5.5 Variadische Schablonen

5.5.1 Variadische Funktionen

Ausgabe beliebig vieler Werte beliebiger Typen

```
// Gewöhnliche Funktionsschablone zur Ausgabe eines Werts.
template <typename T>
void print (T x) {
  cout << x << endl;
}

// Variadische Funktionsschablone zur Ausg. beliebig vieler Werte.
template <typename T, typename ... TT>
void print (T x, TT ... xx) {
  cout << x << ",";
  print(xx ...);
}

// Verwendungsmöglichkeiten.
print(1);
print(1, "abc");
print(1, "abc", true);
```

Erläuterungen

- `typename ... TT` deklariert `TT` als *Schablonenparameterbündel* (template parameter pack), d. h. als Folge beliebig vieler (null oder mehr) Schablonenparameter.

- `TT ... xx` deklariert `xx` als *Funktionsparameterbündel* (function parameter pack), d. h. als Folge entsprechend vieler Funktionsparameter, deren Typen den Schablonenparametern des Bündels `TT` entsprechen.

- `xx ...` ist eine *Bündelexpansion* (pack expansion), die durch die Folge der Funktionsparameter des Bündels `xx` ersetzt wird.

- Allgemeiner kann eine Bündelexpansion z. B auch ein Ausdruck, der ein Bündel enthält, gefolgt von `...` sein, z. B. `xx + 1 ...` oder `f(xx) ...` o. ä.
(Beachte aber den Unterschied zwischen `f(xx ...)` und `f(xx) ...`!)

- Die *variadische* Funktionsschablone `print` kann mit einem Parameter `x` mit beliebigem Typ `T` sowie beliebig vielen weiteren Parametern `xx` mit beliebigen Typen `TT` aufgerufen werden.
- Bei einem Aufruf mit genau einem Parameter wird jedoch die gewöhnliche (nicht-variadische) Funktionsschablone bevorzugt, die den Wert dieses Parameters und einen abschließenden Zeilentrenner auf `cout` ausgibt.
- Die variadische Funktion gibt den Wert ihres ersten Parameters `x` und ein nachfolgendes Komma aus; anschließend wird `print` rekursiv für `xx ...` aufgerufen.
- Wenn das Bündel `xx` genau einen Parameter enthält, wird hierbei wiederum die gewöhnliche Funktionsschablone aufgerufen und somit die Rekursion beendet.
- Andernfalls wird erneut die variadische Funktion wie folgt aufgerufen:
  - Ihr Parameter `x` wird mit dem ersten Parameter des aktuellen Bündels `xx` belegt.
  - Ihr Parameterbündel `xx` wird mit den restlichen Parametern des aktuellen Bündels `xx` belegt.
- Wenn es die gewöhnliche Funktionsschablone nicht gäbe, könnte die variadische Funktion prinzipiell auch mit genau einem Parameter aufgerufen werden; das Bündel `xx` wäre dann leer.
  Aber in diesem Fall würde der darin enthaltene rekursive Aufruf `print (xx ...)` zu einem Übersetzungsfehler führen, weil es keine parameterlose Funktion `print` gibt.

Summe und Durchschnittswert beliebig vieler Zahlen

```
// Leere Summe.
double sum () {
  return 0;
}

// Summe eines oder mehrerer Werte.
template <typename ... TT>
double sum (double x, TT ... xx) {
  return x + sum(xx ...);
}

// Durchschnitt eines oder mehrerer Werte.
template <typename ... TT>
double avg (double x, TT ... xx) {
  return sum(x, xx ...) / (1 + sizeof...(xx));
}
```

Erläuterungen

- Dass die Parameter von `sum` und `avg` alle Typ `double` haben sollen, lässt sich vor C++20 nicht direkt ausdrücken.

- Formal kann die Funktionsschablone `sum` mit einem Parameter `x` des Typs `double` und beliebig vielen weiteren Parametern `xx` mit beliebigen Typen `TT` aufgerufen werden.

- Aber weil jeder dieser weiteren Parameter durch die rekursive Definition von `sum` früher oder später an den Parameter `x` übergeben wird, müssen faktisch doch alle Parameter einen mit `double` kompatiblen Typ besitzen.

- Und da die Parameter von `avg` wiederum an `sum` übergeben werden, müssen auch ihre Typen alle mit `double` kompatibel sein.

- Selbst wenn `sum` niemals direkt ohne Parameter aufgerufen wird, wird die parameterlose Funktion benötigt, damit der rekursive Aufruf `sum(xx ...)` in der variadischen Funktion für ein leeres Bündel `xx` definiert ist.

- Da der Durchschnittswert von 0 Werten zu einer Division durch 0 führen würde, ist `avg` so definiert, dass es mit mindestens einem Wert aufgerufen werden muss.

- `sizeof...` liefert die Größe eines (Schablonen- oder Funktions-) Parameterbündels, d. h. die Anzahl seiner Parameter.

- Anmerkung: Alternativ könnte man `sum` und `avg` als gewöhnliche Funktionen mit „Initialisiererlisten“ (vgl. § 7.3) definieren.

Definition mit Einschränkungen

```
// Eigenschaft: T muss implizit in double umwandelbar sein.
template <typename T>
concept Double = convertible_to<T, double>;

// Leere Summe.
double sum () {
  return 0;
}

// Summe eines oder mehrerer Werte
// als normale Funktionsschablone mit Einschränkung.
template <Double ... TT>
double sum (double x, TT ... xx) {
  return x + sum(xx ...);
}

// Durchschnitt eines oder mehrerer Werte
// als abgekürzte Funktionsschablone mit Einschränkung, vgl. §5.4.5.
double avg (double x, Double auto ... xx) {
  return sum(x, xx ...) / (1 + sizeof...(xx));
}
```

5.5.2 Faltungen (fold expressions)

- Seit C++17 können in einer variadischen Schablone folgende Faltungsausdrücke verwendet werden, die in jeder Ausprägung der Schablone durch die entsprechende Bedeutung ersetzt werden:

Faltungsausdruck	Bedeutung	Bezeichnung
`(... ○ XX)`	$((X_1 \circ X_2) \circ \ldots) \circ X_N$	unäre Linksfaltung
`(XX ○ ...)`	$(X_1 \circ (\ldots \circ (X_{N-1} \circ X_N)))$	unäre Rechtsfaltung
`(X ○ ... ○ XX)`	$(((X \circ X_1) \circ X_2) \circ \ldots) \circ X_N$	binäre Linksfaltung
`(XX ○ ... ○ X)`	$(X_1 \circ (\ldots \circ (X_{N-1} \circ (X_N \circ X))))$	binäre Rechtsfaltung

- Die in der Tabelle verwendeten Symbole haben dabei folgende Bedeutung:
  - ○ ist (auch bei unären Faltungen) ein beliebiger binärer Operator, z. B. `+`, `/`, `=` oder `&&`.
  - `XX` ist ein Ausdruck, der ein Bündel `xx` der Größe N und auf oberster Ebene keinen binären Operator enthält, z. B. `xx`, `f(xx)`, `*xx`, `(2*xx)`, aber nicht `2*xx` ohne Klammern.
  - Für $i = 1, \ldots, N$ entsteht X_i aus dem Ausdruck `XX`, indem das Bündel `xx` durch den i-ten Parameter des Bündels ersetzt wird.
  - `X` ist ein Ausdruck, der kein Bündel und auf oberster Ebene keinen binären Operator enthält.

- In der ersten Spalte der Tabelle sind die Punkte `...` wie bei einer Bündeldeklaration oder -expansion „wörtlich“ gemeint, in der zweiten Spalte handelt es sich jedoch um Auslassungspunkte mit der Bedeutung „und so weiter“.
- Ein Faltungsausdruck muss immer von Klammern umgeben sein.
- Bei einer binären Faltung mit einem leeren Bündel (d. h. $N = 0$) entsteht jeweils der Ausdruck `(X)`.
- Bei einer unären Faltung mit einem leeren Bündel entstehen folgende Ausdrücke:
  - `true`, wenn der Operator $\circ$ gleich `&&` ist (eine leere Konjunktion mit Wert `true`);
  - `false`, wenn der Operator $\circ$ gleich `||` ist (eine leere Disjunktion mit Wert `false`);
  - `void()`, d. h. ein Ausdruck mit Typ `void`, wenn der Operator $\circ$ gleich Komma ist (eine leere Aneinanderreihung von Ausdrücken ohne Wert).

  Andere Operatoren führen hier bei einem leeren Bündel zu einem Übersetzungsfehler.
- Mit Hilfe von Faltungsausdrücken lassen sich variadische Funktionen häufig einfacher und ohne Rekursion formulieren.

Beispiele

```
// Ausgabe beliebig vieler Werte.
// Bei einem Aufruf ohne Parameter wird nur endl ausgegeben.
template <typename ... TT>
void print (TT ... xx) {
    (cout << ... << xx) << endl;
}

// Summe eines oder mehrerer Werte.
// Übersetzungsfehler bei einem Aufruf ohne Parameter.
template <typename ... TT>
double sum (TT ... xx) {
    return (... + xx);
}

// Definition mit Einschränkungen.
template <Double ... TT>
requires (sizeof...(TT) > 0)
double sum (TT ... xx) {
    return (... + xx);
}
```

5.5.3 Unverfälschte Weitergabe von Funktionsargumenten (perfect forwarding)

- Beispiel:

```
// Funktion f mit beliebigen Parameterwerten xx n-mal ausführen.
template <typename R, typename ... PP>
void repeat (int n, R f (PP ...), PP ... xx) {
  for (int i = 0; i < n; i++) f(xx ...);
}

// Verwendungsmöglichkeit.
void print_int (int x) { cout << x << endl; }
print_int(1);
repeat(10, print_int, 1);
```

- Ähnliches Beispiel aus der Standardbibliothek:
  Der Konstruktor der Klasse `thread` erhält als Parameter ebenfalls eine beliebige Funktion `f` und zugehörige Parameterwerte `xx` und führt den Funktionsaufruf `f(xx ...)` in einem neuen Thread aus.

- Erstes Problem:

```
void print_str (const string& s) { cout << s << endl; }
print_str("Hallo");                 // OK.
repeat(10, print_str, "Hallo"); // Fehler.
```

  - Beim Aufruf von `repeat` wird `PP` aus dem Typ `void (const string&)` von `print_str` als `const string&` deduziert.
  - Aus dem Typ `const char*` von `"Hallo"` wird jedoch `PP` gleich `const char*` deduziert.
  - Damit ist der Aufruf von `repat` fehlerhaft, obwohl der direkte Aufruf von `print_str` korrekt ist.

- Lösungsmöglichkeit:

```
template <typename R, typename ... PP1, typename ... PP2>
void repeat (int n, R f (PP1 ...), PP2 ... xx) {
  for (int i = 0; i < n; i++) f(xx ...);
}
```

  - Die Parametertypen `PP1` der Funktion `f` und die Typen `PP2` der Argumente `xx` können (prinzipiell beliebig) verschieden sein.
  - Der Aufruf von `f` gelingt aber nur, wenn die Typen `PP2` kompatibel zu den Typen `PP1` sind.

- Noch allgemeiner:

```
template <typename F, typename ... PP>
void repeat (int n, F f, PP ... xx) {
  for (int i = 0; i < n; i++) f(xx ...);
}
```

  - `f` kann jetzt etwas prinzipiell Beliebiges sein, für das `f(xx ...)` korrekt ist.
  - Neben „echten“ Funktionen, können nun auch Funktionsobjekte (und insbesondere Lambda-Ausdrücke, vgl. § 5.4.6) übergeben werden.

- Beispiel mit Lambda-Ausdrücken:

```
repeat(10,
  [] (const string& s) { cout << s << endl; },
  "Hallo");

// Oder kürzer:
repeat(10, [] { cout << "Hallo" << endl; });
```

- Zweites Problem:

```
int n = 0;
repeat(10, [] (int& x) { x++; }, n);
cout << n << endl;        // Ausgabe: 0
```

  - Der L-Wert `n` wird bei der Übergabe an `repeat` automatisch in einen R-Wert umgewandelt (vgl. § 5.4.2).
  - Damit erhöht die an `repeat` übergebene Funktion nicht die Variable `n`, sondern den Parameter (des Bündels) `xx` von `repeat`, der mit dem Wert von `n` initialisiert wurde.

- Lösung:

```
template <typename F, typename ... PP>
void repeat (int n, F f, PP&& ... xx) {
  for (int i = 0; i < n; i++) f(xx ...);
}
```

  - Durch die Verwendung „universeller“ Referenzen `PP&&` (vgl. ebenfalls § 5.4.2), werden L- und R-Werte jeweils unverfälscht an `repeat` übergeben.

- Drittes Problem:

```
void print_str_r (string&& s) { cout << s << endl; }
print_str_r(string("Hallo"));              // OK.
repeat(10, print_str_r, string("Hallo")); // Fehler.
```

  - Der direkte Aufruf von `print_str_r` ist korrekt, weil `string("Hallo")` ein R-Wert ist, der somit an den R-Wert-Referenz-Parameter `s` übergeben werden kann.
  - Der indirekte Aufruf von `print_str_r` in `repeat` ist jedoch fehlerhaft, weil der Parameter (des Bündels) `xx` immer ein L-Wert ist, selbst wenn sein Typ eine R-Wert-Referenz ist (vgl. § 2.9.5).

❒ Lösung:

```
template <typename F, typename ... PP>
void repeat (int n, F f, PP&& ... xx) {
  for (int i = 0; i < n; i++) f(static_cast<PP&&>(xx) ...);
}
```

❒ Oder besser:

```
template <typename F, typename ... PP>
void repeat (int n, F f, PP&& ... xx) {
  for (int i = 0; i < n; i++) f(std::forward<PP>(xx) ...);
}
```

- Erläuterungen
  - Obwohl der `static_cast` „idempotent“ ist (der Typ von `xx` stimmt bereits mit dem Zieltyp `PP&&` überein), ist der Ausdruck `static_cast<PP&&>(xx)` kein Parameter mehr und deshalb nur noch dann ein L-Wert, wenn `PP` ein L-Wert-Referenztyp ist.
  - Die Bibliotheksfunktion `forward` ist äquivalent zu diesem `static_cast`.
  - Im Gegensatz zur Funktion `move` (vgl. § 3.5.3), die eine ähnliche Typumwandlung vornimmt, muss bei der Verwendung von `forward` der Basistyp der „universellen“ Referenz (hier `PP`) explizit übergeben werden, weil nur er die Information enthält, ob der als Parameter übergebene Wert ursprünglich ein L- oder ein R-Wert ist.
  - Ein weiterer Unterschied zwischen `forward` und `move` besteht darin, dass das Resultat von `move` immer ein R-Wert ist, während das Resultat von `forward` ein L- oder ein R-Wert sein kann.
  - Diese Definition von `repeat` gibt nun beliebige Funktionsargumente `xx` vollkommen unverfälscht an die Funktion `f` weiter.
  - Insbesondere werden bei dem indirekten Aufruf von `f` in `repeat` für jedes Argument exakt die gleichen Kopier- oder Verschiebekonstruktoren aufgerufen wie bei einem direkten Aufruf der Funktion.

- Endgültige Lösung:

```
template <typename F, typename ... PP>
requires std::invocable<F, PP ...>
void repeat (int n, F&& f, PP&& ... xx) {
  for (int i = 0; i < n; i++) {
    std::forward<F>(f)(std::forward<PP>(xx) ...);
  }
}
```

  - Weil `f` nicht nur eine Funktion (bzw. ein Funktionszeiger), sondern auch ein Funktionsobjekt sein kann, sollte es vorsichtshalber ebenfalls als „universelle“ Referenz übergeben und seine Verwendung mit `forward` geklammert werden.
  - Das Übergeben per Referenz vermeidet eventuelles unerwünschtes Kopieren oder Verschieben.
  - Das Klammern mit `forward` vermeidet eine eventuelle Verfälschung des L- oder R-Wert-Status, der sich prinzipiell auf die Korrektheit oder die Bedeutung des Aufrufs von `f` auswirken könnte. (Die Elementfunktion `operator()` von `f` könnte mit einem sog. „ref-qualifier“ so definiert sein, dass sie nur auf einen L-Wert oder nur auf einen R-Wert angewandt werden kann. Oder `f` könnte zwei solche Elementfunktionen besitzen, die sich nur in diesem Detail unterscheiden.)
  - Die optionale Einschränkung mit `requires` bringt explizit zum Ausdruck, dass `f` mit den Argumenten `xx` aufgerufen werden kann.

5.5.4 Variadische Typen

- Generischer Tupeltyp (ähnlich zu `std::tuple`):

```
// Allgemeine Schablone mit beliebig vielen Typparametern.
template <typename ... TT> struct Tuple;

// Spezialisierung für null Typparameter.
template <> struct Tuple <> {};

// Spezialisierung für mindestens einen Typparameter.
template <typename T, typename ... TT>
struct Tuple <T, TT ...> {
  // Erstes Element und Resttupel mit ggf. weiteren Elementen.
  T head;
  Tuple<TT ...> tail;

  // Konstruktion aus erstem Element head und Resttupel tail.
  Tuple (T head, Tuple<TT ...> tail) : head{head}, tail{tail} {}

  // Konstruktion aus beliebig vielen Werten x, xx ...
  Tuple (T x, TT ... xx) : head{x}, tail{xx ...} {}
};
```

- Erzeugungsfunktionen zur bequemeren Verwendung:

```
// Konstruktion aus erstem Element head und Resttupel tail.
template <typename T, typename ... TT>
auto cons (T head, Tuple<TT ...> tail = Tuple<>()) {
  return Tuple<T, TT ...>{head, tail};
}

// Konstruktion aus beliebig vielen Werten xx ...
template <typename ... TT>
auto mktuple (TT ... xx) {
  return Tuple<TT ...>{xx ...};
}
```

- Verwendungsmöglichkeiten:

```
auto t1 = mktuple(1, "abc", true);
cout << t1.head << t1.tail.head << t1.tail.tail.head << endl;

auto t2 = cons(1, cons("abc", cons(true)));
cout << t2.head << t2.tail.head << t2.tail.tail.head << endl;
```