C++20 Templates - The next level

Concepts and more

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

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

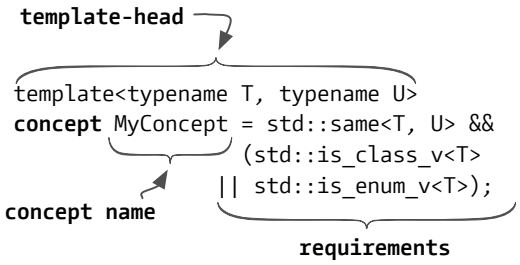
fertig

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`**.

Variadic template parameters of the same type

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


Variadic template parameters of the same type

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

```
 template<typename T, typename... Ts>
constexpr bool are_same_v = std::conjunction_v<std::is_same<T, Ts>...>;

template<typename T, typename...>
struct first_arg {
  using type = T;
};

template<typename... Args>
using first_arg_t = typename first_arg<Args...>::type;

```

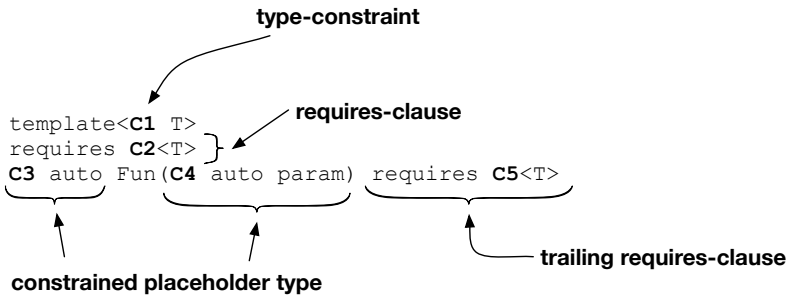
```
template<typename... Args>
std::enable_if_t<are_same_v<Args...>, first_arg_t<Args...>>
Add(Args&&... args) noexcept
{
  return (... + args);
}
```

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 }
```

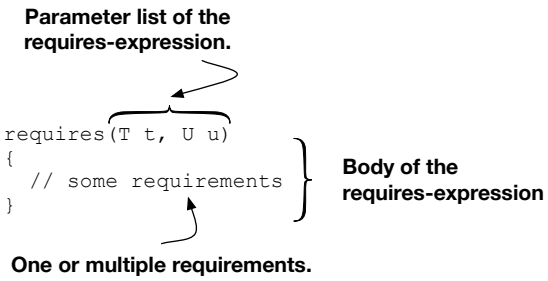

Application areas for Concepts



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 a **`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!

Dependent destructor

```
1  struct COMLike {
2    ~COMLike() {}   A Make it not default destructible
3
4    void Release();   B Release all data
5
6    // Some data fields
7  };
8
9  struct DefaultDestructible {};   C A type which is default destructible
10
11 static_assert(not std::is_trivially_destructible_v<Wrapper<COMLike>>);
12 static_assert(std::is_trivially_destructible_v<Wrapper<DefaultDestructible>>);
```

Andreas Fertig
v1.0

C++20 Templates - The next level

21

Dependent destructor

```
1  template<typename T, typename = void>
2  struct has_release : std::false_type {};
3
4  template<typename T>
5  struct has_release<T, decltype(std::declval<T>().Release())> : std::true_type {};
6
7  template<typename T>
8  class Wrapper {
9    T mData;
10
11 public:
12   ~Wrapper()
13   {
14     if constexpr(has_release<T>::value) { mData.Release(); }
15   }
16 };
```

Andreas Fertig
v1.0

C++20 Templates - The next level

22

Dependent destructor

```
1  template<typename T>
2  class Wrapper {
3    T mData;
4
5  public:
6    ~Wrapper() requires requires(T t) { t.Release(); }
7    {
8      mData.Release();
9    }
10
11   ~Wrapper() = default;
12 };
```


Conditional explicit

```
1  struct B {};
2
3  struct A {
4    A() = default;
5    explicit A(const B&);   (A) Marked explicit
6    operator B() const;
7  };
8
9  void Fun(A a) {}
10
11 void Use()
12 {
13   Fun(A{});
14   // Fun(B{}); (B) Will not compile due to explicit ctor
15 }
```

Conditional explicit

```
1  template<typename T>
2  struct Wrapper {
3    template<typename U>
4    Wrapper(const U&);
5  };
6
7  void Fun(Wrapper<A> a)   A Takes Wrapper<A> now
8  {}
9
10 void Use()
11 {
12   Fun(A{});
13   Fun(B{});   B Does compile!
14 }
```


Conditional explicit

```
1  template<typename T>
2  struct Wrapper {
3    template<typename U>
4     explicit(not std::is_convertible_v<U, T>)  Wrapper(const U&);
5  };
6
7  void Fun(Wrapper<A> a) {}
8
9  void Use()
10 {
11   Fun(A{});
12   //Fun(B{});   A Does not compile anymore
13 }
```

Conditional explicit

```
1  struct A {
2    A() = default;
3    explicit A(const B&) {}
4    explicit operator B() const;   A The conversion to B is now explicit
5  };
6
7  template<typename T>
8  struct Wrapper {
9    template<typename U>
10   explicit(not std::is_convertible_v<U, T>) Wrapper(const U&);
11
12   template<typename U>
13   operator U() const;   B Try to make Wrapper behave correct
14 };
15
16 void Fun(Wrapper<A> a)
17 {
18   B b = a;   C This should not, but does compile
19 }
```


Conditional explicit

```
1  struct A {
2    A() = default;
3    explicit A(const B&) {}
4    explicit operator B() const;
5  };
6
7  template<typename T>
8  struct Wrapper {
9    template<typename U>
10   explicit(not std::is_convertible_v<U, T>) Wrapper(const U&);
11
12   template<typename U>
13   A Make Wrapper behave correct
14    explicit(not std::is_convertible_v<T, U>)  operator U() const;
15 };
16
17 void Fun(Wrapper<A> a)
18 {
19   B b = static_cast<B>(a);   B Compiles only if we are explicit
20 }
```

Conditional explicit

```cpp
struct A {
  A() = default;
  explicit A(int);
};
```

```cpp
struct B {
  B() = default;
  B(int);
};
```

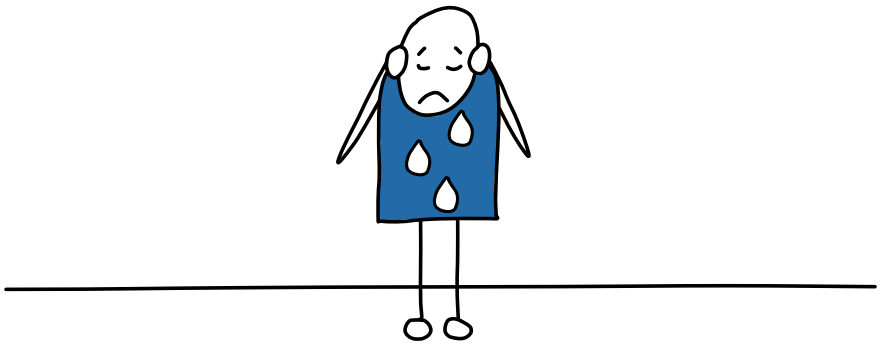

Conditional explicit

```cpp
struct A {
  A() = default;
  explicit A(int);
};
```

```cpp
struct B {
  B() = default;
  explicit(false)  B(int);
};
```

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"};
```


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

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 }
```


Templated Lambdas

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  }
```

Templated Lambdas

C++20

```cpp
int main()
{
  auto max = []<typename T>(T x, T y)
  {
    return (x > y) ? x : y;
  };

  max(2, 3);  // ok
  // max(2, 3.0);  // does not compile anymore
}
```


Templated Lambdas

C++20

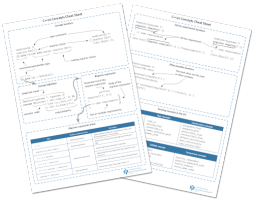
```cpp
auto lambda = []<typename T>(std::vector<T> t){};
std::vector<int> v{};

lambda(v);
// lambda(20);
```

```cpp
auto l = []<size_t N>(std::array<int, N> x) {};

std::array<int, 2> a{};

l(a);
```

}

I am Fertig.

C++20 Concepts Cheat Sheet



andreasfertig.info/newsletter/

Used Compilers & Typography

- Compilers used to compile (most of) the examples.
  - g++ 10.2.0
  - clang version 11.0.0 (https://github.com/llvm/llvm-project.git 176249bd6732a8044d457092ed932768724a6f06)
- 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/

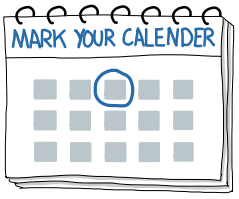
References

Images:

Upcoming Events

- *C++ Insights: How stuff works, C++20 and more!*, Meeting C++ online, April 08
- *C++: • Demystified*, ADC++, May 18

- *C++20: Five Features in Five Weeks*, Andreas Fertig, March 30 - April 27
- *Programming with C++11 to C++17*, Andreas Fertig, April 12 - 16
- *C++1x für eingebettete Systeme*, ADC++, May 17
- *C++ Clean Code – Best Practices für Programmierer*, golem Akademie, June 07 - 11
- *Programmieren mit C++20*, Andreas Fertig, September 27 - 29
- *C++1x für eingebettete Systeme*, QA Systems, October 14 - 15



About Andreas Fertig

Photo: Kristijan Matic www.kristijanmatic.de

Andreas Fertig is the CEO of Unique Code GmbH, which offers training and consulting for C++ specialized in embedded systems. He worked for Philips Medizin Systeme GmbH for ten years as a C++ software developer and architect focusing on embedded systems.
Andreas is involved in the C++ standardization committee. He is a regular speaker at conferences internationally. Textbooks and articles by Andreas are available in German and English.
His passion for teaching people how C++ works is why he created C++ Insights (https://cppinsights.io).