Concepts

Sonntag, 17. Dezember 2023 11:45

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Constrained vs Unconstrained

By default, we have unconstrained template parameter, where the compiler checks only,
at the point when function/class method is actually invoked, the validity of the function template's body:
whether the statements/expressions inside the function body are valid for a given concrete type - when template instantiation happens.

With constrained template parameter, we introduce additional requirements on a type, that compiler needs to consider
upfront as well (apart of the function body) - in order to reduce the overloading set.
The **reason** behind is to have more specialized/correct code, that better communicates the intention - requirements on a type to

- the clients: callers of the generic code,
- but also to compiler: for producing better error messages

Before Concepts

In order to express the constrains on the template parameters - and therefore impose more rigid rules
for the template parameters substitution, in Pre-Concepts era, we used some
nonintuitive language construct that involves

- ***std::enable_if_t***
- and a ***compile-time expression*** that yields the true on success, or false on failure

Let's demonstrate this on the function template example - sum of arbitrary number of arguments, by introducing
the requirement that all arguments are of the same type

```
// boolean variable template - as compile-time expression
template <typename T, typename...Ts>
static constexpr bool all = std::conjunction_v<std::is_same<T, Ts>...>;

// Trailing constrain on the variadic template parameters, that affects result as well
template <typename T, typename...Ts>
auto add(Ts...ts) -> std::enable_if_t<all<T, Ts...>, T>
{
    return (... + ts);
}
```

If we compare this with unconstrained function template variant - we get less flexible code, that behaves more in line
with our expectations - preventing compiler interventions (like type promotion): as the way to enforce the correctness
of the code.

We can apply *std::enable_if_t* :

- As trailing syntax (above example)
- Direct on the resulting type
- It can be placed in the template header itself.
  In this example, as part of the c-tor overloading

  ```
  // C-tors overloading
  template <typename U = T, typename = std::enable_if_t<not std::is_reference_v<U>>>
  constexpr explicit StrongType(T&& val) noexcept(std::is_nothrow_move_constructible_v<T>)
            :value_(std::move(val))
  {}

  constexpr explicit StrongType(const T& val) noexcept(std::is_nothrow_copy_constructible_v<T>)
            :value_(val)
  {}
  ```
- Eventually, it can constraint the template parameter inside the class definition/function template body as well: as part of the *static_assert*()

I guess, by now you've got the picture - what are the major flaws of this approach:
it's tedious to write, difficult to read and it's not reusable: "*ad hock*" rules.
Well, the latter is not completely true - after all, we can define the ***alias template*** for naming the rule (condition inside the *std::enable_if_t*)
that eventually deduces the type

```
// We can create the alias template on the rule that deduce the type - to make it reusable after all
template <typename T, typename...Ts>
using all_same_t = std::enable_if_t<all<T, Ts...>, T>;
```

Now we can reuse it - in all places where *std::enable_if_t* would be used

```
template <typename T, typename...Ts>
auto add(Ts...ts) -> all_same_t<T, Ts...>
{
    return(... + ts);
}
```

Complete code: https://godbolt.org/z/hGar9e79z

Concepts in rescue

Concepts are **compile-time predicates** that specify constraints on types that have to be satisfied, so that concreate types are eligible to participate in template parameters substitution.

Generally, the Concept definition looks like

```
template <typename...T>
concept name = expression<T...>;
```

It consists of the *template header* that can have a variadic template parameters that participate in the compile-time expression.
The compile-time expression yields boolean value: whether the types related requirements are fulfilled or not.
The important property of the Concept is its ***name*** - naming the concepts has two important benefits.
One is readability - having more expressive code, and the other is reusability of the same.

The Concepts are introduced with C++20, as part of the <*concepts*> header file.
It comes with the already predefined (named) Concepts that we can use - out of box.

Requirements

We can express requirements on types in C++20 with ***requires-clause*** that will evaluate any compile-time expression to true or false.

```
template <typename T>
requires condition<T>
auto f(T&& a) {}
```

where *condition* can be any compile-time expression (unnamed, or named - Concept).

Our StrongType c-tor overloading becomes now

```
// C-tors overloading
template <typename U>
requires(std::convertible_to<U, T> && not std::is_reference_v<U>)
constexpr explicit StrongValue(U&& val) noexcept(std::is_nothrow_move_constructible_v<U>):
value_(std::move(val))
{}

template <typename U>
requires std::convertible_to<U, T>
constexpr explicit StrongValue(const U& val) noexcept(std::is_nothrow_copy_constructible_v<U>):
value_(val)
{}
```

We can also specify the trailing requires-clause.
For example, when we have a member method of a class template, where the method is (full) specialization for the particular type that can appear in template parameter substitution:
In this example, the special handling for the byte-like types

```
template <typename T>
class A {
    public:
        template <typename Byte>
        static constexpr bool is_byte = std::is_same_v<Byte, unsigned char> ||
                                        std::is_same_v<Byte, std::uint8_t> ||
                                        std::is_same_v<Byte, std::byte>;

        bool read_bytes(T* ptr, std::size_t& size) requires is_byte<T>
        {}
};
```

⇒ More on that: https://godbolt.org/z/zzs1Ks4Pq

Another way to express requirements on a type is with ***requires-expression***
The requires-expression has fallowing form

```
template <typename...Ts>
requires(Ts...ts)// optional argument list
{//body
    // one or more expressions that define requirements on the type(s)
    ...
}
```

The compiler will evaluate the entire set of requirements specified inside the `requires-expression body.`
If any of these requirements is not satisfied - the compiler will raise an error

The expression definitions inside the requires body can be:

- **simple requirements**
- **type requirements**
- **nested requirements**
- **compound requirements**

Now, we can define the Concept in terms of the requirements - in **more convenient way**, without needing to implement custom specific type-traits (like "has a method"), that we would use with *std::enable_if_t*

```
// We define our own type-trait to inspect the type on
// the method membership

template <typename T, class V = void>
struct has_a_method : std::false_type {};

template <typename T>
struct has_a_method<T, std::void_t<decltype(std::declval<T&>().foo())>> :std::true_type{};

template<typename T>
static constexpr bool has_a_method_v = has_a_method<T>::value;
```

Let's demonstrate this on a simple example: defining the Concept for containers - as iterable collections

```
namespace details
{
    template <typename Collection>
    concept Container = requires(Collection coll)
    {
        typename Collection::value_type;// type requirement
        requires std::is_same_v<typename Collection::value_type,
std::remove_cvref_t<decltype(*std::declval<Collection&>().begin())>>;// nested requirement
        {coll.begin()} -> std::same_as<typename Collection::iterator>;//compound requirement
        {coll.end()} -> std::same_as<typename Collection::iterator>;//compound requirement
    };
}
```

With *type requirement*, we specified the dependency on certain type definition, in this case underlying type - usually, when we want to use it at client code as an argument for some other class/function template instantiation.
With *nested requirement* we usually define a subset of the logically related - grouped requirements.
Compound requirements are typically used to express behavioral aspect of the type that has to be met - the "*has a method*" relationship.
(more on that later)

At the client side, where we rely on these requirements, we can apply the Concept constraints - which narrow the overloading set of the possible types that can participate in the template parameter substitution

```
template <details::Container T>// Concept as constrained template parameter
std::string containerToString(const T& coll)
{
    std::stringstream ss;
    ss <<"[";
    const auto last =std::prev(coll.cend());
    std::copy(coll.cbegin(), last, std::ostream_iterator<typename T::value_type>(ss, ", "));
    ss << *last <<"]\n";

    return ss.str();
}
```

We could apply the Concept restrictions also directly on the function argument - as part of the abbreviated function template signature.
This is known as constrained placeholder (whereby *auto* alone - is unconstrained placeholder)

```
std::string containerToString(const details::Container auto& coll)
{...}
```

<Example_2>: https://godbolt.org/z/jf71388ea

Static Polymorphism

The Concepts - named one, can be used instead of tedious to write CRTP approach

```
// CRTP pattern
template <typename Derived>
class Base
{

    Base() {}// to prevent somehow the wrong usage: class Derived_1 : public Base<Derived_2> {};
    friend Derived;

    static_assert(std::enable_if_t<std::is_base_of_v<Base, Derived>>::value, "Invalid template argument");

     public:
        void foo()
        {
```

```
            impl().fooImpl();
        }
    private:
        Derived& impl() { return static_cast<Derived&>(*this);}
        const Derived& impl() const { return static_cast<const Derived&>(*this);}
};
```

Basically, instead of define the base class - as a gateway for calling derived class specific implementation behind the common interface of a base class - we can use the Concepts.
Namely, we can specify - using the already mentioned compound requirements, the behavioral aspect - for the non-polymorphic types, that have the polymorphic behavior - implement the very same ***interface.***
One practical example for that would be (re)writing the design patterns, like the Factory Method, or Dependency Injection - translating them from dynamic (runtime) - to the static (compile-time) polymorphism.

⇒ More on that: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/desing%20patterns/Factory%20method.pdf

Summary

The Concepts are beneficial for writing generic code in various way - in compare what we had before:
- Improved syntax:
  - having meaningful name for the Concepts that is informative enough for the code to be well documented (without writing additional comments)
  - that is intuitive enough to understand, and therefore to maintain
  - that produces more readable Compiler errors - easier up to certain extend to debug
  - more convenient way to write - with a plenty already predefined Concepts
- Simplified *Static Polymorphism* modeling, in terms of defining compile-time *Interface*: zero-costs abstraction (no virtuality and run-time binding)

Links

Andreas Fertig: https://andreasfertig.com/books/programming-with-cpp20/
Rainer Grimm: https://leanpub.com/c20