); } {
  return 42;
}
```


https://godbolt.org/z/159GhKcdj

Constraints on Concepts

- Recursive concepts are not allowed.

```
template<bool b, bool ... bs>
concept AllTrueRec = b &&
  ((sizeof...(bs) == 0) ? true : AllTrueRec<bs...>); // ERROR
```

- Concepts cannot be directly constrained by other predicates.

```
template <typename T>
concept Inner = requires { typename T::inner; };

template <typename T> requires Inner<T>
concept InOuter = requires { typename T::outer; }; // ERROR
```


https://godbolt.org/z/qq951avWW

Constraints on Concepts

- Concepts cannot be directly constrained by other predicates.

```
template <typename T>
concept Inner = requires { typename T::inner; };

template <typename T> requires Inner<T>
concept InOuter = requires { typename T::outer; }; // ERROR
```

- This can be somewhat mitigated by conjunction.

```
template <typename T>
concept InOuter = Inner<T> &&
  requires { typename T::outer; }; // OK
```


https://godbolt.org/z/6e9cqrc6b

Syntax for Using Concepts

- Basic syntax.

```
template <typename T> requires std::integral<T> void foo(T);
```

- Template parameter or local variable.

```
template <std::integral T> void bar(T t);
void buz(std::integral auto t); // still function template
std::integral auto x = buz(1);
```

- You can restrict a class template as well.

```
template <typename D> requires std::is_class_v<D>
class Foo { /* */ };
```


https://godbolt.org/z/WG6EefEcK

Funny abbreviations

- Constraint for multiple arguments.

```
template <typename T>
requires std::same_as<T, int>
struct S {};
```

- Abbreviation syntax

```
template <std::same_as<int> T>
struct S {};
```

Concepts on member functions

- You can constraint member functions.

```
template <typename D> struct Foo {
  bool empty() requires ranges::forward_range<D>;
```

- We will see how is it used in ranges library.

```
template <typename T> struct Foo {
  T val;
  bool empty() requires requires { val.empty(); } {
    return val.empty();
  }
```

- As shown above you can also use class members.

Concept on ctor or class?

- You can constraint only ctors without constraining type.

```
template <typename T> requires requires(T x) { x.foo(); }
struct Foo {
  T t;
  Foo() requires std::default_initializable<T> : t() {}
  Foo(Foo &f) requires std::copyable<T> : t(f.t) {}
};
```

- You can have both: generic constraint on type and specific constraints on ctors.

https://godbolt.org/z/9ed6KMPq4

Concept partial order

```
template <typename T> concept Ord = requires(T a, T b) { a < b; };
template <typename T> concept Inc = requires(T a) { ++a; };
template <typename T> concept Int = std::is_same_v<T, int>;

template <typename T> requires Ord<T> || Inc<T> || Int<T>
int foo(T x) { return 2; }

template <typename T> requires Ord<T>
int foo(T x) { return 22; }

int foo(int x) { return 42; }

double y;

foo(y); // -> ?
```


https://godbolt.org/z/18eoWE4nT

Discussion

- How does partial specialization work?
- How does the compiler understand that `Ord` is more specialized than `(Ord || Void)`?

Conjuncts and Disjuncts

- Every concept consists of atomic constraints joined by logical operations (with the usual short-circuiting rules for them).

```
template<typename T>
concept Strange = (sizeof(T) == 4) ||
  (requires() {{T::value} -> convertible_to<bool>} &&
  T::value == true);

template<typename T> requires Strange<T>
void f(T);

f(1); // ok (lazy rules)
```


https://godbolt.org/z/cW5rYE9q5

Subsumes

"A constraint P subsumes a constraint Q if and only if:
for every disjunctive clause P_i in the disjunctive normal form of P
P_i subsumes every conjunctive clause Q_j in the conjunctive normal form of Q"
[temp.constr.order]

```
template <typename T>
concept Q1 = Q<T> || sizeof(T) == 4; // How do you think?

template <typename T>
concept P = Q<T> && R<T>; // P subsumes Q and R
```


https://godbolt.org/z/MrYPfz5d4

Atomic constraints

- An atomic constraint A subsumes another atomic constraint B if and only if A and B are identical [temp.constr.order]

```
template <typename T> constexpr bool Atomic = true;

template <typename T> concept C = Atomic<T>;
template <typename T> concept D = Atomic<T*> && true;
```

- Here compiler cannot determine between D and C, this is ill-formed
- Of course in the ideal world we would prefer this:

A constraint P subsumes a constraint Q if and only if Q implies P.

https://godbolt.org/z/6cnjs1M5K

Identical not similar

```
template<typename T>
concept Foo = (sizeof(T) > 4) && std::is_integral_v<T>;

template<typename T>
concept Bar = std::is_integral_v<T>;

template <Foo T> int f(T x) { return x + 1; }

template <Bar T> int f(T x) { return x + 1; }

int main() {
  return f(1ull); // FAIL
}
```


https://godbolt.org/z/47948rM5n

Subsuming not automatic

```
template<typename T> concept Foo = requires(T x) { x.foo(); };
template<typename T> concept Bar = requires(T x) { x.bar(); };

template<typename T> concept FooBar = requires(T x) {
  x.foo(); x.bar();
};

template <Foo T> int f(T x) { return x.foo(); }

template <FooBar T> int f(T x) { return x.foo() + 1; }

struct SBar { /* have both foo() and bar() */ }

f(SBar{}); // FAIL
```


https://godbolt.org/z/1n8qrjTfs

Subsuming not automatic

```
template<typename T> concept Foo = requires(T x) { x.foo(); };
template<typename T> concept Bar = requires(T x) { x.bar(); };

template<typename T> concept FooBar = Foo<T> && Bar<T>;


template <Foo T> int f(T x) { return x.foo(); }

template <FooBar T> int f(T x) { return x.foo() + 1; }

struct SBar { /* have both foo() and bar() */ }

f(SBar{}); // OK
```


https://godbolt.org/z/1n8qrjTfs

Subsumes

- Now we can order contraints on subsuming.

```
template <typename I>
concept InputIterator = Iterator<I> &&
 requires { typename iterator_category_t<I>; } &&
 DerivedFrom<iterator_category_t<I>, input_iterator_tag>;

template <typename I>
concept ForwardIterator = InputIterator<I> &&
 Incrementable<I> && Sentinel<I, I> &&
 DerivedFrom<iterator_category_t<I>, forward_iterator_tag>;
```

- And so on, down to the random access iterator.

Now overloading works

```
template <InputIterator Iter>
int my_distance(Iter first, Iter last) {
  int n = 0;
  while (first != last) { ++first; ++n; }
    return n;
}

template <RandomAccessIterator Iter>
  int my_distance(Iter first, Iter last) {
  return last - first;
}
```

- Because `InputIterator` is less general (it is a subcondition of `RandomAccessIterator`), there is no ambiguity here.

https://godbolt.org/z/9bKaj4Yds

Sutton's Counterexample

- Lets suppose we have this implementation of copy

```
template <InputIterator In, OutputIterator<value_type_t<In>> Out>
Out copy(In first, In last, Out out) {
 // direct loop
}

template <ContIterator In, ContIterator Out>
  requires MemCopyable<In, Out>
Out copy(In first, In last, Out out) {
 // memcpy
}
```

- Here subsume relationships are hard and we can run into unexpected issues for some types.

Sutton's Counterexample

- Sutton advises not to rely too heavily on subsumption.

```
template <InputIterator In, OutputIterator<value_type_t<In>> Out>
Out copy(In first, In last, Out out) {
 if constexpr(MemCopyable<In, Out>) {
    // реализация через memcpy
  } else {
    // реализация явным циклом
  }
}
```

- After all, how often do we introduce new iterator categories?

The Concepts We Dreamed Of

- In the early articles about concepts, they were much more interesting.

```
concept EqualityComparable<typename T> {
  requires constraint Equal<T>; // syntactic
  requires axiom Equivalence_relation<Equal<T>, T>; // semantic

  // if x == y then for any Predicate p, p(x) == p(y)
  template <Predicate P> axiom Equality(T x, T y, P p) {
    x == y => p(x) == p(y);
  }

  // inequality is the negation of equality
  axiom Inequality(T x, T y) { (x != y) == !(x == y); }
};
```

Homework assignment

- $[HW3.1][1]$ Design realistic overload set for generic function `nth_power` using constraints. Account for integers, floating point and matrices.
- $[HW3.2][1]$ Try to explain what is going wrong here

```
template <typename... Ts> concept Addable =
  requires(Ts... p) { (... + p); requires sizeof...(Ts) > 1; };

template <Addable... Ts> auto sum_all(Ts... p) {
  return (... + p);
}

sum_all(1, 2); // FAIL
```


https://godbolt.org/z/jKq1GG7hr

Bibliography, part 1

- ISO/IEC, "Information technology – Programming languages – C++", ISO/IEC 14882: 2023
- Bjarne Stroustrup, The C++ Programming Language (4th Edition), 2013
- Bjarne Stroustrup, Gabriel Dos Reis – Concepts – Design choices for template argument checking, 2003
- Alexander A. Stepanov, Paul McJones, Elements of programming, Addison-Wesley, 2009
- Bjarne Stroustrup, Andrew Sutton – Design of Concept Libraries for C++, 2011
- Bjarne Stroustrup, Gabriel Dos Reis, Andrew Sutton – Concepts Lite, 2013
- Alexander A. Stepanov, Daniel E. Rose, From mathematics to generic programming, Addison-Wesley, 2014

Bibliography, part 2

- Titus Winters, Modern C++ Design (2 parts), CppCon, 2018
- Andrew Sutton – Concepts in 60: everything you need to know about concepts, CppCon, 2018
- Arthur O'Dwyer – Concepts as she is spoke, CppCon, 2018
- Matias Pusz – C++ concepts and ranges, C++ meeting, 2018
- Andreas Fertig – C++20 Templates: The next level: Concepts and more, CppCon, 2021
- Nicolai Josuttis – Back to Basics: Concepts in C++, CppCon, 2024