C++20 Templates - The next level

Concepts and more

Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info
@Andreas__Fertig

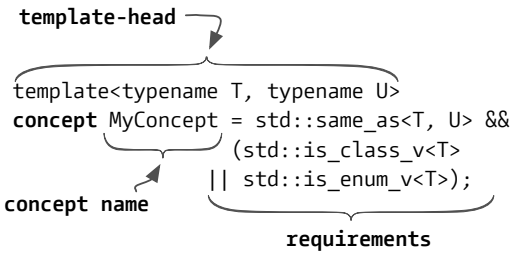
fertig

adjective /ˈfɛrtɪç/

finished
ready
complete
completed

Concepts

- With Concepts we can formulate requirements for a type.
- Comparable to `std::enable_if`.
- Concepts consist of the definition of the concept (**`concept`**) and requirements (**`requires`**):



- A concept is always a template and can be recognized by the new keyword **`concept`**. A concept itself consists of other concepts or requirements. The latter are defined by the keyword **`requires`**.

https://AndreasFertig.Info
post@AndreasFertig.Info

Variadic template parameters of the same type

```
1 const auto x = Add(2,3,4,5);
2 const auto y = Add(2,3);
3 const auto z = Add(2,3, 3.14); // ERROR
```


Variadic template parameters of the same type

C++17 variant: `enable_if` to block instantiation.

```
1  template<typename T, typename... Ts>
2 constexpr inline bool are_same_v = std::conjunction_v<std::is_same<T, Ts>...>;
3
4 template<typename T, typename...>
5 struct first_arg {
6   using type = T;
7 };
8
9 template<typename... Args>
10 using first_arg_t = typename first_arg<Args...>::type;
11
```

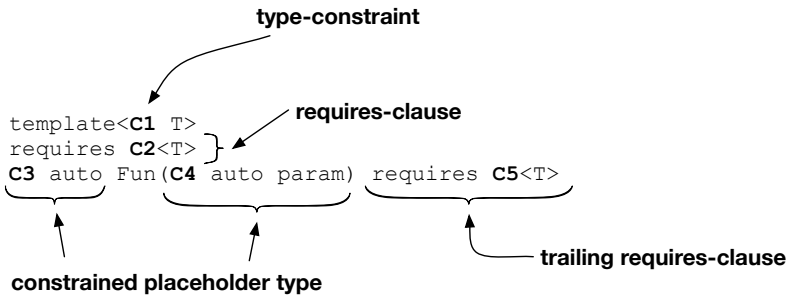
```
1 template<typename... Args>
2 std::enable_if_t<are_same_v<Args...>, first_arg_t<Args...>>
3 Add(Args&&... args) noexcept
4 {
5   return (... + args);
6 }
```

Variadic template parameters of the same type

C++20 variant: `are_same_v` as requirement.

```
1 template<typename... Args>
2 A Requires-clause using are_same_v to ensure all Args are of the same type.
3 requires are_same_v<Args...>
4 auto Add(Args&&... args) noexcept
5 {
6   return (... + args);
7 }
```

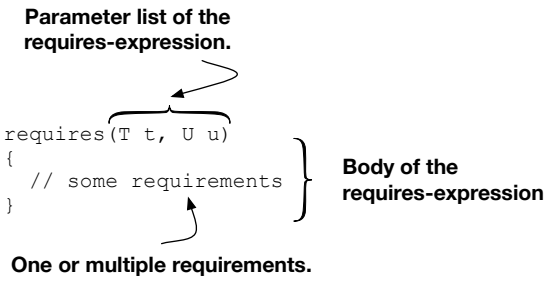

Application areas for Concepts



There is more

- Currently Add only prevents
  - A mixed types.
- The current version of Add leaves a lot unspecified:
  - B Add can nonsensically be called with only one parameter.
  - C The type used in `Args` must support the + operation.
  - D The operation + should be **`noexcept`** since Add itself is **`noexcept`**.
  - E The return type of the operation + should match that of `Args`.

Requires-expression



- Requires-clause (C3, C5) is a boolean expression. A requires-expression is more complicated. Hello, **`noexcept`**.
- We can see a requires-clause like an **`if`** that evaluates a boolean expression and a requires-expression returns such a boolean value.

Requirement kinds

- Simple requirement (SR)
- Nested requirement (NR)
- Compound requirement (CR)
- Type requirement (TR)

Simple requirement

- Checks whether an expression is valid and will compile.

```
1 requires(Args... args)
2 {
3   (... + args);   C SR: args provides +
4 }
```

```
struct NoAdd{};

Add(NoAdd{}, NoAdd{}); C
```

Nested requirement

- Evaluates the boolean return value of an expression.

```
1 requires(Args... args)
2 {
3   (... + args);                    C SR: args provides +
4   requires are_same_v<Args...>;    A NR: All types are the same
5   requires sizeof...(Args) > 1;    B NR: Pack contains at least two elements
6 }
```

```
struct NoAdd{};

Add(NoAdd{}, NoAdd{}); C
Add(2, NoAdd{});  A
Add(2); B
```


Compound requirement

- Checks the return type of an expression. Can be recognized by the curly brackets and the trailing return type.

```
1  requires(Args... args)
2  {
3    (... + args);                    C SR: args provides +
4    requires are_same_v<Args...>;    A NR: All types are the same
5    requires sizeof...(Args) > 1;    B NR: Pack contains at least two elements
6
7    D E CR: ...+args is noexcept and the return type is the same as the first argument type
8    // { (... + args) } noexcept;
9    // { (... + args) } -> same_as<first_arg_t<Args...>>;
10   { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
11 }
```

```
struct NoAdd{};
struct NotNoexcept { NotNoexcept& operator+(const NotNoexcept&); };
struct DifferentReturnType { int& operator+(const DifferentReturnType&) noexcept; };

Add(NoAdd{}, NoAdd{}); C
Add(2, NoAdd{});  A
Add(2); B
Add(NotNoexcept{}, NotNoexcept{}); D
Add(DifferentReturnType{}, DifferentReturnType{}); E
```

Ad hoc constraints: `requires requires`

- The requirements are defined directly after the requires-clause (C2, C5).
- The first **`requires`** introduces the requires-clause, the second **`requires`** begins the requires-expression.
- First sign of a code-smell.

```
 1 template<typename... Args>
 2 requires  requires(Args... args)
 3 {
 4   (... + args);
 5   requires are_same_v<Args...>;
 6   requires sizeof...(Args) > 1;
 7   { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
 8 }
 9 auto Add(Args&&... args)
10 {
11   return (... + args);
12 }
```


Defining a concept

- Definition of a concept is *probably* better:

```
 1 template<typename... Args>
 2 concept Addable = requires(Args... args)
 3 {
 4   (... + args);
 5   requires are_same_v<Args...>;
 6   requires sizeof...(Args) > 1;
 7   { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
 8 };
 9
10 template<typename... Args>
11 requires Addable<Args...>
12 auto Add(Args&&... args)
13 {
14   return (... + args);
15 }
```

Testing the created concept

```cpp
// Class template stub to create the different needed properties
template<bool nexcept, bool operatorPlus, bool validReturnType>
struct Stub {
  // Operator plus with controlled noexcept can be enabled
  Stub& operator+(const Stub& rhs) noexcept(nexcept)
      requires(operatorPlus && validReturnType)
  { return *this; }

  // Operator plus with invalid return type
  int operator+(const Stub& rhs) noexcept(nexcept)
      requires(operatorPlus && not validReturnType)
  { return {}; }
};

// Create the different stubs from the class template
using NoAdd               = Stub<true, false, true>;
using ValidClass          = Stub<true, true, true>;
using NotNoexcept         = Stub<false, true, true>;
using DifferentReturnType = Stub<true, true, false>;
```

Testing the created concept

```cpp
A Assert, that mixed types are not allowed
static_assert(not Addable<int, double>);

B Assert that Add is used with at least two parameters
static_assert(not Addable<int>);

C Assert that type has operator+
static_assert(Addable<int, int>);
static_assert(Addable<ValidClass, ValidClass>);
static_assert(not Addable<NoAdd, NoAdd>);

D Assert that operator+ is noexcept
static_assert(not Addable<NotNoexcept, NotNoexcept>);

E Assert that operator+ returns the same type
static_assert(not Addable<DifferentReturnType, DifferentReturnType>);
```

Abbreviated function templates

C++17:

```
1 template<typename T>
2 void DoLocked(T&& f)
3 {
4   std::lock_guard lock{globalOsMutex};
5
6   f();
7 }
```


Abbreviated function templates

C++20:

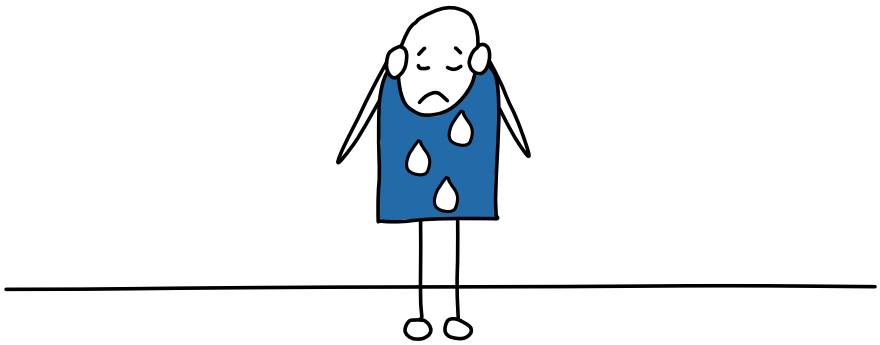
```
1 void DoLocked(std::invocable auto&& f)
2 {
3   std::lock_guard lock{globalOsMutex};
4
5   f();
6 }
```

- Note: Functions with **auto** parameters are always templates!

https://AndreasFertig.Info
post@AndreasFertig.Info

Error messages

```
1  template<typename T, typename U = void>  A
2  struct is_container : std::false_type {};
3
4  template<typename T>
5  struct is_container<
6    T,
7    std::void_t<typename T::value_type,  B
8                typename T::size_type,
9                typename T::allocator_type,
10               typename T::iterator,
11               typename T::const_iterator,
12               decltype(std::declval<T>().size()),
13               decltype(std::declval<T>().begin()),
14               decltype(std::declval<T>().end()),
15               decltype(std::declval<T>().cbegin()),
16               decltype(std::declval<T>().cend())>>
17 : std::true_type {};
18
19 struct A {};
20
21 static_assert(!is_container<A>::value);  C
22 static_assert(is_container<std::vector<int>>::value);
```

Error messages: Now helpful

```
1  template<typename T>
2  concept container = requires(T t)
3  {
4    typename T::value_type;
5    typename T::size_type;
6    typename T::allocator_type;
7    typename T::iterator;
8    typename T::const_iterator;
9    t.size();
10   t.begin();
11   t.end();
12   t.cbegin();
13   t.cend();
14 };
15
16 struct A {};
17
18 static_assert(not container<A>);
19 static_assert(container<std::vector<int>>);
```


C++20 non-type template parameters (NTTPs)

```
1 template<double D>
2 void Fun();
3
4 void Use()
5 {
6   Fun<+0.0>();
7   Fun<-0.0>();
8 }
```


C++20 NTTPs

C++20

```
1 template<typename CharT, std::size_t N>
2 struct fixed_string {
3   CharT data[N]{};
4
5   constexpr fixed_string(const CharT (&str)[N]) { std::copy_n(str, N, data); }
6 };
7
8 fixed_string fs{"Hello, C++20"};
```

C++20 NTTPs

C++20

```
1  template<fixed_string Str>   A Here we have a NTTP
2  struct FixedStringContainer {
3    B Use Str
4    void print() { std::cout << Str.data << '\n'; }
5  };
6
7  void Use()
8  {
9    C We can instantiate the template with a string
10   FixedStringContainer<"Hello, C++"> fc{};
11   fc.print();   D For those who believe it only if they see it
12 }
```


Lambdas with a template-head

C++20

```
1  int main()
2  {
3    auto max = [](auto x, auto y) {
4      return (x > y) ? x : y;
5    };
6
7    max(2, 3);    // ok
8    max(2, 3.0);  // not wanted
9  }
```

Lambdas with a template-head

C++20

```
1  int main()
2  {
3    auto max = []<typename T>(T x, T y)
4    {
5      return (x > y) ? x : y;
6    };
7
8    max(2, 3);  // ok
9    // max(2, 3.0);  // does not compile anymore
10 }
```


Lambdas with a template-head

C++20

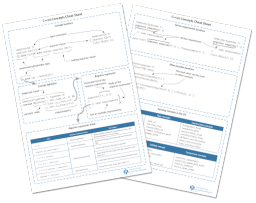
```
1 auto lamb = []<typename T>(std::vector<T> t) {};
2 std::vector<int> v{};
3
4 lamb(v);
5 // lamb(20);
```

```
1 auto lamb = []<size_t N>(std::array<int, N> x) {};
2
3 std::array<int, 2> a{};
4
5 lamb(a);
```

}

I am Fertig.

C++20 Concepts Cheat Sheet



fertig.to/subscribe

Used Compilers & Typography

Used Compilers

- Compilers used to compile (most of) the examples.
  - g++ 11.1.0
  - clang version 13.0.0

Typography

- Main font:
  - Camingo Dos Pro by Jan Fromm (https://janfromm.de/)
- Code font:
  - CamingoCode by Jan Fromm licensed under Creative Commons CC BY-ND, Version 3.0 http://creativecommons.org/licenses/by-nd/3.0/

References

Images:

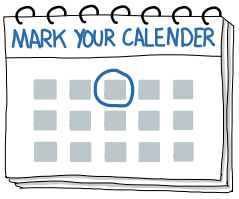
22: Franziska Panter
24: Franziska Panter
34: Franziska Panter

Upcoming Events

Training Classes

- *C++ Clean Code – Best Practices für Programmierer*, golem Akademie, March 29 - 31, 2022

For my upcoming talks you can check https://andreasfertig.info/talks/.
For my courses you can check https://andreasfertig.info/courses/.
Like to always be informed? Subscribe to my newsletter: https://andreasfertig.info/newsletter/.



About Andreas Fertig

Photo: Kristijan Matic www.kristijanmatic.de

Andreas Fertig, CEO of Unique Code GmbH, is an experienced trainer and lecturer for C++ for standards 11 to 20.
Andreas is involved in the C++ standardization committee, in which the new standards are developed. At international conferences, he presents how code can be written better. He publishes specialist articles, e.g., for iX magazine, and has published several textbooks on C++.
With C++ Insights (https://cppinsights.io), Andreas has created an internationally recognized tool that enables users to look behind the scenes of C++ and thus to understand constructs even better.
Before working as a trainer and consultant, he worked for Philips Medizin Systeme GmbH for ten years as a C++ software developer and architect focusing on embedded systems.