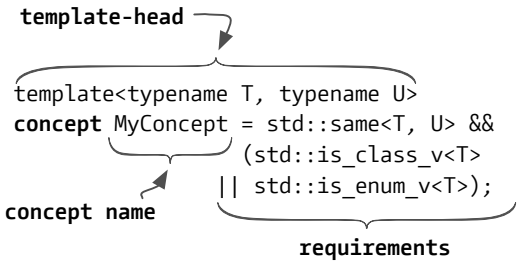
C++20 Templates

The Next Level

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

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;

```

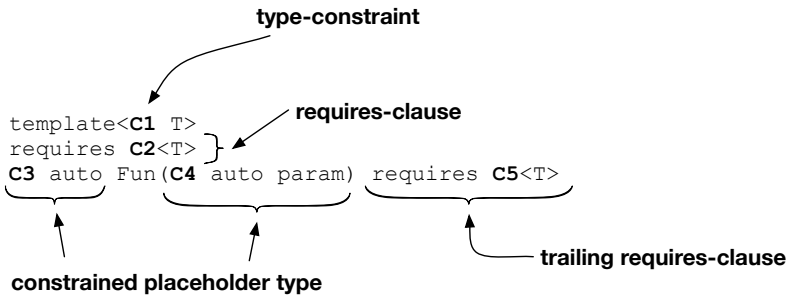
```
template<typename... Args>
std::enable_if_t<are_same_v<Args...>, first_arg_t<Args...>>
Add(Args&&... args) noexcept
{
  return (... + args);
}
```

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 }
```

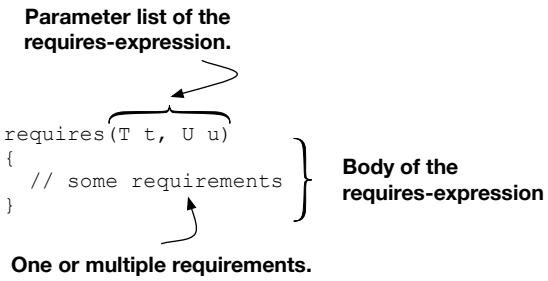

Application areas for Concepts



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.
- Optionally, before the trailing return type, it can be checked with **noexcept** whether the expression is **noexcept**.

```
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 -> same_as<first_arg_t<Args...>>;
9 }
```

```
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.

```
template<typename... Args>
requires requires(Args... args)
{
  (... + args);
  requires are_same_v<Args...>;
  requires sizeof...(Args) > 1;
  { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
}
auto Add(Args&&... args)
{
  return (... + args);
}
```

Defining a concept

- Definition of a concept is *probably* better:

```
template<typename... Args>
concept Addable = requires(Args... args)
{
  (... + args);
  requires are_same_v<Args...>;
  requires sizeof...(Args) > 1;
  { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
};

template<typename... Args>
requires Addable<Args...>
auto Add(Args&&... args)
{
  return (... + args);
}
```

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!

Error messages

- Check whether a type meets the requirements of a Standard Template Library (STL) container.
- Disadvantages with C++17
  - `is_container` exists twice.
  - Functioning of (A) and (B) difficult to see.
  - Duplication of **`decltype`** and `std::declval`.
  - Elimination of `::value` (C) requires even more code.
  - Error message only for experts and even then not very helpful.
- We have learned to live with it.

```
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

- Check whether a type meets the requirements of a STL container.
- The new world: Concepts.
  - Only one type `container`.
  - No `std::true_type`.
  - No more duplication of **`decltype`** and `std::declval`.
  - Looks a lot more like *normal* code.
  - error messages?

```
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>>);
```

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

```
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 }
```

Templated Lambdas

C++20

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

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

```cpp
#include <array>

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

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

  l(a);
}
```


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

C++20

```cpp
template<typename CharT, std::size_t N>
struct fixed_string {
  CharT data[N]{};

  constexpr fixed_string(const CharT (&str)[N]) { std::copy_n(str, N, data); }
};

fixed_string fs{"Hello, C++20"};
```

C++20 NTTPs

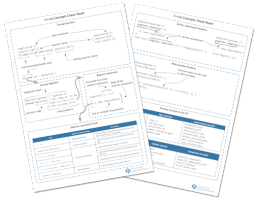
C++20

```
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 }
```


}

I am Fertig.

C++20 Concepts Cheat Sheet



andreasfertig.info/newsletter/

Used Compilers & Typography

Used Compilers

- 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)

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:

27: Franziska Panter

Upcoming Events

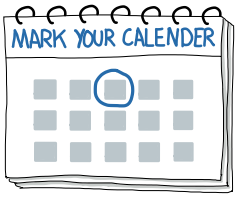
Talks

- *C++ Insights: How stuff works, Lambdas and more!*, Cpp Europe, 2021-02-23
- *C++20 Templates - The next level: Concepts and more*, ACCU, 2021-03-13

Training Classes

- *Programmieren mit C++11 bis C++17*, Andreas Fertig, February 22 - 26, 2021
- *C++ Clean Code – Best Practices für Programmierer*, golem Akademie, March 08 - 12, 2021
- *C++20: Five Features in Five Weeks*, Andreas Fertig, March 30 - April 27, 2021
- *Programming with C++11 to C++17*, Andreas Fertig, April 12 - 16, 2021
- *C++1x für eingebettete Systeme*, QA Systems, October 14 - 15, 2021

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 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).