5.3 Eingeschränkte Schablonen

5.3.1 Grundprinzip

- Seit C++20 können bei der Definition einer Schablone (oder bei einer Elementfunktion einer Typschablone, die selbst keine Schablone ist) *Einschränkungen* (constraints) für die Schablonenparameter angegeben werden.

- Die Schablone kann dann nur verwendet werden, wenn die Schablonenargumente diese Einschränkungen erfüllen.

- Andernfalls bemerkt der Übersetzer das unmittelbar an der Verwendungsstelle der Schablone und nicht erst bei der – möglicherweise tief verschachtelten rekursiven – Expansion der Schablone, was normalerweise zu kürzeren und besser verständlichen Fehlermeldungen führt.

- Außerdem werden Einschränkungen bereits bei der Auflösung überladener Funktionsschablonen berücksichtigt, das heißt:
  - Wenn bei der Expansion der Schablone, die anhand der Parametertypen am besten passt, ein Fehler auftritt, ist das Programm fehlerhaft, und es wird nicht versucht, eine weniger gut passende Schablone zu verwenden.
  - Wenn die am besten passende Schablone aber aufgrund einer nicht erfüllten Einschränkung nicht verwendet werden kann, kann immer noch eine weniger gut passende Schablone verwendet werden, deren Einschränkung erfüllt ist.

- Desweiteren ist es möglich, mehrere Funktionsschablonen (oder Elementfunktionen von Typschablonen) mit dem gleichen Namen und den gleichen Parametertypen zu definieren, sofern sie unterschiedliche Einschränkungen besitzen.

- Eine *Eigenschaft* (concept) ist eine benannte Menge, die aus einer oder mehreren Einschränkungen besteht.

5.3.2 Eigenschaften und Anforderungen

- Die Definition einer Eigenschaft (concept) besteht aus
  - dem Schlüsselwort `template` und einer Schablonenparameterliste,
  - dem Schlüsselwort `concept`, dem Namen der Eigenschaft und einem Gleichheitszeichen,
  - einem Ausdruck mit Typ `bool` (in dem Operatoren mit niedrigerem Vorrang als `||` nur innerhalb von Klammern auftreten dürfen), der normalerweise von den Schablonenparametern abhängt und dessen Wert bereits zur Übersetzungszeit feststehen muss,
  - und einem abschließenden Semikolon.
- Der Ausdruck ist häufig eine logische Verknüpfung bereits früher definierter Eigenschaften und/oder ein *Anforderungsausdruck* (requires expression), der aus folgenden Teilen besteht:
  - dem Schlüsselwort `requires`,
  - einer optionalen Parameterliste analog zu einer Funktionsparameterliste, die normalerweise die Schablonenparameter der Eigenschaft verwendet,
  - und einem Rumpf mit beliebig vielen Anforderungen in geschweiften Klammern (einfache, kombinierte, geschachtelte oder Typanforderungen), die normalerweise die o. g. Parameter und eventuell auch umgebende Schablonenparameter verwenden und jeweils mit einem Semikolon abgeschlossen sind.

- Der Wert eines Anforderungsausdrucks ist für eine bestimmte Belegung der Schablonenparameter genau dann `true`, wenn beim Ersetzen der Schablonenparameter durch ihre Belegungen im Rumpf des Ausdrucks keine fehlerhaften Typen oder Ausdrücke entstehen und wenn alle Anforderungen mit dieser Belegung erfüllt sind (vgl. § 5.3.3, § 5.3.4, § 5.3.5, § 5.3.6).
- Wenn der Ausdruck in der Definition einer Eigenschaft für eine bestimmte Belegung der Schablonenparameter den Wert `true` besitzt, ist die Eigenschaft für diese Belegung der Schablonenparameter *erfüllt* und besitzt dann ebenfalls den Wert `true`.
- Obwohl Eigenschaften Schablonen sind, können für sie selbst keine Einschränkungen angegeben werden, und sie können nicht spezialisiert werden.

5.3.3 Einfache Anforderungen

- Eine einfache Anforderung (simple requirement) ist ein beliebiger Ausdruck (der nicht mit dem Schlüsselwort `requires` beginnt, vgl. § 5.3.6).

- Sie ist immer erfüllt, wenn der Ausdruck mit der jeweiligen Belegung der Schablonenparameter korrekt ist (vgl. § 5.3.2). Der Ausdruck wird zur Laufzeit des Programms nie ausgewertet.

- Zum Beispiel:

```
template <typename T>
concept Multable = requires (T x) { x * x; };
```

- Damit der Ausdruck `x * x` korrekt ist, muss es für den Typ `T` einen Multiplikationsoperator geben (der allerdings noch einen beliebigen Resultattyp haben könnte).

- Dementsprechend besitzt der `requires`-Ausdruck – und damit auch die Eigenschaft `Multable<T>` – beispielsweise für alle numerischen Typen `T` den Wert `true` und für alle Zeigertypen den Wert `false`.

5.3.4 Kombinierte Anforderungen

- Eine kombinierte Anforderung (compound requirement) ist von der Gestalt `{ expr } -> constr`.
- Sie ist erfüllt, wenn der Ausdruck mit der jeweiligen Belegung der Schablonenparameter korrekt ist (vgl. § 5.3.2) und sein Typ `T` die Bedingung `constr` erfüllt. Auch hier wird der Ausdruck zur Laufzeit nie ausgewertet.
- Dabei ist `constr` der Name einer Eigenschaft `C` mit optionalen Schablonenargumenten `<args>`.
- Der Typ `T` erfüllt diese Bedingung genau dann, wenn `C<T>` bzw. `C<T, args>` erfüllt ist (vgl. § 5.3.2), d. h. der Typ des Ausdrucks wird als erstes Schablonenargument in die Eigenschaft `C` eingesetzt. (Dementsprechend muss der erste Schablonenparameter von `C` ein Typparameter sein.)
- Zum Beispiel:

```
template <typename T>
concept Multable = requires (T x) {
  { x * x } -> same_as<T>;
};
```

Jetzt ist `Multable<T>` für einen Typ `T` nur erfüllt, wenn der Multiplikationsoperator für `T` auch Resultattyp `T` besitzt.

- `same_as` ist eine Eigenschaft, die in der Datei `<concepts>` der Standardbibliothek definiert ist und die für zwei Typen `X` und `Y` genau dann erfüllt ist, wenn diese Typen gleich sind.
- Wenn anstelle von `same_as` die ebenfalls in der Standardbibliothek definierte Eigenschaft `convertible_to` verwendet wird, genügt es, wenn der Resultattyp des Multiplikationsoperators implizit in den Typ `T` umgewandelt werden kann.
- Das könnte aber auch fast gleichbedeutend mit einer einfachen Anforderung (vgl. § 5.3.3) ausgedrückt werden:

```
template <typename T>
concept Multable = requires (T x) {
  T(x * x);
};
```

  Der Konstruktoraufruf `T(x * x)` ist genau dann korrekt, wenn `x * x` explizit oder implizit in den Typ `T` umgewandelt werden kann, d. h. anders als bei `convertible_to`, werden hier auch Konstruktoren von `T` berücksichtigt, die `explicit` deklariert sind (vgl. § 3.2).
- Wenn `T{x * x}` statt `T(x * x)` verwendet wird, darf die Umwandlung jedoch keine „narrowing conversion“ sein (vgl. § 5.2.2).

5.3.5 Typanforderungen

- Eine Typanforderung (type requirement) besteht aus dem Schlüsselwort `typename` und einem allgemeinen, ggf. qualifizierten Namen, der normalerweise von den umgebenden Schablonenparametern abhängt.
- Sie ist erfüllt, wenn der Name mit der jeweiligen Belegung der Schablonenparameter tatsächlich einen Typ bezeichnet.
- Zum Beispiel:

```
template <typename X, typename Y>
concept HaveCommonType = requires {
  typename std::common_type<X, Y>::type;
};
```

- `HaveCommonType<X, Y>` ist für Typen `X` und `Y` genau dann erfüllt, wenn der Typ `std::common_type<X, Y>` einen inneren Typ mit dem Namen `type` besitzt, was genau dann der Fall ist, wenn `X` und `Y` einen „gemeinsamen“ Typ besitzen, in den beide implizit umgewandelt werden können (vgl. § 5.4.4).
- Anstelle von `std::common_type<X, Y>::type` kann auch der Alias `std::common_type_t<X, Y>` verwendet werden.
- Allerdings gibt es in der Standardbibliothek bereits eine Eigenschaft `common_with` mit derselben Bedeutung wie `HaveCommonType`.

5.3.6 Geschachtelte Anforderungen

- Eine geschachtelte Anforderung (nested requirement) besteht aus dem Schlüsselwort `requires` und einem Ausdruck wie in der Definition einer Eigenschaft (vgl. § 5.3.2).
- Sie ist erfüllt, wenn der Ausdruck mit der jeweiligen Belegung der Schablonenparameter korrekt ist (vgl. § 5.3.2) und den Wert `true` besitzt.
- Geschachtelte Anforderungen können verwendet werden, um zusätzliche Einschränkungen zu formulieren, die von den Parametern des umgebenden Anforderungsausdrucks abhängen.
- Zum Beispiel:

```
template <typename T>
concept ArrayPointerEquiv = requires (T x, int i) {
  requires same_as<decltype(*(x+i)), decltype(x[i])>;
};
```

- Die Eigenschaft `ArrayPointerEquiv<T>` ist genau dann erfüllt, wenn die Ausdrücke `*(x+i)` und `x[i]` für einen Wert `x` des Typs `T` und eine ganze Zahl `i` denselben Typ besitzen (der jeweils mit dem Schlüsselwort `decltype` ermittelt wird), d. h. wenn für den Typ `T` die gleiche Äquivalenz wie für Reihen- und Zeigertypen gilt (vgl. § 2.3.8). (Dass die beiden Ausdrücke zur Laufzeit auch den gleichen Wert liefern, kann natürlich nicht ausgedrückt werden.)

- Beachte: Ohne das Schlüsselwort `requires` vor `same_as` würde nur eine einfache Anforderung definiert werden, die erfüllt ist, sobald der Ausdruck korrekt ist, egal ob sein Wert `true` oder `false` ist.

5.3.7 Anforderungsklauseln

- Einschränkungen für die Parameter einer Schablone können nach der Schablonenparameterliste mit einer *Anforderungsklausel* (requires clause) angegeben werden, die aus dem Schlüsselwort `requires` und einem Ausdruck wie bei der Definition einer Eigenschaft (vgl. § 5.3.2) besteht (wobei alle Operatoren außer `&&` und `||` hier nur innerhalb von Klammern auftreten dürfen).

- Die Schablone kann dann nur für Belegungen der Schablonenparameter verwendet werden, für die dieser Ausdruck den Wert `true` besitzt.

- Wie bei der Definition einer Eigenschaft, ist dieser Ausdruck häufig eine logische Verknüpfung von Eigenschaften und/oder ein Anforderungsausdruck.

- Zum Beispiel (vgl. § 5.1.1 und § 5.2.2):

```
template <typename T> requires Multable<T>
T square (T x) { return x * x; }

template <typename X, typename Y>
requires Multable<X> && Multable<Y>
struct MultablePair { ..... };
```

- Alternativ kann die Anforderungsklausel bei einer Funktionsschablone (oder auch bei einer Elementfunktion einer Typschablone, die selbst keine Schablone ist) auch nach der Funktionsparameterliste angegeben werden, zum Beispiel:

```
template <typename T>
T square (T x) requires Multable<T> { return x * x; }
```

- Tatsächlich können bei einer Funktionsschablone auch Anforderungsklauseln an beiden o. g. Stellen angegeben werden, die dann beide erfüllt sein müssen.

- Anforderungsklauseln und Anforderungsausdrücke sind grundsätzlich verschiedene Dinge, obwohl beide mit dem Schlüsselwort `requires` beginnen. Allerdings kann der Ausdruck einer Anforderungsklausel auch ein Anforderungsausdruck sein, zum Beispiel:

```
template <typename T>
requires requires (T x) { T(x * x); }
T square (T x) { return x * x; }
```

  Das erste Schlüsselwort `requires` leitet die Anforderungsklausel ein, das zweite den Ausdruck dieser Klausel, bei dem es sich um einen Anforderungsausdruck handelt.

5.3.8 Typeinschränkungen

- Außerdem kann eine Einschränkung für einen Typparameter `T` auch direkt bei seiner Definition angegeben werden, indem anstelle des Schlüsselworts `typename` der Name einer Eigenschaft `C` mit optionalen Schablonenargumenten `<args>` angegeben wird.

- Die Schablone kann dann nur verwendet werden, wenn die Eigenschaft `C<T>` bzw. `C<T, args>` für die Belegung des Parameters `T` erfüllt ist (vgl. § 5.3.5), zum Beispiel:

```
template <Multable T>
T square (T x) { return x * x; }

template <Multable X, Multable Y>
struct MultablePair { ..... };
```

- Wenn es zusätzlich eine oder (bei Funktionsschablonen) zwei Anforderungsklauseln gibt, müssen diese zusätzlich erfüllt sein, zum Beispiel:

```
template <Multable X, typename Y> requires Multable<Y>
struct MultablePair { ..... };
```

5.3.9 Weitere Beispiele

- Generische Maximumfunktion (vgl. § 5.1.2):

```
template <typename T>
concept GtCmpable = requires (T x) { bool(x > x); };

template <GtCmpable T>
T maximum (T x, T y) { return x > y ? x : y; }
```

- Spezialisierung für Zeigertypen:

```
template <typename T>
concept EqCmpable = requires (T x) { bool(x == x); };

template <typename T>
concept HasMaximum = requires (T x) {
  { maximum(x, x) } -> convertible_to<T>;
};
```

```
template <EqCmpable T> requires HasMaximum<T>
T* maximum (T* x, T* y) {
  // Ein echter Zeiger soll immer größer als ein Nullzeiger sein.
  if (!x) return y;
  if (!y) return x;

  // Ansonsten soll x größer als y sein, wenn *x größer als *y
  // ist. Wenn *x und *y selbst Zeiger sind, wird dieses
  // Kriterium rekursiv angewandt.
  return maximum(*x, *y) == *x ? x : y;
}
```

- Mögliche Verwendung:

```
struct X {} x1, x2;
maximum(&x1, &x2)
```

Aufgrund des Typs `X*` der Funktionsargumente passt die Spezialisierung für Zeigertypen eigentlich besser als die allgemeine Schablone (vgl. § 5.1.2). Aber weil ihre Einschränkungen `EqCmpable` und `HasMaximum` für den Typ `X` nicht erfüllt sind, kann sie nicht verwendet werden und wird deshalb auch nicht expandiert (was sonst zu Fehlern führen würde, weil es für den Typ `X` weder einen Gleichheitsoperator noch eine Funktion `maximum` gibt). Stattdessen wird die allgemeine Schablone verwendet, deren Einschränkung `GtCmpable` für den Typ `X*` erfüllt ist, weil es für jeden Zeigertyp einen Größer-Operator gibt.

- Typkorrekte Matrixoperationen (vgl. § 5.2.3):

```
template <typename T>
concept Addable =
requires (T x) { { x + x } -> convertible_to<T>; };

template <Addable T, int M, int N>
Matrix<T, M, N> operator+ (Matrix<T, M, N> a, Matrix<T, M, N> b) {
  Matrix<T, M, N> c;
  for (int i = 1; i <= M; i++) for (int k = 1; k <= N; k++) {
    c.set(i, k, a.get(i, k) + b.get(i, k));
  }
  return c;
}
```

```
template <typename T, int L, int M, int N>
requires Addable<T> && Multable<T>
Matrix<T, L, N> operator* (Matrix<T, L, M> a, Matrix<T, M, N> b) {
  Matrix<T, L, N> c;
  for (int i = 1; i <= L; i++) for (int k = 1; k <= N; k++) {
    T sum = a.get(i, 1) * b.get(1, k);
    for (int j = 2; j <= M; j++) {
      // Verwendung von + und = anstelle von +=, weil Addable<T>
      // nur das Vorhandensein von + und nicht von += garantiert.
      sum = sum + a.get(i, j) * b.get(j, k);
    }
    c.set(i, k, sum);
  }
  return c;
}
```

- Paare (vgl. § 5.2.2):

```
template <typename X, typename Y>
class Pair {
  // Datenelemente.
  X x;
  Y y;
public:
  // Normaler Konstruktor zur Initialisierung mit zwei Werten.
  Pair (X x, Y y) : x{x}, y{y} {}

  // Zugriff auf die privaten Datenelemente.
  X getX () const { return x; }
  Y getY () const { return y; }

  // Konstruktorschablone zur Initialisierung mit einem Paar
  // eines anderen Typs Pair<U, V>, wenn U implizit in X und
  // V implizit in Y umgewandelt werden kann.
  template <convertible_to<X> U, convertible_to<Y> V>
  Pair (const Pair<U, V>& that)
                          : x(that.getX()), y(that.getY()) {}
};
```

5.3.10 Partielle Ordnung von Einschränkungen

Beispiel

- Für ein Objekt `c` eines beliebigen Containertyps (z. B `std::vector`, `std::list` oder `std::forward_list`) und ein Element `x` soll `add(c, x)` das Element `x` zum Container `c` hinzufügen, und zwar mittels `push_back`, wenn der Container diese Operation anbietet, oder sonst mittels `push_front`, wenn der Container diese Operation anbietet. (Das heißt, wenn der Container beide Operationen anbietet, soll `push_back` verwendet werden. Wenn er keine anbietet, soll der Aufruf von `add` fehlerhaft sein.)

- Zum Beispiel:

```
// vector bietet nur push_back an.
std::vector<int> v; add(v, 1);

// forward_list bietet nur push_front an.
std::forward_list<int> f; add(f, 1);

// list bietet sowohl push_front als auch push_back an.
std::list<int> l; add(l, 1);
```

Benötigte Eigenschaften

```
template <typename C, typename X>
concept HasPushFront = requires (C& c, const X& x) {
  c.push_front(x);
};

template <typename C, typename X>
concept HasPushBack = requires (C& c, const X& x) {
  c.push_back(x);
};
```

Möglichkeit 1

- Definition von zwei eingeschränkten Funktionsschablonen `add` mit disjunkten Einschränkungen:

```
template <typename C, typename X>
requires HasPushFront<C, X> && (!HasPushBack<C, X>)
void add (C& c, const X& x) { c.push_front(x); }

template <typename C, typename X>
requires HasPushBack<C, X>
void add (C& c, const X& x) { c.push_back(x); }
```

Möglichkeit 2

- Definition von zwei Funktionsschablonen mit überlappenden Einschränkungen sowie einer dritten Schablone mit der Schnittmenge der beiden Einschränkungen, die deshalb spezieller als die beiden anderen ist und deshalb ggf. bevorzugt wird:

```
template <typename C, typename X>
requires HasPushFront<C, X>
void add (C& c, const X& x) { c.push_front(x); }

template <typename C, typename X>
requires HasPushBack<C, X>
void add (C& c, const X& x) { c.push_back(x); }

template <typename C, typename X>
requires HasPushFront<C, X> && HasPushBack<C, X>
void add (C& c, const X& x) { c.push_back(x); }
```

- Nachteil: Die zweite und dritte Funktion enthalten genau denselben Code.
  Um diese Codeverdopplung zu vermeiden, könnte der Code in eine Hilfsfunktion verlagert werden, die dann von beiden Funktionen aufgerufen wird.

Möglichkeit 3

- Definition von zwei Funktionsschablonen, wobei die Einschränkung der zweiten Funktion eine echte Teilmenge der Einschränkung der ersten ist, sodass sie wiederum ggf. bevorzugt wird:

```
template <typename C, typename X>
requires HasPushFront<C, X> || HasPushBack<C, X>
void add (C& c, const X& x) { c.push_front(x); }

template <typename C, typename X>
requires HasPushBack<C, X>
void add (C& c, const X& x) { c.push_back(x); }
```

- Obwohl die erste Funktion aufgrund ihrer Einschränkung prinzipiell auch im Fall `HasPushBack` verwendet werden kann, wird in diesem Fall immer die zweite Funktion verwendet, weil ihre Einschränkung stärker ist.

Allgemeine Regeln

- Um zwei Einschränkungen zu vergleichen, werden beide zunächst *normalisiert*, indem Eigenschaften durch ihre definierenden Ausdrücke ersetzt werden.

- Um dann zu überprüfen, ob Einschränkung `Q` aus Einschränkung `P` folgt, wird `P` in *disjunktive Normalform* (d. h. eine Disjunktion von Konjunktionen von atomaren Einschränkungen) und `Q` in *konjunktive Normalform* (d. h. eine Konjunktion von Disjunktionen von atomaren Einschränkungen) gebracht.

- Dann gilt:
  - Einschränkung `Q` folgt aus Einschränkung `P`, wenn jede Disjunktion in der konjunktiven Normalform von `Q` aus jeder Konjunktion in der disjunktiven Normalform von `P` folgt.
  - Eine Disjunktion von atomaren Einschränkungen folgt aus einer Konjunktion von atomaren Einschränkungen, wenn beide eine identische atomare Einschränkung enthalten.
  - Zwei atomare Einschränkungen sind *identisch*, wenn es sich um *denselben Ausdruck* an *derselben Stelle des Programmcodes* handelt.

- Einschränkung `P` ist *stärker* als Einschränkung `Q`, wenn `Q` aus `P` folgt, aber nicht umgekehrt.

- Wenn die Einschränkung einer Schablone stärker als die Einschränkung einer anderen Schablone ist, wird die erste gegenüber der zweiten bevorzugt, sofern keine von ihnen aufgrund der Regel in § 5.1.2 spezieller als die andere ist.
  (Ebenso für Elementfunktionen einer Typschablone, die selbst keine Schablonen sind).

Anmerkungen

- Wenn bei der Definition einer Schablone mehrere Einschränkungen angegeben sind (eine oder mehrere Typeinschränkungen und/oder eine oder zwei Anforderungsklauseln), werden sie konjunktiv zu einer Einschränkung zusammengefasst.

- Jede explizite Einschränkung (selbst `requires true`) ist stärker als keine Einschränkung.

- Bei der Überführung in disjunktive oder konjunktive Normalform werden nur Anwendungen von `&&` und `||` berücksichtigt, die sich höchstens innerhalb von Klammern, aber nicht innerhalb von Anwendungen anderer Operatoren befinden.

- Insbesondere haben Anwendungen des Negationsoperators `!` keine besondere Bedeutung, sondern stellen normale atomare Einschränkungen dar.
  Damit ist z. B. `(!(!HasPushFront<C, X> && !HasPushBack<C, X>))` nicht gleichbedeutend mit `HasPushFront<C, X> || HasPushBack<C, X>`.

- Weil atomare Einschränkungen nur dann identisch sind, wenn sie von derselben Stelle des Programmcodes stammen, ist es sinnvoll, Einschränkungen möglichst „kleinteilig“ in Eigenschaften zu „verpacken“ und erst diese dann mit logischen Verknüpfungen zu kombinieren.

5.4 Weitere Themen

5.4.1 Variablenschablonen

- Beispiel: Leere Listen (vgl. § 5.2.6)

```
template <typename T>
const List<T> empty = nullptr;

List<int> ls = empty<int>;
```

- Beispiel: Fibonacci-Zahlen (vgl. § 5.2.5)

```
template <int N>
const int fib = fib<N-2> + fib<N-1>;

template <>
const int fib<0> = 0;

template <>
const int fib<1> = 1;

cout << fib<46> << endl;
```

- Beispiel: Wertebereiche numerischer Typen

```
#include <limits>
#include <utility>
using namespace std;

template <typename T>
const T minimum = numeric_limits<T>::min();

template <typename T>
const T maximum = numeric_limits<T>::max();

template <typename T>
const pair range = {minimum<T>, maximum<T>};

cout << range<int>.first << " " << range<int>.second << endl;
```

- Ebenso wie Funktionsschablonen (vgl. § 5.1.1), sind Variablenschablonen immer `inline` (vgl. § 2.12).
  Um eine Variablenschablone in einer Übersetzungseinheit verwenden zu können, muss es in dieser Übersetzungseinheit eine vollständige Definition geben.

5.4.2 Deduktion von Referenztypen

- Gegeben seien folgende Funktionsschablonen:

```
template <typename T> void f1 (T x);
template <typename T> void f2 (T& x);
template <typename T> void f3 (T&& x);
```

- Wenn `f1` mit einem L- oder R-Wert eines Typs `U` aufgerufen wird, wird in beiden Fällen `T` gleich `U` deduziert, d. h. `f1` kann nicht „erkennen“, ob es mit einem L- oder R-Wert aufgerufen wird.

- Wenn `f2` mit einem L-Wert eines Typs `U` aufgerufen wird, wird `T` gleich `U` deduziert, während ein Aufruf mit einem R-Wert fehlschlägt.

- Wenn `f3` mit einem L-Wert eines Typs `U` aufgerufen wird, wird `T` gleich `U&` deduziert (sodass `T&&` aufgrund der „reference collapsing rules“ in § 2.9.7 gleich `U&` ist!), während bei einem Aufruf mit einem R-Wert `T` gleich `U` deduziert wird, d. h. `f3` kann „erkennen“, ob es mit einem L- oder R-Wert aufgerufen wird.

- `T&&` wird deshalb auch als „universal“ oder „forwarding reference“ bezeichnet. (Der Unterschied zu einer normalen R-Wert-Referenz besteht darin, dass `T` hier kein fester Typ, sondern ein Typparameter ist.)

5.4.3 Untersuchung von Typen

- Hilfstypen:

```
struct true_type {
  static const bool value = true;
};

struct false_type {
  static const bool value = false;
};
```

- Vergleich von Typen:

```
// Allgemeine Typschablone.
template <typename X, typename Y>
struct is_same : false_type {};

// Spezialisierung für zwei gleiche Typen.
template <typename T>
struct is_same <T, T> : true_type {};

// Variablenschablone zur etwas angenehmeren Verwendung.
template <typename X, typename Y>
const bool is_same_v = is_same<X, Y>::value;
```

- Verwendungsmöglichkeiten:

```
is_same_v<int, signed int>                // true
is_same_v<int, unsigned int>              // false
is_same_v<char, signed char>              // false
is_same_v<char, unsigned char>            // false
is_same_v<int, int_least32_t>             // meist true
is_same_v<int, vector<int>::value_type> // true
```

❒ Identifizierung von Zeigertypen:

```
// Allgemeine Schablone.
template <typename T>
struct is_pointer : false_type {};

// Spezialisierungen für Zeigertypen.
template <typename T>
struct is_pointer <T*> : true_type {};

template <typename T>
struct is_pointer <T* const> : true_type {};

template <typename T>
struct is_pointer <T* volatile> : true_type {};

template <typename T>
struct is_pointer <T* const volatile> : true_type {};

// Variablenschablone zur etwas angenehmeren Verwendung.
template <typename T>
const bool is_pointer_v = is_pointer<T>::value;
```

- Entfernen von Referenzen:

```
// Hilfstyp.
template <typename T>
struct declare_type {
  using type = T;
};

// Allgemeine Schablone.
template <typename T>
struct remove_reference : declare_type<T> {};

// Spezialisierung für L-Wert-Referenzen.
template <typename T>
struct remove_reference <T&> : declare_type<T> {};

// Spezialisierung für R-Wert-Referenzen.
template <typename T>
struct remove_reference <T&&> : declare_type<T> {};

// Alias zur etwas angenehmeren Verwendung.
template <typename T>
using remove_reference_t = typename remove_reference<T>::type;
```

- Zur Bedeutung von `typename` in der Definition von `remove_reference_t`:
  - Wenn ein qualifizierter Name der Gestalt `x<y>::z` von einem Schablonenparameter abhängt, kann der Übersetzer nicht erkennen, ob dieser Name einen Typ bezeichnet oder nicht.
    (Selbst wenn die Schablone `x` bereits definiert ist, könnte das Element `z` in einer später definierten Spezialisierung eine andere Bedeutung haben.)
  - Diese Information kann für die syntaktische Analyse jedoch von entscheidender Bedeutung sein, zum Beispiel:
    - Wenn `x<y>::z` einen Typ bezeichnet, stellt `int a (x<y>::z)` die Deklaration einer Funktion `a` mit entsprechendem Parametertyp dar.
    - Andernfalls handelt es sich jedoch um die Definition einer Variablen `a`, die mit dem Wert `x<y>::z` initialisiert wird.
  - Wenn die Bedeutung von `x<y>::z` nicht bekannt ist, geht der Übersetzer davon aus, dass es sich *nicht* um einen Typ handelt, selbst wenn im vorliegenden Kontext ein Typ benötigt wird.
  - Um zum Ausdruck zu bringen, dass `x<y>::z` einen Typ bezeichnet, muss in diesem Fall das Schlüsselwort `typename` davorgesetzt werden.
  - Im konkreten Beispiel: `typename remove_reference<T>::type`
  - Ausnahme: Seit C++20 ist `typename` an dieser Stelle nicht mehr nötig.

- Anwendung von `remove_reference_t`:

```
template <typename T>
remove_reference_t<T>&& move (T&& x) {
  return static_cast<remove_reference_t<T>&&>(x);
}
```

- Erläuterung:
  - Wenn `move` mit einem L-Wert eines Typs `U` aufgerufen wird, wird `T` gleich `U&` deduziert (vgl. § 5.4.2), sodass `remove_reference_t<T>` gleich `U` und der Resultattyp von `move` somit gleich `U&&` ist.
  - Beim Aufruf mit einem R-Wert eines Typs `U` wird `T` gleich `U` deduziert, sodass `remove_reference_t<T>` ebenfalls gleich `U` und der Resultattyp von `move` somit wiederum gleich `U&&` ist.
  - Also wandelt `move` den Wert `x` in jedem Fall in einen R-Wert um (vgl. § 3.5.3).
  - Damit der Wert `x` dabei nicht kopiert wird, muss sowohl der Parametertyp als auch der Resultattyp von `move` ein Referenztyp sein.

- Anmerkung: Die Schablonen `is_same`, `is_pointer`, `remove_reference` sowie viele weitere sind in der Standardbibliothek in der Datei `<type_traits>` definiert.

5.4.4 Gemeinsamer Typ mehrerer Typen

Resultattyp des Verzweigungsoperators

- Wenn der zweite und dritte Operand eines Ausdrucks `x ? y : z` unterschiedliche Typen besitzen, ist der Typ des Ausdrucks der „gemeinsame“ Typ der beiden Typen, sofern dieser nach bestimmten (im Detail komplizierten) Regeln existiert.

- Um diesen gemeinsamen Typ zu bestimmen, werden u. a. die üblichen arithmetischen Umwandlungen (vgl. § 2.1.8) oder Umwandlungen von Klassentypen und zugehörigen Zeiger- oder Referenztypen in mögliche Basis- (Zeiger- bzw. Referenz-) Typen angewandt.

- Wenn `B` und `C` beide von `A` abgeleitet sind, gilt zum Beispiel:

Typ 1	Typ 2	Gemeinsamer Typ
`short`	`int`	`int`
`short`	`unsigned short`	`int`
`int`	`unsigned int`	`unsigned int`
`int`	`double`	`double`
`B*`	`C*`	`A*`

Gemeinsamer Typ zweier Typen

- Dies kann wie folgt verwendet werden, um den gemeinsamen Typ `common_type_t<X, Y>` zweier Typen `X` und `Y` zu ermitteln (sofern er existiert):

```
template <typename X, typename Y>
struct common_type {
  using type = std::decay_t<
    decltype(true ? std::declval<X>() : std::declval<Y>())>;
};

template <typename X, typename Y>
using common_type_t = typename common_type<X, Y>::type;
```

- Erläuterungen:
  - Für einen Ausdruck `x` wird `decltype(x)` vom Übersetzer durch den Typ von `x` ersetzt. (`decltype` ist ein Schlüsselwort.)
    Der Ausdruck `x` wird zur Laufzeit nicht ausgewertet.
  - Für einen Typ `X` ist `std::declval<X>()` ein Ausdruck mit Typ `X&&`, der jedoch nur in Ausdrücken verwendet werden darf, die zur Laufzeit nicht ausgewertet werden. (`declval` ist eine Bibliotheksfunktion, die in `<utility>` definiert ist.)
    `declval` kann auch dann verwendet werden, wenn der Typ `X` keinen parameterlosen Konstruktor besitzt und der einfachere Ausdruck `X()` deshalb nicht verwendet werden kann.
  - `std::decay_t<X>` wendet auf den Typ `X` dieselben Standardumwandlungen an, die bei der Übergabe von Funktionsargumenten „by value“ angewandt werden, das heißt:
    - L-Werte werden in R-Werte umgewandelt.
    - Reihen und Funktionen werden in korrespondierende Zeiger umgewandelt.
    - `const`- und `volatile`-Qualifizierer auf oberster Ebene werden entfernt.

- Die entsprechende Definition von `common_type_t` in der Datei `<type_traits>` der Standardbibliothek kann auf beliebig viele Typen angewandt werden und bestimmt dann zunächst den gemeinsamen Typ des ersten und zweiten Typs, dann den gemeinsamen Typ dieses Typs und des dritten Typs usw.
  (Das kann in bestimmten Fällen aber dazu führen, dass ein gemeinsamer Typ aller Typen nicht gefunden wird oder dass nicht jeder der Typen in den ermittelten Typ umgewandelt werden kann.)

Anwendungsbeispiele

- Maximum zweier Werte (vgl. § 5.1.2):

```
// Maximum zweier Werte x und y
// mit eventuell verschiedenen Typen X und Y.
template <typename X, typename Y>
common_type_t<X, Y> maximum (X x, Y y) {
  return x > y ? x : y;
}
```

```
                          // Typ X    Typ Y        Resultattyp
maximum(3, 4)             // int      int          int
maximum(3.0, 4.0)         // double   double       double
maximum(3, 4.0)           // int      double       double
```

- Definition mit Einschränkungen:

```
template <typename X, typename Y>
concept GtCmpableWith = requires (X x, Y y) { bool(x > y); };

template <typename X, typename Y>
requires GtCmpableWith<X, Y> && common_with<X, Y>
common_type_t<X, Y> maximum (X x, Y y) {
  return x > y ? x : y;
}
```

- Die Einschränkung `common_with<X, Y>` (vgl. § 5.3.5) kann auch weggelassen werden, denn wenn die für `X` und `Y` deduzierten Typen keinen gemeinsamen Typ besitzen, scheitert ihre Einsetzung in den Resultattyp `common_type_t<X, Y>` (substitution failure), was – genauso wie eine nicht erfüllte Einschränkung – dazu führt, dass die Schablone nicht verwendet werden kann. Wenn es dann noch andere Funktionen oder Funktionsschablonen `maximum` gäbe, würde trotzdem noch überprüft werden, ob eine von ihnen verwendet werden kann, d. h. ein solches Scheitern führt nur zum Ausschluss der jeweiligen Schablone, aber nicht automatisch zu einem Übersetzungsfehler (substitution failure is not an error, kurz SFINAE).

- Die Einschränkung `GtCmpableWith<X, Y>` ist jedoch wichtig, damit die obige Funktionsschablone ausgeschlossen wird, wenn es keinen passenden Größer-Operator gibt, weil sonst ihre Expansion zu einem echten Fehler führen würde.

- Verkettung von Reihen:

```
// Verkettung der Reihen xs mit m Elementen des Typs X
// und ys mit n Elementen des Typs Y.
template <typename X, typename Y,
                              typename Z = common_type_t<X, Y>>
Z* concat (X* xs, int m, Y* ys, int n) {
  Z* zs = new Z [m + n];
  for (int i = 0; i < m; i++) zs[i] = xs[i];
  for (int i = 0; i < n; i++) zs[m+i] = ys[i];
  return zs;
}
```

Auch hier gilt: Wenn die für `X` und `Y` deduzierten Typen keinen gemeinsamen Typ besitzen, scheitert ihre Einsetzung in `common_type_t<X, Y>`, sodass auch hier keine zusätzliche Einschränkung `common_with<X, Y>` nötig ist.

5.4.5 Automatische Ermittlung von Typen

Typ eines Ausdrucks (vgl. § 5.4.4)

- Für einen beliebigen Ausdruck `x` wird `decltype(x)` vom Übersetzer durch den Typ von `x` ersetzt. Der Ausdruck `x` wird zur Laufzeit nicht ausgewertet.

Typ von Variablen

- Wenn das Schlüsselwort `auto` (optional umgeben von `const`, `volatile`, `&`, `&&` und beliebig oft `*`) als Typ einer Variablen verwendet wird, wird der Typ der Variablen, ähnlich wie bei template argument deduction, automatisch aus dem Typ ihrer Initialisierung ermittelt.

- Zum Beispiel:

```
auto x = 1;                         // x hat Typ int.
auto p = &x;                        // p hat Typ int*.
const auto& r = x;                  // r hat Typ const int&.
const auto * const q = p;           // q hat Typ const int * const.
```

Strukturzerlegungen (structured bindings)

- Wenn dem Schlüsselwort `auto` (optional umgeben von `const`, `volatile`, `&` und `&&`) eine Liste von Variablennamen in eckigen Klammern folgt, wird das Objekt, das sich aus der nachfolgenden Initialisierung ergibt, in seine Bestandteile zerlegt, die dann der Reihe nach zur Initialisierung der Variablen verwendet werden. Dabei kann sich für jede Variable ein anderer Typ ergeben.

- Zur Initialisierung können dabei folgende Arten von Objekten `x` verwendet werden:
  - Eine Reihe, deren Bestandteile ihre Elemente `x[i]` sind.
  - Ein Tupel oder ähnliches, dessen Bestandteile durch Funktionsaufrufe `x.get<i>()` oder `get<i>(x)` mit `i` = 0, 1, ... ermittelt werden.
  - Ein sonstiges Strukturobjekt, dessen Bestandteile seine Elementvariablen sind.

  Die Anzahl der Bestandteile des Objekts `x` muss in jedem Fall mit der Anzahl der Variablen übereinstimmen. Für Tupel-artige Objekte ermittelt der Übersetzer diese Anzahl mittels `std::tuple_size<X>::value`, wobei `X` der Typ des Objekts `x` ist.

- Zum Beispiel:

```
int a [] = { 1, 2, 3 };
auto& [a1, a2, a3] = a;
// a1, a2, a3 sind Referenzen auf a[0], a[1], a[2].

// Alle Schlüssel-Wert-Paare [k, v] der Tabelle tab durchlaufen.
map<Key, Value> tab;
for (auto [k, v] : tab) {
  ......
}
```

- Geschachtelte Strukturen müssen schrittweise zerlegt werden, zum Beispiel:

```
map<Key, pair<Value1, Value2>> tab;
for (auto [k, v] : tab) {
  auto [v1, v2] = v;
  ......
}
```

Resultattypen

- Wenn das Schlüsselwort `auto` als Resultattyp einer Funktion verwendet wird, kann der tatsächliche Resultattyp der Funktion entweder nach der Parameterliste angegeben werden (trailing return type), oder er wird automatisch aus den `return`-Anweisungen im Rumpf der Funktion ermittelt (dann kann `auto` auch wieder optional von `const`, `volatile`, `&`, `&&` und beliebig oft `*` umgeben sein).
- Zum Beispiel:

```
// Quadrat eines Werts x mit beliebigem Typ X.
template <typename X>
auto square (X x) -> decltype(x * x) {
  return x * x;
}

// Maximum zweier Werte x und y
// mit eventuell verschiedenen Typen X und Y.
template <typename X, typename Y>
auto maximum (X x, Y y) {
  return x > y ? x : y;
}
```

- Wenn der Resultattyp nicht angegeben ist, sind rekursive Aufrufe der Funktion vor der ersten `return`-Anweisung nicht möglich.

Anmerkungen

- Die Verwendung von `auto` ist praktisch, wenn ein Typ nicht (genau) bekannt oder mühsam hinzuschreiben ist. Sinnvoll eingesetzt, können Programme dadurch einfacher und übersichtlicher werden.

- Umgekehrt geht dadurch natürlich auch Information verloren, die für das Verständnis des Codes nützlich sein kann.

- Vor C++11 hatte das Schlüsselwort `auto` eine vollkommen andere, aber letztlich überflüssige Bedeutung, um eine lokale Variable ausdrücklich als nicht `static` zu kennzeichnen, was sie ohne diese Kennzeichnung aber sowieso ist.

Abgekürzte Funktionsschablonen (seit C++20):

- Wenn das Schlüsselwort `auto` (optional umgeben von `const`, `volatile`, `&`, `&&` und beliebig oft `*`) als Typ eines Funktionsparameters verwendet wird, wird es durch einen eindeutigen Schablonenparameter `T` ersetzt und `typename T` am Ende der Schablonenparameterliste hinzugefügt.

- Wenn vor dem Schlüsselwort `auto` eine Typeinschränkung steht, kann die Schablone nur verwendet werden, wenn diese Einschränkung erfüllt ist (vgl. § 5.3.8).

- Wenn die Funktion ursprünglich keine Funktionsschablone ist, wird sie dadurch automatisch zu einer Schablone mit einer ursprünglich leeren Schablonenparameterliste.

- Wenn `auto` mehrmals in der Funktionsparameterliste vorkommt, wird für jedes Vorkommen ein separater Schablonenparameter eingesetzt und zur Schablonenparameterliste hinzugefügt.

- Zum Beispiel:

```
auto maximum (auto x, auto y) { return x > y ? x : y; }

auto square (const Multable auto& x) { return x * x; }
```

5.4.6 Funktionsobjekte und Lambda-Ausdrücke

Funktionsobjekte

- Wenn eine Klasse eine (oder mehrere) Elementfunktion(en) `operator()` definiert, können ihre Objekte wie Funktionen verwendet werden, zum Beispiel:

```
using str = const char*;

struct StrHash {
  // Streuwert der Zeichenkette s wie bei
  // java.lang.String.hashCode berechnen.
  // (Beachte: Für vorzeichenlose Typen wie size_t
  // ist arithmetischer Überlauf wohldefiniert.)
  size_t operator() (str s) const {
    size_t h = 0;
    while (char c = *s++) h = h * 31 + c;
    return h;
  }
};

// Definition und Verwendung des Funktionsobjekts h.
StrHash h;
int i = h("abc"); // Bedeutet eigentlich: h.operator()("abc")
```

- Ein Vorteil gegenüber normalen Funktionen besteht darin, dass ein Funktionsobjekt zusätzliche Daten speichern kann, die in der Elementfunktion `operator()` verwendet werden können, zum Beispiel:

```
struct StrHash {
  // Bei der Berechnung des Streuwerts verwendeter Faktor.
  size_t factor;

  // Faktor mit f initialisieren.
  explicit StrHash (size_t f = 31) : factor{f} {}

  // Streuwert der Zeichenkette s ähnlich wie
  // bei java.lang.String.hashCode berechnen.
  size_t operator() (str s) const {
    size_t h = 0;
    while (char c = *s++) h = h * factor + c;
    return h;
  }
};

// Funktionsobjekte h31 und h47 mit Faktor 31 bzw. 47 erzeugen
// und verwenden.
StrHash h31;     int i = h31("abc");
StrHash h47{47}; int j = h47("abc");
```

Simulation lokaler Funktionen

- Da Klassen auch lokal in einer Funktion definiert werden können, lassen sich damit prinzipiell lokale Funktionen simulieren, zum Beispiel:

```
// Reihe a der Größe m mit Haldensortierung sortieren.
template <typename T>
void heapsort (T* a, int m) {
  // Funktionsobjekt sink zum Absenken des Elements i in
  struct {                  // der Maximum-Halde a mit Größe m.
    void operator() (T* a, int m, int i) { ..... }
  } sink;

  // Reihe a durch Absenken von Elementen
  // in eine Maximum-Halde umformen.
  for (int i = m/2 - 1; i >= 0; i--) sink(a, m, i);

  while (m > 1) {           // Solange die Halde nicht leer ist:
    swap(a[0], a[m-1]); // Erstes und letztes Element der Halde
    m--;                    // vertauschen, letztes Element abtrennen
    sink(a, m, 0);          // und erstes Element passend absenken.
  }
}
```

- Da lokale Klassen nicht auf die Parameter und lokalen Variablen der umschließenden Funktion zugreifen dürfen, müssen diese bei Bedarf als Parameter an das Funktionsobjekt übergeben werden (wie zuvor) oder über den Konstruktor in die Klasse „geschleust“ werden, entweder als Kopien oder als Referenzen, zum Beispiel:

```
template <typename T>
void heapsort (T* a, int m) {
  // Funktionsobjekt sink mit Zugriff auf die Parameter a (Kopie)
  struct Sink {  // und m (Referenz) der umschließenden Funktion.
    T* a; int& m;
    Sink (T* a, int& m) : a{a}, m{m} {}
    void operator() (int i) { ..... }
  } sink{a, m};

  // Reihe a durch Absenken von Elementen
  // in eine Maximum-Halde umformen.
  for (int i = m/2 - 1; i >= 0; i--) sink(i);

  while (m > 1) {          // Solange die Halde nicht leer ist:
    swap(a[0], a[m-1]); // Erstes und letztes Element der Halde
    m--;                  // vertauschen, letztes Element abtrennen
    sink(0);              // und erstes Element passend absenken.
  }
}
```

Lambda-Ausdrücke

- Lambda-Ausdrücke sind nichts anderes als angenehme syntaktische Verpackungen solcher Funktionsobjekte, zum Beispiel:

```
// Reihe a der Größe n mit Haldensortierung sortieren.
template <typename T>
void heapsort (T* a, int m) {
  // Initialisierung des Funktionsobjekts sink
  // durch einen Lambda-Ausdruck.
  auto sink = [a, &m] (int i) { ..... };

  // Reihe a durch Absenken von Elementen
  // in eine Maximum-Halde umformen.
  for (int i = m/2 - 1; i >= 0; i--) sink(i);

  while (m > 1) {          // Solange die Halde nicht leer ist:
    swap(a[0], a[m-1]); // Erstes und letztes Element der Halde
    m--;                  // vertauschen, letztes Element abtrennen
    sink(0);              // und erstes Element passend absenken.
  }
}
```

- Hier ist `sink` ein Objekt einer anonymen Klasse
  - mit Elementvariablen `T* a` und `int& m`,
  - einem Konstruktor mit Parametern `T* a` und `int& m`, der diese Elementvariablen initialisiert,
  - einer Elementfunktion `operator()` mit Parameter `int i`, Resultattyp `auto` und der in geschweiften Klammern angegebenen Implementierung,

  dessen Konstruktor die Parameter `a` und `m` von `heapsort` übergeben werden.

Anmerkungen

- Wenn die Parameterliste eines Lambda-Ausdrucks leer ist, können die runden Klammern auch weggelassen werden.
- Wie bei einer normalen Funktion mit Resultattyp `auto`, kann der tatsächliche Resultattyp optional als „trailing return type“ (vgl. § 5.4.5) vor den geschweiften Klammern angegeben werden.
- Anders als bei einer expliziten lokalen Klasse, bezeichnet `this` innerhalb eines Lambda-Ausdrucks das aktuelle Objekt der umschließenden Klasse und nicht das Lambda- bzw. Funktionsobjekt.
- Die Bezeichnung Lambda-Ausdruck wurde in Anlehnung an den Lambda-Kalkül von Church und Kleene gewählt.

Variablenbindungen (captures)

- Innerhalb der eckigen Klammern sind u. a. folgende Angaben möglich:
  - = bzw. `&` am Anfang drückt aus, dass innerhalb des Lambda-Ausdrucks alle lokalen Variablen und Parameter der umschließenden Gültigkeitsbereiche, für die anschließend keine abweichenden Angaben gemacht werden, als Kopien bzw. Referenzen zugreifbar sind.
  - `x` bzw. `&x` drückt aus, dass die Variable oder der Parameter `x` als Kopie bzw. Referenz zugreifbar ist.
    Sofern = bzw. `&` am Anfang angegeben wurde, können anschließend nur noch davon abweichende Angaben gemacht werden, d. h. Angaben der Form `&x` bzw. `x`.
  - `*this` bzw. `this` drückt aus, dass das aktuelle Objekt der umschließenden Klasse – und damit auch dessen Elementvariablen – als Kopien bzw. Referenzen zugreifbar sind.
    Die Angaben = oder `&` am Anfang implizieren beide `this`, sofern anschließend nicht explizit `*this` angegeben wird.
- Wenn Variablen per Referenz gebunden werden, darf der Lambda-Ausdruck nur verwendet werden, solange die gebundenen Variablen existieren.
- Die eckigen Klammern müssen immer angegeben werden, selbst wenn sie leer sind.

Generische Lambda-Ausdrücke

- Wenn das Schlüsselwort `auto` wie bei abgekürzten Funktionsschablonen (vgl. § 5.4.5) in der Parameterliste eines Lambda-Ausdrucks verwendet wird, ist die Elementfunktion `operator()` des Lambda-Objekts eine entsprechende Funktionsschablone, zum Beispiel:

```
auto max = [] (auto x, auto y) { return x > y ? x : y; }
```

  Das Lambda-Objekt `max` kann prinzipiell mit zwei beliebigen Parametern `x` und `y` aufgerufen werden, sofern diese miteinander verglichen werden können und einen gemeinsamen Typ besitzen (vgl. § 5.4.4).

- Durch Kombination von `auto` mit `...` (vgl. § 5.5.1) können auch variadische Lambda-Ausdrücke definiert werden, zum Beispiel:

```
auto error = [] (auto ... xx) {
  cout << "Error:";
  (..., (cout << " " << xx));
  cout << endl;
};
```

  Das Lambda-Objekt `error` kann prinzipiell mit beliebig vielen Parametern `xx` aufgerufen werden, sofern diese auf `cout` ausgegeben werden können.

- Seit C++20 können Lambda-Ausdrücke auch eine explizite Schablonenparameterliste nach den eckigen Klammern sowie eine Anforderungsklausel nach der Schablonenparameterliste und/oder nach der Funktionsparameterliste besitzen, zum Beispiel:

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

- Anmerkung: Normale lokale Klassen dürfen keine Schablonen enthalten.

Rekursive Lambda-Ausdrücke

- Da die von einem Lambda-Ausdruck definierte Funktion an sich keinen Namen besitzt, kann sie sich nicht direkt rekursiv aufrufen.

- Wenn der Lambda-Ausdruck wie üblich zur Initialisierung einer Variablen mit Typ `auto` verwendet wird, darf diese Variable innerhalb des Lambda-Ausdrucks ebenfalls nicht verwendet werden, weil ihr tatsächlicher Typ dort noch nicht bekannt ist.

- Wenn der Lambda-Ausdruck jedoch zur Initialisierung einer Variablen mit Typ `std::function<R (PP ...)>` verwendet wird, kann diese Variable (ggf. mit geeigneter Bindung) innerhalb des Lambda-Ausdrucks für rekursive Aufrufe verwendet werden, weil ihr Typ bereits bekannt ist, zum Beispiel:

```
struct Tree {   // Binärer Baum.
  int elem;     // Im Wurzelknoten gespeichertes Element.
  Tree* left;   // Linker ...
  Tree* right;  // ... und rechter Teilbaum (oder nullptr).
};
                // Element x im Baum t suchen.
function<bool (Tree*, int)> search = [search] (Tree* t, int x) {
  if (!t) return false;
  if (t->elem == x) return true;
  return search(t->left, x) || search(t->right, x);
};
bool found = search(t, x);
```

- Alternativ kann folgender Trick verwendet werden, bei dem man das Funktionsobjekt `search` jeweils als zusätzlichen Parameter an seine eigenen Aufrufe übergeben muss (was man auch in einem weiteren Funktionsobjekt „verstecken“ könnte):

```
auto search = [] (Tree* t, int x, auto search) {
  if (!t) return false;
  if (t->elem == x) return true;
  return search(t->left, x, search) ||
                                   search(t->right, x, search);
};
bool found = search(t, x, search);
```