Improve generic APIs with constraints (concepts)

Akhil Mithran

PhD Student

GSI C++ User Group

Frankfurt Institute for Advanced Studies (FIAS), Frankfurt am Main, Germany

Goethe-Universität Frankfurt, Frankfurt am Main, Germany

What are constraints and concepts?

- Constraints?
  constraints specify requirements on template arguments to select the most appropriate function overloads and template specializations.
- Concepts?
  concepts are named set of such constraints/requirements

- Introduced from C++20 onwards, new keywords - requires, concept

Why constraints and concepts?

- Violations of constraints detected at compile time before template instantiation
- –-> leads to easy to follow error messages
- On a subjective note:
  - More safety
  - Better readability of code
  - speed up compilation time

Why constraints and concepts?

- leads to easy to follow error messages

```cpp
#include <array>

template<typename T>
auto getSum(T& items){
    typename T::value_type sum{};
    for (auto& item : items)
        sum+=item;
    return sum;
}

class A{};

int main(){

    A a;

    [[maybe_unused]] auto c = getSum(a);
}
```

```
└> bash compile_cpp.sh error_msg.cpp
Compiling the following:
g++ -std=c++20 -march=native -fstrict-aliasing -Wall -Wextra -pedantic -fsanitize=address,undef
ined   error_msg.cpp
error_msg.cpp: In instantiation of 'auto getSum(T&) [with T = A]':
error_msg.cpp:19:37:   required from here
   19 |     [[maybe_unused]] auto c = getSum(a);
      |                               ~~~~~~^~~
error_msg.cpp:7:28: error: no type named 'value_type' in 'class A'
    7 |     typename T::value_type sum{};
      |                            ^~~
error_msg.cpp:8:5: error: 'begin' was not declared in this scope
    8 |     for (auto& item : items)
      |     ^~~
error_msg.cpp:8:5: note: suggested alternatives:
In file included from /usr/include/c++/14.2.1/array:44,
                 from error_msg.cpp:3:
/usr/include/c++/14.2.1/bits/range_access.h:114:37: note:   'std::begin'
  114 |   template<typename _Tp> const _Tp* begin(const valarray<_Tp>&) noexcept;
      |                                     ^~~~~
In file included from /usr/include/c++/14.2.1/bits/stl_iterator_base_types.h:71,
                 from /usr/include/c++/14.2.1/bits/stl_algobase.h:65,
                 from /usr/include/c++/14.2.1/array:43:
/usr/include/c++/14.2.1/bits/iterator_concepts.h:983:10: note:   'std::ranges::__access::begin'
  983 |     void begin() = delete;
      |          ^~~~~
error_msg.cpp:8:5: error: 'end' was not declared in this scope; did you mean 'std::end'?
    8 |     for (auto& item : items)
      |     ^~~
      |     std::end
/usr/include/c++/14.2.1/bits/range_access.h:116:37: note: 'std::end' declared here
  116 |   template<typename _Tp> const _Tp* end(const valarray<_Tp>&) noexcept;
      |                                     ^~~
```

Why constraints and concepts?

- leads to easy to follow error messages

```
#include <array>

template<typename T>
requires std::is_array<T>::value
auto getSum(T& items){
    typename T::value_type sum{};
    for (auto& item : items)
        sum+=item;
    return sum;
}

class A{};

int main(){

    A a;

    [[maybe_unused]] auto c = getSum(a);
}
```

Why constraints and concepts?

- leads to easy to follow error messages

```cpp
#include <array>

template<typename T>
requires std::is_array<T>::value
auto getSum(T& items){
    typename T::value_type sum{};
    for (auto& item : items)
        sum+=item;
    return sum;
}

class A{};

int main(){

    A a;

    [[maybe_unused]] auto c = getSum(a);
}
```

```
└> bash compile_cpp.sh error_msg.cpp
Compiling the following:
g++ -std=c++20 -march=native -fstrict-aliasing -Wall -Wextra -pedantic -fsanitize=address,undef
ined   error_msg.cpp
error_msg.cpp: In function 'int main()':
error_msg.cpp:20:37: error: no matching function for call to 'getSum(A&)'
   20 |     [[maybe_unused]] auto c = getSum(a);
      |                               ~~~~~~^~~
error_msg.cpp:7:6: note: candidate: 'template<class T>  requires  std::is_array<_Tp>::value aut
o getSum(T&)'
    7 | auto getSum(T& items){
      |      ^~~~~~
error_msg.cpp:7:6: note:   template argument deduction/substitution failed:
error_msg.cpp:7:6: note: constraints not satisfied
error_msg.cpp: In substitution of 'template<class T>  requires  std::is_array<_Tp>::value auto
getSum(T&) [with T = A]':
error_msg.cpp:20:37:   required from here
   20 |     [[maybe_unused]] auto c = getSum(a);
      |                               ~~~~~~^~~
error_msg.cpp:7:6:   required by the constraints of 'template<class T>  requires  std::is_array
<_Tp>::value auto getSum(T&)'
error_msg.cpp:6:28: note: the expression 'std::is_array<_Tp>::value [with T = A]' evaluated to
 'false'
    6 | requires std::is_array<T>::value
      |                            ^~~~~
```

How to define?

- *requires expressions*: yields a prvalue expression of type bool that describes the constraints (different from *requires clauses*).

  requires (*parameter list*) {*sequence of requirements*}

  (*parameter list*: *optional*)

- You can also name this expression (or their combination of multiple expressions) as a concept

- Four main categories as of now:
  - simple requirement
  - type requirement
  - compound requirement
  - nested requirement

Simple Requirement

- Must start parenthesized requires expression
- Cannot start with *requires*
- Only checks if the expression is valid
- Operand is unevaluated

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

template<class T, class U = T>
concept Convertible = requires(T&& t, U&& u)
{
    static_cast<T>(u);
    static_cast<U>(t);
};
```

Type Requirement

- Has the form: *typename identifier*
- Asserts that the type named by identifier is valid
- Only checks if the expression is valid
- Operand is unevaluated

```
template<typename T>
struct S
{};

template<typename T>
concept C = requires
{
    typename T::inner; // required nested member name
    typename S<T>;     // required class template specialization
};
```

Compound Requirement

- Has the form: *{ expression } noexcept -> type-constraint ;*

  optional

- Asserts properties of expression
- Substitution and semantic constraint checking
- If exists takes the *decltype((expression))* as the first argument to the *type-constraint*

```cpp
template<typename T>
concept C = requires(T x)
{
    // the expression x + 1 must be valid
    // AND std::same_as<decltype((x + 1)), int> must be satisfied
    {x + 1} -> std::same_as<int>;
};
```

Additional notes on definition of constraints

- Local parameters have:
  - no linkage
  - no storage
  - no lifetime.

- If substitution of template arguments result in invalid types or expressions, requires expression evaluates to false
- If substitution (if any) and semantic constraint checking succeed, the requires expression evaluates to true

- ill-formed no diagnostic required (IFNDR) If:
  - a substitution failure would occur for every possible template argument
  - a requires expression contains invalid types or expressions in its requirements (we will see later)
  - A local parameter has a default argument.
  - The parameter list terminate with an ellipsis.

```
template<typename T>
concept C1 = requires(T t = 0) // IFNDR
{
    t;
};

template<typename T>
concept C2 = requires(T t, ...) // IFNDR
{
    t;
};
```

How to use them?

- *requires clauses*: keyword used to constraints on template arguments or on a function declaration.
- Must be followed by some constant expression (even *requires* true is valid)
- *Example*

```
template <typename T>
requires CONDITION<T>
void fn(T x) { }

template <typename T>
void fn(T x) requires CONDITION<T> { }

template <CONDITION T>
void fn(T x) { }
```

```
// -*- C++ -*-
#include <type_traits>

// Defining concepts
template<class T>
concept IsNumber =  std::is_integral<T>::value || std::is_floating_point<T>::value;

template <typename T>
requires IsNumber<T>
void fn(T x) { }
```

type_traits header consists of many helpful bits to use with requires expression. Helpful concepts are also defined in the *concepts* header

How to use them?

```
template <typename T>
requires CONDITION<T>
void fn(T x) { }

template <typename T>
void fn(T x) requires CONDITION<T> { }

template <CONDITION T>
void fn(T x) { }
```

- takes one less template argument than its parameter list demands
- contextually deduced type is implicitly used as the first argument of the concept.

```
template<class T, class U>
concept Derived = std::is_base_of<U, T>::value;

template<Derived<Base> T>
void f(T); // T is constrained by Derived<T, Base>
```

Additional notes on using constraints

- You can redeclare them, no problem

```cpp
// Two identical redeclaration of fn
template<Incrementable T>
void fn(T) requires Decrementable<T>;

template<Incrementable T>
void fn(T) requires Decrementable<T>;
```

- IFNDR if:
  - Logically equivalent but syntactically different
  - Logically equivalent but different order of constraints

```cpp
// These first two declarations of f are IFNDR
template<Incrementable T>
void f(T) requires Decrementable<T>;

template<typename T>
    requires Incrementable<T> && Decrementable<T>
void f(T);
```

```cpp
// These two declarations of g are IFNDR.
// 2 different constraints
template<Incrementable T>
void g(T) requires Decrementable<T>;

template<Decrementable T>
void g(T) requires Incrementable<T>;
```

Types of constraints

3 types until C++26 and 4 since C++26:

1. Conjunctions
2. Disjunctions
3. Atomic constraints
4. Fold expanded constraints (added in C++26)

Types of constraints

- Conjunctions
  - formed by using the && operator in the constraint expression
  - satisfied only if both constraints are satisfied
  - evaluated left to right
  - short-circuited

```cpp
// Concept to check if a type has a foo() member function
template <typename T>
concept HasFoo = requires(T t) {
    { t.foo() };  // Checks if foo() exists
};

// Concept to check if a type has a bar() member function
template <typename T>
concept HasBar = requires(T t) {
    { t.bar() };  // Checks if bar() exists
};

// Concept that combines HasFoo and HasBar
template <typename T>
concept HasFooBar = HasFoo<T> && HasBar<T>;
```

Types of constraints

- Disjunctions
  - formed by using the || operator in the constraint expression

```
// -*- C++ -*-
#include <type_traits>

// Defining concepts
template<class T>
concept IsNumber =  std::is_integral<T>::value || std::is_floating_point<T>::value;
```

  - satisfied if either constraint is satisfied.
  - evaluated left to right
  - short-circuited

Atomic constraints

- Most fundamental expression of a contraint
- Formed during constraint normalization
- Consists of two parts:
  - expression *E*
  - parameter mapping
- *E* is never a logical AND or logical OR expression
- The type of *E* after substitution must be exactly bool

```
// -*- C++ -*-
template<typename T>
struct S
{
    constexpr operator bool() const { return true; }
};

template<typename T>
    requires (S<T>{})
void f(T);    // #1

void f(int); // #2

void g()
{
    f(0); // Error: cannot check constraint because
          // of invalid type. Only bool permitted
}
```

Constraint normalization

- process that transforms a constraint expression into a sequence of conjunctions and disjunctions of atomic constraints
- The normal form of an expression (*E*) is the normal form of *E*
- Usually the parameter mapping is the identity mapping.
- However, if another concept, say *C*, is named within the constraint, then we'll have substitution of *C*'s respective template parameters in the parameter mappings of each atomic constraint of *C*
- If any such substitution into the parameter mappings results in an invalid type or expression, the program is IFNDR

Constraint normalization

Examples of valid and invalid constraint normalizations

```
template<class T> constexpr bool always_true = true;

// E: always_true<X>
// parameter mapping: X ↦ T (identity)
template<class T> concept Base = always_true<T>;

// E: always_true<X>
// parameter mapping: X ↦ U::type
template<class U> concept Foo = Base<typename U::type>;

// IFNDR (invalid type in parameter mapping)
// E: always_true<X>
// parameter mapping: X ↦ V*::type
template<class V> concept Bar = Foo<V*>;

// IFNDR (invalid type in parameter mapping)
// E: always_true<X>
// parameter mapping: X ↦ W&*
template<class W> concept Baz = Foo<W&>;
```

Additional notes

- Concepts cannot recursively refer to themselves and cannot be constrained.

- A constraint *P* can *subsume* constraint *Q*

```
template<typename T>
concept V = V<T*>; // error: recursive concept

template<class T>
concept C1 = true;
// Error: C1 T attempts to constrain a concept definition
template<C1 T>
concept Error1 = true;
```

```
template<typename T>
concept Decrementable = requires(T t) { --t; };
// RevIterator1 subsumes Decrementable but not vis-versa
template<typename T>
concept RevIterator1 = Decrementable<T> && requires(T t) { *t; };
// RevIterator2 forms distinct atomic
// constraints from RevIterator1 and
// so do not subsume Decrementable
template<typename T>
concept RevIterator2 = requires(T t) { --t; *t; };
```

- If *P* *subsume* constraint *Q*, then *P* is at least as constrained as *Q*.

- If *Q* is not at least as constrained as *P*, then *P* is more constrained than *Q*.
  This can lead to *Partial ordering* which is used to determine best viable overload among others

Additional use cases

- When initializing variables with type as auto for type deduction, we can constrain auto to constrain the possible initializer values

  ```
  #include <concepts>
  std::integral auto i = 2;
  ```

- *constexpr if statement* (C++17): concepts can be used here because they evaluate to prvalue of type bool

- You can use them with *static_assert* (for tests etc)

```cpp
// -*- C++ -*-
#include <iostream>
#include <vector>

template<typename T>
concept hasSize = requires (T a){ a.size();};

auto f(auto x){
    if constexpr (hasSize<decltype(x)>){
        return x.size();
    } else {
        return 1;
    }
}

int main(){
    std::vector<int> vec{1,2};
    int i=5;
    std::cout<< f(i) << '\n'; // 1
    std::cout<< f(vec) << '\n';// 2
}
```

Alternative to SFINAE

- Assume we want to add to container.
  We wish to create a unified generic interface for this.

- Incorrect version (*redefinition*)

```cpp
#include <iostream>
#include <vector>
#include <set>

template <typename Container, typename T>
bool insertElement(Container& container, const T& value) {
    container.push_back(value);
    return true;
}

template <typename Container, typename T>
bool insertElement(Container& container, const T& value) {
    return container.insert(value).second;
}

int main() {
    std::vector<int> vec;
    std::set<int> s;

    std::cout << "Push to vector: " << insertElement(vec, 10) << "\n";
    std::cout << "Insert into set: " << insertElement(s, 10) << "\n";

    return 0;
}
```

Alternative to SFINAE

- Correct version (*std::enable_if*)

```
#include <iostream>
#include <vector>
#include <set>
#include <type_traits>

// Function to push_back into containers like std::vector
template <typename Container, typename T>
auto insertElement(Container& container, const T& value)
    -> std::enable_if_t<std::is_same_v<decltype(container.push_back(value)), void>, bool> {
    container.push_back(value);
    return true;
}

// Function to insert into containers like std::set
template <typename Container, typename T>
auto insertElement(Container& container, const T& value)
    -> std::enable_if_t<std::is_same_v<decltype(container.insert(value)), std::pair<typename Container::iterator,
bool>>, bool> {
    return container.insert(value).second;
}

int main() {
    std::vector<int> vec;
    std::set<int> s;

    std::cout << "Push to vector: " << insertElement(vec, 10) << "\n";
    std::cout << "Insert into set: " << insertElement(s, 10) << "\n";

    return 0;
}
```

Alternative to SFINAE

- Correct version (*concepts*)

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <concepts>

// Concept to check if a container has push_back
template <typename Container, typename T>
concept SupportsPushBack = requires(Container c, T value) {
    { c.push_back(value) };
};

// Concept to check if a container has insert and returns a std::pair<iterator, bool>
template <typename Container, typename T>
concept SupportsInsert = requires(Container c, T value) {
    { c.insert(value) } -> std::same_as<std::pair<typename Container::iterator, bool>>;
};

// Function to push_back if container supports it
template <SupportsPushBack<T> Container, typename T>
bool insertElement(Container& container, const T& value) {
    container.push_back(value);
    return true;
}

// Function to insert if container supports it
template <SupportsInsert<T> Container, typename T>
bool insertElement(Container& container, const T& value) {
    return container.insert(value).second;
}

int main() {
    std::vector<int> vec;
    std::set<int> s;

    std::cout << "Push to vector: " << insertElement(vec, 10) << "\n";
    std::cout << "Insert into set: " << insertElement(s, 10) << "\n";

    return 0;
}
```

Thank You …

References

- https://en.cppreference.com/w/cpp/language/constraints
- https://www.cppstories.com/2021/concepts-intro/
- https://www.youtube.com/watch?v=jzwqTi7n-rg
  (Back to Basics: Concepts in C++ - Nicolai Josuttis - CppCon 2024)
- https://carbon.now.sh (For creating terminal screenshots)