C++20 Templates

Die nächste Generation: Concepts

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

Concepts

- Mit Concepts können wir Anforderungen (requirements) für einen Typ formulieren.
- Vergleichbar mit `std::enable_if`.
- Concepts bestehen aus der Definition des Konzepts (**`concept`**) und Requirements (**`requires`**):

```
template<typename T, typename U>
concept MyConcept = std::same_v<T, U> &&
                    (std::is_class_v<T>
                    || std::is_enum_v<T>);
```


- Ein Konzept ist immer ein Template und ist durch das neue Schlüsselwort **`concept`** zu erkennen. Ein Konzept besteht selbst aus anderen Konzepten oder Anforderungen. Letztere werden durch das Schlüsselwort **`requires`** definiert.

© 2021 Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info

Variadisches Template Parameter mit gleichem Typ

```
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
```


Variadische Template Parameter mit gleichem Typ

C++17 Variante: `enable_if` um Instantiierung zu blockieren.

```
1  template<typename T, typename... Ts>
2 constexpr 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 }
```


© 2021 Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info

Variadische Template Parameter mit gleichem Typ

C++20 Variante: `are_same_v` als Anforderung.

```
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 }
```

Einsatzorte für Konzepte

© 2021 Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info

Da geht mehr

- Aktuell verhindert Add nur
  - **A** keine gemischten Typen.
- Die aktuelle Version von Add lässt vieles unspezifiziert:
  - **B** Add kann unsinniger Weise mit nur einem Parameter aufgerufen werden.
  - **C** Der verwendete Typ Args muss die Operation + unterstützen.
  - **D** Die Operation + sollte **noexcept** sein da Add selbst **noexcept** ist.
  - **E** Der Rückgabetyp der Operation + sollte dem von Args entsprechen.

Requires-Expression

Parameter list of the requires-expression.

```
requires(T t, U u)
{
  // some requirements
}
```

Body of the requires-expression

One or multiple requirements.

- Requires-Clause (C3, C5) ist ein boolscher Ausdruck. Eine Requires-Expression ist komplizierter. Hallo, **noexcept**.
- Wir können eine Requires-Clause wie ein **if** ansehen, dass einen boolschen Ausdruck auswertet und eine Requires-Expression liefert einen solchen boolschen Wert.

© 2021 Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info

Requirement Arten

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

Simple requirement

- Prüft, ob ein Ausdruck valid ist und compilieren wird.

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

```
struct NoAdd{};

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

Nested requirement

- Wertet den boolschen Rückgabewert eines Ausdrucks aus.

```
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

- Prüft den Rückgabetyp eines Ausdrucks. Ist an den geschweiften Klammern und dem trailing-Return-Typ zu erkennen.
- Optional kann vor dem trailing Return-Typ mit **noexcept** geprüft werden, ob der Ausdruck **noexcept** ist.

```
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
```

© 2021 Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info

Ad hoc Constraints: `requires requires`

- Die Requirements werden direkt nach der Requires-Clause definiert (C2, C5).
- Das erste **`requires`** leitet die Requires-Clause ein, das zweite **`requires`** beginnt die Requires-Expression.
- Erster Hinweis auf einen 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 }
```

Ein Konzept definieren

- Definition eines Konzepts ist *wahrscheinlich* besser:

```
 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 }
```

© 2021 Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info

Das erstellte Konzept testen

```
 1 // Class template stub to create the different needed properties
 2 template<bool nexcept, bool operatorPlus, bool validReturnType>
 3 struct Stub {
 4   // Operator plus with controlled noexcept can be enabled
 5   Stub& operator+(const Stub& rhs) noexcept(nexcept)
 6       requires(operatorPlus && validReturnType)
 7   { return *this; }
 8
 9   // Operator plus with invalid return type
10   int operator+(const Stub& rhs) noexcept(nexcept)
11       requires(operatorPlus && not validReturnType)
12   { return {}; }
13 };
14
15 // Create the different stubs from the class template
16 using NoAdd               = Stub<true, false, true>;
17 using ValidClass          = Stub<true, true, true>;
18 using NotNoexcept         = Stub<false, true, true>;
19 using DifferentReturnType = Stub<true, true, false>;
```

Das erstellte Konzept testen

```
 1 (A) Assert, that mixed types are not allowed
 2 static_assert(not Addable<int, double>);
 3
 4 (B) Assert that Add is used with at least two parameters
 5 static_assert(not Addable<int>);
 6
 7 (C) Assert that type has operator+
 8 static_assert(Addable<int, int>);
 9 static_assert(Addable<ValidClass, ValidClass>);
10 static_assert(not Addable<NoAdd, NoAdd>);
11
12 (D) Assert that operator+ is noexcept
13 static_assert(not Addable<NotNoexcept, NotNoexcept>);
14
15 (E) Assert that operator+ returns the same type
16 static_assert(not Addable<DifferentReturnType, DifferentReturnType>);
```

© 2021 Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info

Abgekürzte Funktions-Templates

C++17:

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

Andreas Fertig v1.0 C++20 Templates 17

Abgekürzte Funktions-Templates

C++20:

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

- Merke: Funktionen mit **auto**-Parametern sind immer Templates!

Andreas Fertig v1.0 C++20 Templates 18

Abhängiger Destruktor

```
 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>>);
```

Abhängiger Destruktor

```
 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 };
```

© 2021 Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info

Abhängiger Destruktor

```
 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 };
```

Fehlermeldungen

- Prüfen, ob ein Typ die Anforderungen eines Standard Template Library (STL) Containers erfüllt.
- Nachteile mit C++17
  - is_container existiert zweimal.
  - Funktionsweise von (A) und (B) schwierig zu erkennen.
  - Duplikation von **decltype** und std::declval.
  - Eliminierung von ::value (C) erfordert noch mehr Code.
  - Fehlermeldung nur für Experten und auch dann wenig hilfreich.
- Wir haben gelernt damit zu leben.

```
 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);
```

© 2021 Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info

Fehlermeldungen: Jetzt hilfreich

- Prüfen, ob ein Typ die Anforderungen eines STL Containers erfüllt.
- Die neue Welt: Concepts.
  - Nur noch ein Type `container`.
  - Kein `std::true_type`.
  - Keine Duplikation von **`decltype`** und `std::declval` mehr.
  - Sieht wesentlich mehr aus wie *normaler* Code.
  - Fehlermeldungen?

```
 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>>);
```

Wann sollte ich ein Konzept definieren

- Ein Konzept sollte eine Aussagekraft haben.
- Ein Konzept sollte in der Regel mehrere Requirements zusammenfassen.
  - Vermeiden Sie es Concepts in kleine Fragmente zu zerlegen, bei denen die Bedeutung verloren geht.
- Wie immer: Nur weil etwas geht, muss man es nicht tun!

Anzeichen für die Definition eines Konzepts:

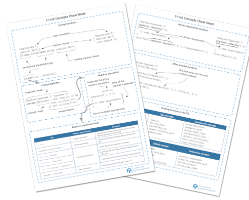
- Wenn das Prädikat wiederverwendet wird, ist das ein Anzeichen für ein Konzept.
- Wenn das Prädikat semantische Anforderung an die mit dem Typ verbundenen Operationen stellt.
- Wenn das Prädikat in Schnittstellen auftaucht.

© 2021 Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info

}

Ich bin Fertig.

C++20 Concepts Cheat Sheet



andreasfertig.info/newsletter/

Verwendete Compiler & Typografie

Verwendete Compiler

- Compiler welche zum Übersetzen des (meisten) Codes verwendet wurden.
  - g++ 10.2.0
  - clang version 11.0.0 (https://github.com/llvm/llvm-project.git 176249bd6732a8044d457092ed932768724a6f06)

Typografie

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

© 2021 Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info

Quellen

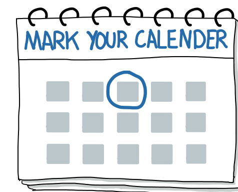
Bilder:
28: Franziska Panter

Nächste Events

Training Classes

- *Programmieren mit C++11 bis C++17*, Andreas Fertig, February 22, 2021
- *C++ Clean Code – Best Practices für Programmierer*, golem Akademie, March 08, 2021
- *Programming with C++11 to C++17*, Andreas Fertig, April 12, 2021
- *C++1x für eingebettete Systeme*, QA Systems, October 14, 2021

Zukünftige Vorträge: https://andreasfertig.info/talks/.
Mehr zu meinen Trainingsangeboten gibt es hier: https://andreasfertig.info/training/.
Immer aktuell Informiert? Hier geht es zum Newsletter: https://andreasfertig.info/newsletter/.



© 2021 Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info

Über Andreas Fertig

Foto: Kristijan Matic www.kristijanmatic.de

Andreas Fertig ist Geschäftsführer der Unique Code GmbH, die Schulungen und Beratung für C++ anbietet mit dem Spezialgebiet eingebettete Systeme. Er arbeitete zehn Jahre für die Philips Medizin Systeme GmbH als C++ Softwareentwickler und Architekt mit Schwerpunkt auf eingebetteten Systemen.
Andreas engagiert sich im C++ Standardisierungskomitee. Als Referent ist er regelmäßig international auf Konferenzen anzutreffen. Fachbücher sowie Fachartikel von Andreas gibt es in Deutsch und Englisch.
Andreas hat eine Leidenschaft dafür, Menschen beizubringen, wie C++ funktioniert, weshalb er C++ Insights (cppinsights.io) geschaffen hat.

Andreas Fertig v1.0 C++20 Templates 29