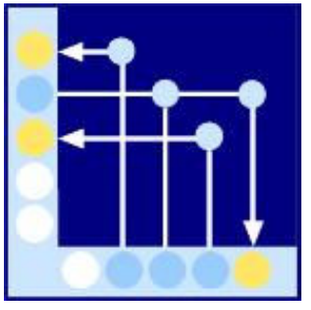

*Fakultät Elektronik und Informatik*
*Studienbereich Informatik*



# Programmieren in C++

Vorlesung im Wintersemester 2024/2025

*Prof. Dr. habil. Christian Heinlein*

christian.heinleins.net

# 1 Einleitung

## 1.1 Vorlesungsüberblick

### Ziele

- Vermittlung wesentlicher Konzepte von „modernem“ C++
- Schwerpunkt auf den „Besonderheiten“ der Sprache gegenüber anderen Programmiersprachen

## Form der Wissensvermittlung

- Vorlesung
  - Folien auf der Vorlesungs-Webseite
  - In der Regel kapitelweise, normalerweise vor Beginn des jeweiligen Kapitels
  - Bei Bedarf nachträgliche Ergänzungen oder Korrekturen
- Übung
  - Regelmäßige Aufgaben als Anregung zum eigenen Programmieren
  - Teilweise Besprechung in der Vorlesung
  - Musterlösung auf der Vorlesungs-Webseite
- Selbststudium

## Geplante Themen

- Grundlegende Datentypen, Operatoren und Anweisungen
- Klassen, einfache und mehrfache Vererbung, dynamisches Binden
- Konstruktoren, Destruktoren, Kopieren und Verschieben von Objekten
- Überladen von Funktionen und Operatoren
- Typ- und Funktionsschablonen (templates) inklusive Typkategorien (concepts)
- Funktionsobjekte, Lambda-Ausdrücke
- Container und Iteratoren
- Module

## Zugangsvoraussetzungen

- Zwingende Voraussetzung für die Zulassung zur Prüfung sind bestandene Prüfungen in Strukturierter und Objektorientierter Programmierung (Modul IN-57004 oder DS-43004).

# 1.2 Kurze Geschichte von C++

- 1979 Bjarne Stroustrup entwickelt „C with classes“
- 1984 Umbenennung in C++
- 1998 Erster Standard ISO/IEC 14882 (C++98)
- 2003 Überarbeitung/Korrektur des Standards von 1998 (C++03)
- 2011 Neuer Standard mit sehr vielen Erweiterungen
  (C++11, seit 2002 als C++0x bezeichnet, weil Fertigstellung vor 2010 erwartet wurde)
  Ab hier spricht man üblicherweise von „modernem“ C++.
- 2014 Kleine Erweiterung gegenüber C++11
  (C++14, zuvor als C++1y bezeichnet)
- 2017 Mittelgroße Erweiterung gegenüber C++14
  (C++17, zuvor als C++1z bezeichnet)
- 2020 Mittelgroße Erweiterung gegenüber C++17
  (C++20, zuvor als C++2a bezeichnet)
  Grundlage für diese Vorlesung
- 2023 Der nächste (noch nicht offiziell verabschiedete) Standard
  (C++23, vorläufig als C++2b bezeichnet)

# 1.3 Hallo, Welt!

```
// Definitionsdateien der C++-Standardbibliothek haben kein Suffix.
#include <iostream>

// Namensbereich std einbinden,
// in dem sich alle Namen der C++-Standardbibliothek befinden.
using namespace std;

// main hat Resultattyp int.
// Der Resultatwert wird als Exitstatus des Programms
// an das Betriebssystem zurückgeliefert.
// Parameter argc (Anzahl der Kommandozeilenargumente)
// und argv (Reihe mit den Kommandozeilenargumenten)
// sind optional.
int main (int argc, char* argv []) {
  // cout ist der Standardausgabestrom.
  // endl ist ein Zeilentrenner.
  cout << "Hallo, Welt!" << endl;
}
// Wenn main keine return-Anweisung ausführt, ist der Exitstatus 0.
// (Für andere Funktionen mit Resultattyp ungleich void wäre das
// Verhalten undefiniert.)
```

# 1.4 Bekannte C++-Übersetzer

## 1.4.1 GCC (GNU Compiler Collection)

- Je nach Version, werden unterschiedliche C++-Standards unterstützt.
- Seit Version 12 wird C++20 größtenteils unterstützt.
  (Vor allem die Unterstützung für Module ist noch unvollständig.)
- Seit Version 15 werden auch bereits große Teile von C++23 unterstützt.
- Siehe auch https://gcc.gnu.org/projects/cxx-status.html

## 1.4.2 Clang (Frontend für LLVM)

- Auch hier werden ja nach Version unterschiedliche C++-Standards unterstützt.
- Seit Version 19 wird C++20 größtenteils unterstützt.
  (Auch hier ist vor allem die Unterstützung für Module noch unvollständig.)
- Seit Version 19 werden auch bereits große Teile von C++23 unterstützt.
- Siehe auch https://clang.llvm.org/cxx_status.html

## 1.4.3 Wichtige Aufrufoptionen

- `-std=c++{98,03,11,14,17,20,2b}`
  Auswahl der Sprachversion

- `-c`
  Quelldatei(en) nur übersetzen, aber nicht zu einem ausführbaren Programm zusammenbinden

- `-o` *`program`*
  Ausführbares Programm mit dem Namen *`program`* statt `a.out` erzeugen

- `-O{0,1,2,3}`
  Zielcode mehr oder weniger stark optimieren

- `-g`
  Informationen für Debugger (z. B. GDB) in das übersetzte Programm integrieren

- `-D` *`name`* `[`*`definition`*`]`
  Quasi `#define` *`name definition`* bzw. `#define` *`name`* `1` (wenn *`definition`* nicht angegeben ist) von der Kommandozeile aus

- `-I` *`directory`*
  `#include`-Dateien zusätzlich im Verzeichnis *`directory`* suchen

- `-L` *`directory`*
  Bibliotheken zusätzlich im Verzeichnis *`directory`* suchen

- `-l`*`library`*
  Bibliothek mit dem Namen *`library`* zum Programm hinzufügen

# 1.5 Literaturhinweise


- B. Stroustrup: *Die C++-Programmiersprache* (Aktuell zum C++11-Standard). Carl Hanser Verlag, München, 2015.
  (Korrektes Deutsch wäre wohl eher: *Die Programmiersprache C++*)
- S. Meyers: *Effective Modern C++*. O'Reilly, Sebastopol, CA, 2014.
- ISO-C++-Standards 1998, 2003, 2011, 2014, 2017, 2020
  (bzw. die entsprechenden Entwurfsdokumente)
- http:// www.cppreference.com
- http://www.cplusplus.com/reference
- https://de.wikipedia.org
- https://en.wikipedia.org

# 2 Typen und Deklarationen

## 2.1 Elementare Typen (fundamental types)

### 2.1.1 Nichts

- Typ: `void`
- Werte: keine
- Operatoren: keine
- Zweck: Deklaration von Funktionen ohne Rückgabewert

## 2.1.2 Wahrheitswerte

- Typ: `bool`
- Werte: `false`, `true`
- Operatoren:
  - logisches Und: `&&` (infix)
  - logisches Oder: `||` (infix)
  - logische Negation: `!` (präfix)
  - bedingte Auswertung: `?:` (ternär)
- `&&` und `||` werten ihren rechten Operanden nur aus, wenn dies zur Ermittlung des Ergebnisses notwendig ist.
- `?:` verwendet den Wert seines ersten Operanden, um zu entscheiden, ob der zweite *oder* dritte Operand ausgewertet wird.
- Alle anderen elementaren Typen können implizit nach `bool` konvertiert werden, wobei `0` und `nullptr` als `false` und alle anderen Werte als `true` interpretiert werden.

## 2.1.3 Zeichen

- Typen: `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`
- Werte: `'A'`, `'0'`, `'\n'`, `'\"'`, `'\007'`, `'\x20'`, `'\0'`, ...
- Optionales Präfix: `u8`, `u`, `U` oder `L`
- Typ eines Werts:
  - Ohne Präfix → `char`
  - Mit Präfix `u8` → `char8_t` ab C++20, `char` in C++17
  - Mit Präfix `u` → `char16_t`
  - Mit Präfix `U` → `char32_t`
  - Mit Präfix `L` → `wchar_t`
- Operatoren: wie bei ganzen Zahlen (vgl. § 2.1.4)

## 2.1.4 Ganze Zahlen

- Typen:
  - Mit Vorzeichen: `signed char`, `short`, `int`, `long`, `long long`
  - Ohne Vorzeichen: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`
  - Mit oder ohne Vorzeichen (implementierungsabhängig): `char`
- Werte:
  - Dezimal: `123`, `1'000'000`, ...
  - Oktal: `0377`, ...
  - Hexadezimal: `0xff`, `0X1E`, ...
  - Dual: `0b1011'0011`, ...
- Optionale Suffixe (in beliebiger Reihenfolge):
  - `u` oder `U`
  - `l`, `L`, `ll` oder `LL`

- Typ eines Werts:
  - Normalerweise der erste Typ aus der Liste `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, der den Wert enthält.
  - Bei einem Wert mit Suffix `l` oder `L` werden jedoch die Typen `int` und `unsigned int` aus der Liste entfernt.
  - Bei einem Wert mit Suffix `ll` oder `LL` werden zusätzlich die Typen `long` und `unsigned long` aus der Liste entfernt.
  - Bei einem Wert mit Suffix `u` oder `U` werden außerdem alle vorzeichenbehafteten Typen aus der Liste entfernt.
  - Bei einem Dezimalwert ohne Suffix `u` oder `U` werden außerdem alle vorzeichenlosen Typen aus der Liste entfernt.

- Operatoren:
  - Grundrechenarten: `+`, `-`, `*`, `/`, `%` (infix)
  - Vorzeichen: `+`, `-` (präfix)
  - Inkrement und Dekrement: `++`, `--` (prä- oder postfix)
  - bitweises Und, Exklusiv-Oder und Oder: `&`, `^`, `|` (infix)
  - bitweises Komplement: `~` (präfix)
  - bitweises Verschieben nach links bzw. rechts: `<<`, `>>` (infix)
  - Vergleiche: `==`, `!=`, `<`, `>`, `<=`, `>=`, `<=>` (infix)
- Operationen mit vorzeichenlosen Werten der Größe $n$ Bit sind immer korrekt modulo $2^n$, d. h. das Verhalten bei Überlauf ist wohldefiniert.
- Wenn das Resultat einer Operation mit vorzeichenbehafteten Werten außerhalb des Wertebereichs des Resultattyps liegt, ist das Verhalten jedoch undefiniert.
- Ganzzahlige Division schneidet Nachkommastellen ab, d. h. es wird immer in Richtung 0 gerundet.

- Wenn der eigentliche Wert der Division `x/y` existiert und innerhalb des Wertebereichs des Resultattyps liegt, ist `x/y * y + x%y` gleich `x`.

  Daraus folgt, dass `x%y` nur für $x \geq 0$ und $y > 0$ mit der mathematischen Definition von `x` mod `y` übereinstimmt.

  Für $y > 0$ stimmt `(x%y + y) % y` immer mit `x` mod `y` überein.

## 2.1.5 Aufzählungswerte

### Einfache Aufzählungstypen (unscoped enumeration types)

- Beispieltyp: `enum Color { red, green, blue };`
- Werte des Beispieltyps: `red` (0), `green` (1), `blue` (2)
- Operatoren: i. w. wie bei ganzen Zahlen (vgl. § 2.1.4),
  da Werte einfacher Aufzählungstypen bei Bedarf in ganze Zahlen umgewandelt werden (vgl. § 2.1.8)

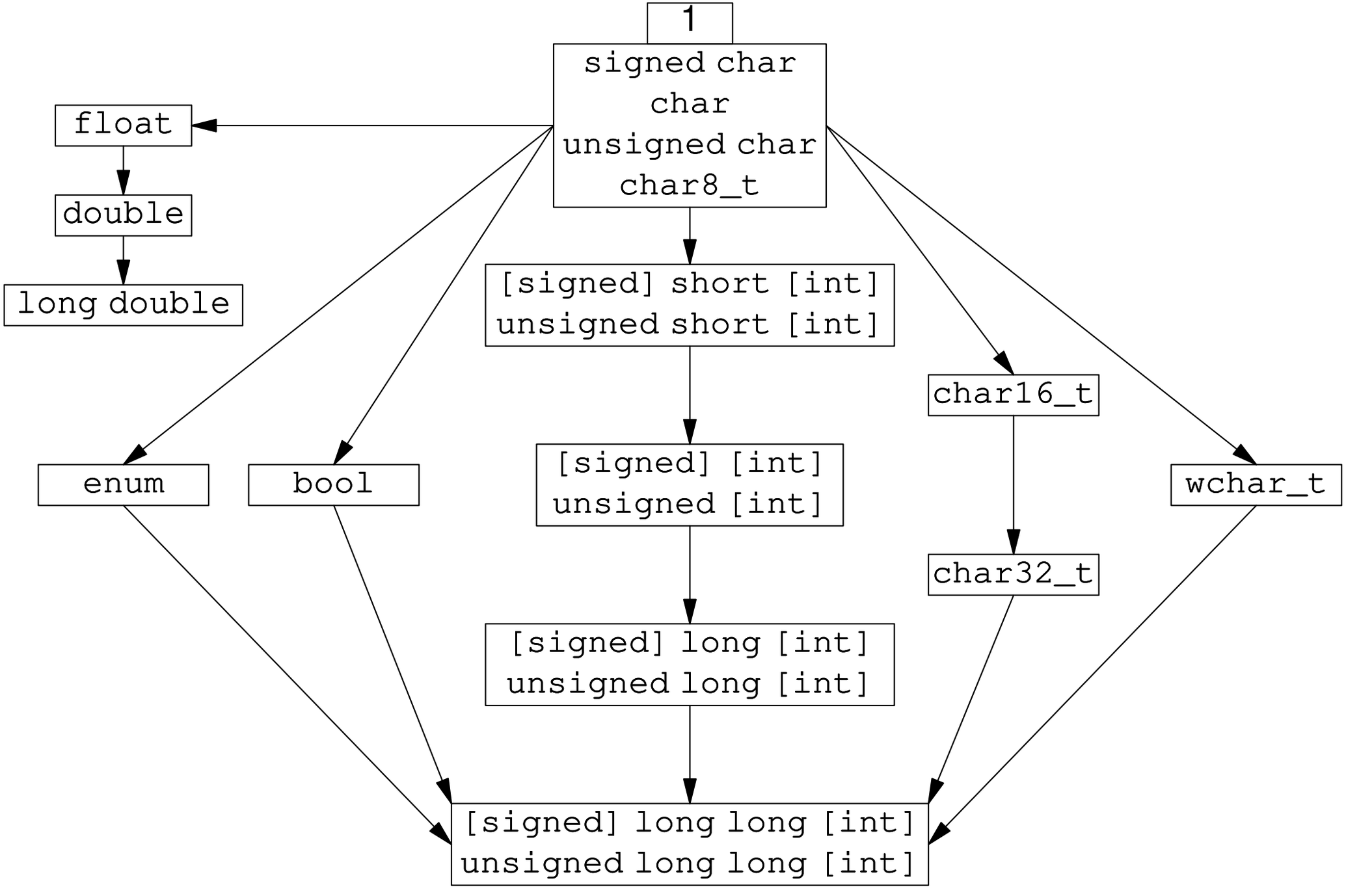
### Aufzählungstypen mit eigenem Namensbereich (scoped enumeration types)

- Beispieltyp: `enum class Color { red, green, blue };`
- Werte des Beispieltyps: `Color::red`, `Color::green`, `Color::blue`
- Operatoren: nur Vergleiche,
  da Werte von Aufzählungstypen mit eigenem Namensbereich *nicht implizit* in ganze Zahlen umgewandelt werden (*explizite* Umwandlungen sind aber möglich)

## 2.1.6 Gleitkommazahlen

- Typen: `float`, `double`, `long double`
- Werte: `1.0`, `1.`, `0.5`, `.5`, `2e6`, `3.5e-8`, `1.999'999`, `...`
- Optionales Suffix: `f`, `F`, `l` oder `L`
- Typ eines Werts:
  - Ohne Suffix → `double`
  - Mit Suffix `f` oder `F` → `float`
  - Mit Suffix `l` oder `L` → `long double`
- Operatoren: wie bei ganzen Zahlen (vgl. § 2.1.4), ausgenommen Modulo- und Bitoperationen

## 2.1.7 Größenverhältnisse



- Alle Typen innerhalb eines Rechtecks besitzen die gleiche Größe.

- Die Größe von `char` ist 1 Byte.
  Allerdings ist die Anzahl von Bits pro Byte nicht exakt festgelegt.

- Ein Pfeil von einem Rechteck `X` zu einem anderen Rechteck `Y` bedeutet, dass die Typen in `Y` mindestens so groß wie die Typen in `X` sind. Ob sie echt größer sind, ist jeweils implementierungsabhängig.

- Alle in der Abbildung dargestellten Typen sind logisch verschieden, selbst wenn einige von ihnen in einer bestimmten Implementierung die gleiche Wertemenge besitzen.

- Die folgende Tabelle zeigt die vom Standard vorgeschriebenen Mindestgrößen der Typen in Bit sowie ihre typische Größen auf 16-, 32- und 64-Bit-Prozessoren:

| *Typ* | *Mindest-größe* | *Typische Größe auf Prozessoren mit . . . Bit* | | |
|---|---|---|---|---|
| | | *16* | *32* | *64* |
| char | 8 | 8 | 8 | 8 |
| short | 16 | 16 | 16 | 16 |
| int | 16 | 16 | 32 | 32 |
| long | 32 | 32 | 32 | 64 |
| long long | 64 | 64 | 64 | 64 |

- In der Definitionsdatei `<limits>` gibt es für jeden in der Abbildung dargestellten Typ `T` (außer `enum`) eine Klasse `numeric_limits<T>`, die zahlreiche Eigenschaften des Typs `T` in der vorliegenden Implementierung enthält, z. B. `numeric_limits<T>::min()` und `numeric_limits<T>::max()`.

- In der Definitionsdatei `<cstdint>` sind folgende Typen als Aliase definiert:

```
int_least8_t    uint_least8_t   int_fast8_t     uint_fast8_t
int_least16_t   uint_least16_t  int_fast16_t    uint_fast16_t
int_least32_t   uint_least32_t  int_fast32_t    uint_fast32_t
int_least64_t   uint_least64_t  int_fast64_t    uint_fast64_t
intmax_t        uintmax_t
```

- Ein `least`-Typ ist der kleinste Typ, der mindestens die genannte Anzahl von Bits besitzt.

- Ein `fast`-Typ ist der schnellste (effizienteste) Typ, der mindestens die genannte Anzahl von Bits besitzt.

- Die `max`-Typen sind die größten verfügbaren ganzzahligen Typen.

- *Optional* sind zusätzlich folgende Aliase definiert:

```
int8_t          uint8_t
int16_t         uint16_t
int32_t         uint32_t
int64_t         uint64_t
intptr_t        uintptr_t
```

- Die `ptr`-Typen sind mindestens so groß wie Zeigertypen, d. h. bei einer Umwandlung eines Zeigerwerts in einen dieser Typen und wieder zurück erhält man denselben Zeigerwert (d. h. es geht keine Information verloren).

- Die anderen Typen besitzen exakt die genannte Anzahl von Bits.
  Wenn sie vorzeichenbehaftet sind, werden negative Werte im Zweierkomplement dargestellt (was ansonsten grundsätzlich nicht festgelegt ist).

## 2.1.8 Umwandlungen

### Übliche arithmetische Umwandlungen (usual arithmetic conversions)

- Bei arithmetischen Operationen und Vergleichen werden die Typen der Operanden „normalisiert“, d. h. in einen gemeinsamen Typ umgewandelt, der wie folgt bestimmt wird:

- Wenn sich die Operandentypen in der gleichen Gruppe der nachfolgenden Tabelle befinden, wird diese Gruppe ausgewählt; andernfalls die weiter oben stehende Gruppe.

```
long double
-----------------------------------------------------
double
-----------------------------------------------------
float
-----------------------------------------------------
long long       unsigned long long
-----------------------------------------------------
long            unsigned long
-----------------------------------------------------
int             unsigned int    short   unsigned short
signed char     unsigned char   char    bool
```

- Wenn die so ausgewählte Gruppe nur einen Typ enthält (`float`, `double` oder `long double`), wird dieser als gemeinsamer Typ verwendet.

- Andernfalls:
  - Wenn der erste Typ in dieser Gruppe (`int`, `long` oder `long long`) alle Werte der Operandentypen enthält, wird er als gemeinsamer Typ verwendet.
  - Andernfalls wird der zweite Typ in dieser Gruppe (`unsigned int`, `unsigned long` oder `unsigned long long`) verwendet (selbst wenn er nicht alle Werte der Operandentypen enthält).
    Wenn dieser Typ Größe $n$ Bit besitzt, wird ein vorzeichenbehafteter Wert modulo $2^n$ abgebildet.

    Dies kann überraschende Folgen haben:
    Für `unsigned int x = ...` liefert der Vergleich `x > -1` immer `false` (!), weil der `int`-Wert `-1` in `unsigned int` umgewandelt wird und dann dem größtmöglichen `unsigned-int`-Wert entspricht.

- Die Typen `char8_t`, `char16_t`, `char32_t`, `wchar_t` sowie einfache Aufzählungstypen werden zuvor in den ersten Typ aus der Liste `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long` umgewandelt, der alle Werte des Aufzählungstyps enthält.

- Die o. g. Umwandlungen werden auch bei unären Operatoren angewandt.

## Weitere Umwandlungen

- Alle in § 2.1.7 dargestellten Typen (wobei `enum` einfache Aufzählungstypen repräsentiert) können beliebig ineinander umgewandelt werden.

- Wenn der Zieltyp ein Aufzählungstyp ist, muss die Umwandlung explizit erfolgen (*Cast*), andernfalls erfolgt sie bei Bedarf auch implizit (selbst wenn Werte dabei verfälscht werden).

# 2.2 Reihen (arrays)

## 2.2.1 Deklaration

- Eine Deklaration `T a [N]` mit einem (nicht ganz) beliebigen Typ `T` und einer positiven ganzzahligen Konstanten `N` deklariert eine *Reihe* `a` vom Typ `T[N]` mit `N` Elementen des Typs `T`.

- Eine Deklaration `T a [N1] ... [Nk]` mit zwei oder mehr positiven ganzzahligen Konstanten `N1` bis `Nk` deklariert eine `k`-dimensionale Reihe vom Typ `T[N1]...[Nk]` mit `N1`$\cdot \ldots \cdot$`Nk` Elementen des Typs `T`, die gleichbedeutend mit einer eindimensionalen Reihe mit `N1` Elementen des Typs `T[N2]...[Nk]` ist.

## 2.2.2 Beispiele

```
char s [256];
int x [10];
double m [4] [8];
long* v [5];
```

## 2.2.3 Elementzugriff

- Für eine Reihe `a` vom Typ `T[N]` und eine ganze Zahl `i` zwischen 0 einschließlich und `N` ausschließlich liefert der Ausdruck `a[i]` das `i`-te Element von `a`.

- Wenn `a` ein L-Wert ist (vgl. § 2.9.1), ist auch `a[i]` ein L-Wert, d. h. es kann sowohl als Wert in Ausdrücken als auch als Ziel von Zuweisungen verwendet werden.

- Wenn `i` außerhalb des zulässigen Bereichs liegt, ist das Verhalten undefiniert.
  (Es findet keine Überprüfung statt.)

## 2.2.4 Initialisierung

- Reihen können bei ihrer Deklaration direkt mit einer Liste von Werten in geschweiften Klammern initialisiert werden.
- In diesem Fall kann die Elementzahl `N` (bei einer mehrdimensionalen Reihe die erste Elementzahl `N1`) weggelassen werden, weil sie aus der Anzahl der Werte ermittelt werden kann.
- Bei mehrdimensionalen Reihen können (müssen aber nicht) verschachtelte Listen verwendet werden.
- Die Werte dürfen beliebige Ausdrücke (nicht nur Konstanten) sein.
- Beispiele:

```
int a [5] = { 10, 20, 30 };
int b [] [3] = { { 1, 2, 3 }, { 4 } };
int c [] [2] = { 1, 2, { 3 }, 4, 5, 6 };
```

- Seit C++11 kann das Gleichheitszeichen vor den geschweiften Klammern auch weggelassen werden.

# 2.3 Zeiger

## 2.3.1 Deklaration

- Eine Deklaration `T * p` mit einem (nicht ganz) beliebigen Typ `T` ungleich `void` deklariert einen *Zeiger* `p` vom Typ `T*` auf ein Objekt des Typs `T`.

- Eine Deklaration `void * p` deklariert einen Zeiger `p` vom Typ `void*`, der auf Objekte beliebiger Typen zeigen kann.

## 2.3.2 Beispiele

```
int* p;          // Zeiger auf int.
char** q;        // Zeiger auf Zeiger auf char.
double* r [10]; // Reihe von Zeigern auf double.
bool (*s) [5];  // Zeiger auf Reihe von bool.

void* t;         // Zeiger auf irgendetwas.
void** u;        // Zeiger auf einen Zeiger vom Typ void*,
                 // d. h. Zeiger auf Zeiger auf irgendetwas.
```

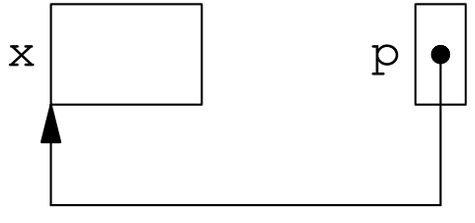
## 2.3.3 Achtung

```
int* x, y;
// Bedeutet nicht:
int* x; int* y;
// Sondern:
int* x; int y;
```

## 2.3.4 Inhalts- und Adressoperator

- In diesem und allen weiteren Teilabschnitten von § 2.3 sei `T` ein Typ ungleich `void`.
- Für einen Zeiger `p` vom Typ `T*` liefert der Ausdruck `*p` das Objekt vom Typ `T`, auf das `p` zeigt.
- Das Resultat dieser Operation ist ein L-Wert (vgl. § 2.9.1), egal ob `p` ein L-Wert ist oder nicht.
- Für einen L-Wert `x` vom Typ `T` liefert der Ausdruck `&x` einen Zeiger vom Typ `T*`, der auf `x` zeigt.
- Somit gilt:

```
&*p == p
*&x == x
```



### 2.3.5 Nullzeiger

- Das Schlüsselwort `nullptr` kann implizit in einen *Nullzeiger* jedes Zeigertyps `T*` umgewandelt werden.

- Da ein Nullzeiger auf *kein Objekt* zeigt, ist die Anwendung des Inhaltsoperators in diesem Fall undefiniert (und führt in der Regel zu einem Programmabsturz).

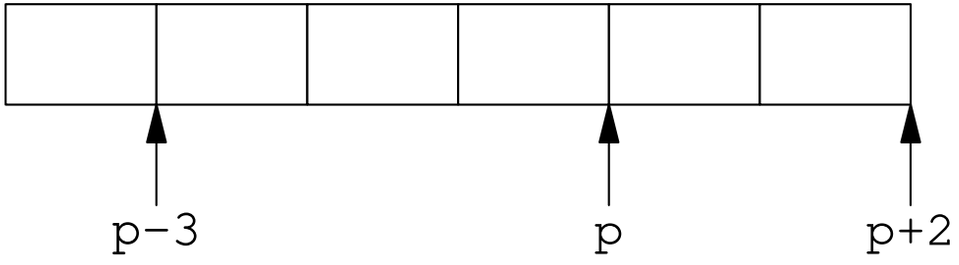
### 2.3.6 Zeiger auf dynamische Objekte

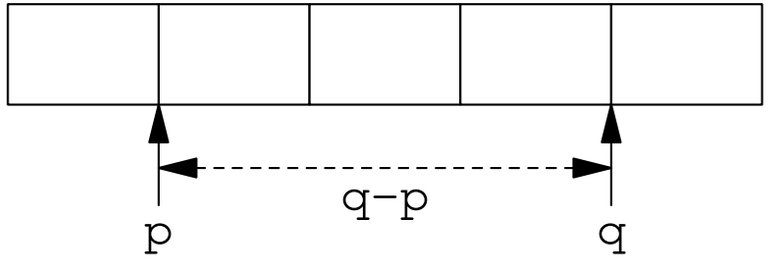
- Für einen (nicht ganz) beliebigen Typ `T` ungleich `void` (und eine nicht-negative ganze Zahl `N`) erzeugt der Ausdruck `new T` (bzw. `new T [N]`) ein *dynamisches Objekt* (bzw. eine *dynamische Reihe* mit `N` Objekten) des Typs `T` und liefert einen Zeiger darauf zurück.

- Für einen von `new` gelieferten Zeigerwert `p` löscht der Ausdruck `delete p` (bzw. `delete [] p`) das dynamische Objekt (bzw. die dynamische Reihe), auf das (bzw. die) `p` zeigt. Anschließend darf `p` nicht mehr dereferenziert werden.
  Der Unterschied zwischen `delete p` und `delete [] p` ist wichtig, wenn der Typ `T` einen (nicht trivialen) Destruktor (vgl. § 3.3) besitzt. (In diesem Fall kann die Verwendung der falschen Variante fatale Folgen haben.)

## 2.3.7 Adressarithmetik

- Für einen Zeiger `p` vom Typ `T*` und eine ganze Zahl `i` sind `p + i` und `p - i` ebenfalls Zeiger vom Typ `T*`.
- Wenn `p` auf das Element mit Index `k` einer Reihe mit `N` Elementen des Typs `T` zeigt, zeigt `p + i` bzw. `p - i` auf das Element mit Index `k + i` bzw. `k - i` dieser Reihe.
- Zeiger auf das nicht vorhandene Element mit Index `N` dieser Reihe, das unmittelbar hinter der Reihe liegen würde, sind hierbei als Grenzfall zulässig. (Sie dürfen aber nicht dereferenziert werden.)
- Ein einzelnes Objekt des Typs `T` wird hierbei wie eine Reihe mit einem Element behandelt, sodass für einen Zeiger `p` auf ein Objekt des Typs `T` die Ausdrücke `p + 1` und `(p + 1) - 1` semantisch korrekt sind.
- Für zwei Zeiger `p` und `q` vom Typ `T*` ist `q - p` eine ganze Zahl vom Typ `ptrdiff_t`.
- Wenn `p` bzw. `q` auf das Element mit Index `i` bzw. `j` einer Reihe mit `N` Elementen des Typs `T` zeigt (wobei `i` bzw. `j` gleich `N` wiederum zulässig ist), ist `q - p` gleich `j - i`.
- Falls dieser Wert nicht zum Wertebereich von `ptrdiff_t` gehört (was prinzipiell möglich ist), ist das Verhalten undefiniert.




## 2.3.8 Äquivalenz von Reihen und Zeigern

- Eine Reihe `a` vom Typ `T[]` ist gleichzeitig ein (konstanter) Zeiger vom Typ `T*`, der auf das „nullte“ Element von `a` zeigt.

- Umgekehrt kann ein Zeiger `p` vom Typ `T*` auch als Reihe (unbekannter Größe) vom Typ `T[]` aufgefasst werden.

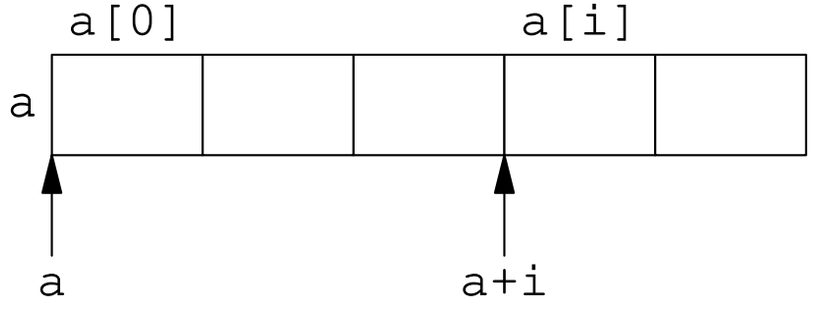
- Somit gilt für eine ganze Zahl `i`:

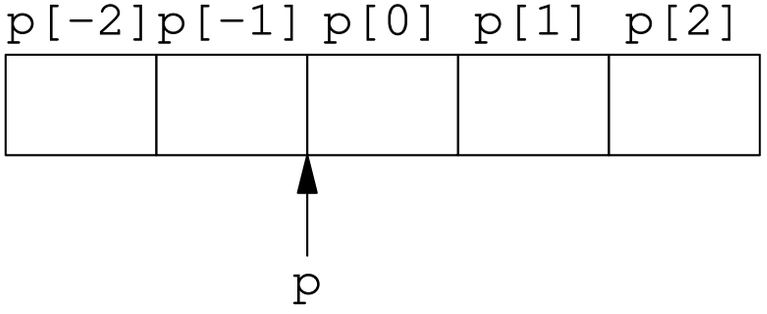
```
a[i] == *(a+i)          p[i] == *(p+i)
&a[i] == a+i            &p[i] == p+i
```

und speziell:

```
a[0] == *a              p[0] == *p
&a[0] == a              &p[0] == p
```

- Ausnahme: Wenn `a` kein L-Wert (vgl. § 2.9.1) ist, ist auch `a[i]` kein L-Wert (vgl. § 2.2) und somit `&a[i]` nicht korrekt (obwohl es von GCC 7.2 akzeptiert wird). Nichtsdestotrotz ist `a+i` korrekt, und `*(a+i)` ist ein L-Wert.



## 2.3.9 Umwandlungen

- Ein Zeiger vom Typ `T*` kann implizit in einen Zeiger vom Typ `void*` umgewandelt werden.
  Umgekehrt kann ein Zeiger vom Typ `void*` explizit in einen Zeiger vom Typ `T*` umgewandelt werden:

```
T* p = ......;

void* q = p;

T* r = (T*)q;
// Oder:
T* r = static_cast<T*>(q);
```

- Ein Zeiger vom Typ `T*` kann explizit in einen Zeiger eines anderen Typs `S*` (insbesondere `char*`) umgewandelt werden.
  Ebenso kann ein Zeiger explizit in eine ganze Zahl umgewandelt werden und umgekehrt:

```
T* p = ......;

S* q = (S*)p;
// Oder:
S* q = reinterpret_cast<S*>(p);

uintptr_t u = (uintptr_t)p;
// Oder:
uintptr_t u = reinterpret_cast<uintptr_t>(p);

T* r = (T*)u;
// Oder:
T* r = reinterpret_cast<T*>(u);
```

- Sofern der Typ der ganzen Zahl ausreichend groß ist (was für `intptr_t` und `uintptr_t` der Fall ist, sofern diese Typen definiert sind, vgl. § 2.1.7), liefert eine Umwandlung eines Zeigers in eine ganze Zahl und wieder zurück den ursprünglichen Zeigerwert.

# 2.4 Konstante Objekte

## 2.4.1 Konzept

- (Fast) jeder Typ kann `const`-qualifiziert werden, um auszudrücken, dass seine Objekte nicht verändert werden dürfen.

- Ausnahmen sind Reihen (§ 2.2), Referenzen (§ 2.9) und Funktionen (§ 2.10), die grundsätzlich nicht verändert werden können.
  (Die Elemente einer Reihe und das Objekt, auf das eine Referenz verweist, können natürlich verändert werden, sofern *ihr* Typ nicht `const`-qualifiziert ist.)

- Da ein `const`-Objekt nicht verändert werden darf, muss es bereits bei seiner Deklaration *initialisiert* werden.

## 2.4.2 Konstante Zeiger und Zeiger auf konstante Objekte

- Bei Zeigertypen muss unterschieden werden, ob der Zeiger oder das Objekt, auf das er zeigt, oder beides konstant sein soll.

- Beispiel:

```
// Gewöhnlicher Zeiger.
char* p;
p = ......;                          // OK.
*p = ......;                         // OK.

// Zeiger auf konstantes Objekt.
const char* pc;                      // Deklaration entweder so ...
char const* pc;                      // ... oder so.
pc = ......;                         // OK.
*pc = ......;                        // Fehler.

// Konstanter Zeiger.
char* const cp = ......;
cp = ......;                         // Fehler.
*cp = ......;                        // OK.

// Konstanter Zeiger auf konstantes Objekt.
const char* const cpc = ......;      // Deklaration entweder so ...
char const* const cpc = ......;      // ... oder so.
cpc = ......;                        // Fehler.
*cpc = ......;                       // Fehler.
```

- Bei mehrstufigen Zeigern kann jede Ebene unabhängig `const`-qualifiziert werden.

- Bei einer Zuweisung oder Initialisierung `p = q` muss der Typ von `p` auf jeder Ebene außer der obersten `const`-qualifiziert sein, auf der der Typ von `q` `const`-qualifiziert ist.

  Das heißt:

  - Die `const`-Qualifizierer auf oberster Ebene sind irrelevant.
    (Allerdings darf `p` bei einer Zuweisung grundsätzlich nicht konstant sein.)
  - Auf den übrigen Ebenen darf der Typ von `p` „mehr“ `const`-Qualifizierer besitzen als der Typ von `q`, aber nicht umgekehrt.

### 2.4.3 „Unbeständige“ Objekte

- Analog zur Qualifizierung mit `const`, können Typen `volatile`-qualifiziert werden, um anzuzeigen, dass ihre Objekte „unbeständig“ sind, d. h. sich in einer für den Übersetzer unerwarteten Weise ändern können (und deshalb z. B. bestimmte Optimierungen vermieden werden müssen).

# 2.5 Zeichenketten (Strings)

## 2.5.1 C-Strings

- Eine Zeichenkette in Anführungszeichen wie z. B. `"hallo"` ist eine Reihe vom Typ `const char [N]`, wobei `N` die Anzahl der Zeichen plus eins (im Beispiel also 6) ist.
- Das letzte Zeichen der Reihe ist ein Nullzeichen `'\0'`.
- Aufgrund der Äquivalenz von Reihen und Zeigern (vgl. § 2.3.8) kann man an Zeiger des Typs `const char*` Zeichenketten beliebiger Länge zuweisen.
- Die Definitionsdatei `<cstring>` definiert Funktionen wie z. B. `strlen`, `strcpy` und `strcat` zur Verarbeitung solcher Zeichenketten.
- Allerdings muss man als Programmierer selbst den für Resultat-Zeichenketten benötigten Speicherplatz beschaffen und wieder freigeben.

## 2.5.2 C++-Strings

- Die Definitionsdatei `<string>` definiert einen wesentlich komfortableren Typ `string` mit Operationen zum Anfügen von Zeichen, Verketten von Zeichenketten, Extraktion von Teilzeichenketten etc.

- Die Verwaltung des benötigten Speicherplatzes erfolgt vollkommen automatisch.

- Werte des Typs `const char*` („C-Strings“), d. h. insbesondere Zeichenketten wie `"hallo"`, können implizit in Objekte des Typs `string` („C++-Strings“) umgewandelt werden.

# 2.6 Typ-Aliase

- Wenn eine Deklaration `D` ein Objekt oder eine Funktion `x` mit Typ `T` deklariert, dann deklariert die Deklaration `typedef D` den Namen `x` als Alias des Typs `T`.

  Zum Beispiel:

```
// Deklaration D.
const char* str;
double* const* array [5];
```

```
// Entsprechende Typ-Deklaration.
typedef const char* str;
typedef double* const* array [5];
```

- Wenn man aus der Deklaration `D` den deklarierten Namen `x` entfernt, bleibt der Typ `T` übrig.

  Zum Beispiel:

```
// Deklaration D.
const char* str;
double* const* array [5];
```

```
// Typ T.
const char*
double* const* [5]
```

- Eine Deklaration `using x = T` mit einem Namen `x` und einem Typ `T` deklariert den Namen `x` ebenfalls als Alias des Typs `T`.

  Zum Beispiel:

```
using str = const char*;
using array = double* const* [5];
```

# 2.7 Strukturen (structs, records)

## 2.7.1 Deklaration (Beispiele)

```
// Rationale Zahlen.
struct Rational {
  int num;                // Zähler.
  int den;                // Nenner.
} x, y;

// Knoten eines binären (Such-)Baums.
struct Node {
  const char* name;       // Z. B. der Name einer Person.
  Node* left;             // Zeiger auf den linken und ...
  Node* right;            // ... rechten Teilbaum (ggf. Nullzeiger).
};
Node n;
Node* p;
```

## 2.7.2 Elementzugriff

- Für ein Objekt `x` eines Strukturtyps `S` und den Namen `m` eines *Elements* (*member*) von `S` liefert der Ausdruck `x.m` das Element mit dem Namen `m` von `x`.
- Wenn das Strukturobjekt `x` ein L-Wert ist (vgl. § 2.9.1), ist auch `x.m` ein L-Wert.
- Für einen Zeiger `p` vom Typ `S*` liefert der Ausdruck `p->m` das Element mit dem Namen `m` des Objekts, auf das `p` zeigt.
- `p->m` ist immer ein L-Wert, egal ob der Strukturzeiger `p` ein L-Wert ist oder nicht.
- Somit gilt: `p->m == (*p).m`

## 2.7.3 Initialisierung

- Strukturobjekte können bei ihrer Deklaration direkt mit einer Liste von Werten in geschweiften Klammern initialisiert werden, die in der angegebenen Reihenfolge den Strukturelementen zugeordnet werden.

- Um Reihen von Strukturobjekten und/oder Strukturobjekte mit Reihen zu initialisieren, können, ähnlich wie bei mehrdimensionalen Reihen, verschachtelte Listen verwendet werden.

- Beispiele:

```
Node l = { "drei", nullptr, nullptr };
Node r = { "zwei", nullptr, nullptr };
Node t = { "eins", &l, &r };

Node n [] = {
  { "eins", n + 2, n + 1 },
  { "zwei", nullptr, nullptr },
  { "drei", nullptr, nullptr },
};
```

- Seit C++11 kann das Gleichheitszeichen vor den geschweiften Klammern auch weggelassen werden.

## 2.7.4 Elementzeiger (pointers to member)

- Eine Deklaration `T S::* pm` mit einem Strukturtyp `S` und einem (nicht ganz) beliebigen Typ `T` deklariert einen *Elementzeiger* mit Typ `T S::*` auf ein Element des Typs `S` mit Typ `T`.
- Für den Namen `m` eines Elements von `S` mit Typ `T` ist `&S::m` ein Elementzeiger mit Typ `T S::*`, der auf das Element `m` (eines beliebigen Objekts des Typs `S`) zeigt.
- Für ein Objekt `x` mit Typ `S` (bzw. einen Zeiger `p` mit Typ `S*`) und einen Elementzeiger `pm` mit Typ `T S::*` liefert der Ausdruck `x.*pm` (bzw. `p->*pm`) das Element von `x` (bzw. `*p`), auf das `pm` zeigt.
- Wenn das Strukturobjekt `x` ein L-Wert ist (vgl. § 2.9.1), ist auch `x.*pm` ein L-Wert. `p->*pm` ist immer ein L-Wert, egal ob der Strukturzeiger `p` ein L-Wert ist oder nicht.
- Somit gilt: `p->*pm == (*p).*pm`
- Anschaulich kann man sich einen Elementzeiger als (typsichere) relative Adresse eines Elements in einem Strukturobjekt vorstellen.

## 2.7.5 Veränderbare Elemente

- Elemente eines `const`-qualifizierten Strukturtyps sind automatisch `const`-qualifiziert, es sei denn sie sind `mutable` deklariert.

# 2.8 Überlagerungen (unions, variant records)

## 2.8.1 Konzept

- Ersetzt man das Schlüsselwort `struct` in der Deklaration eines Strukturtyps durch `union`, so erhält man einen Typ, dessen Objekte zu jedem Zeitpunkt genau *eines* der deklarierten Elemente enthalten.

- Rein syntaktisch kann man trotzdem zu jedem Zeitpunkt auf *jedes* Element zugreifen.

- Typischerweise verwendet man Überlagerungen als Teil einer Struktur, in der ein anderes Strukturelement die Information enthält, welches Element der Überlagerung momentan gültig ist.

- Anders als in C, können Überlagerungen in C++ *anonym* sein.

## 2.8.2 Beispiel: Repräsentation arithmetischer Ausdrücke

```
struct Expr {
  enum {
    Const, Var, Neg, Add, Sub, Mul, Div, Mod
  } kind;        // Art des Ausdrucks.

  union {
  char* name;    // Name einer Variablen ...
  Expr* body;    // ... oder Operand eines unären Ausdrucks ...
  Expr* left;    // ... oder linker Operand eines binären Ausdrucks.
  };

  union {
  double value; // Wert einer Variablen oder Konstanten ...
  Expr* right;  // ... oder rechter Operand eines binären Ausdrucks.
  };
};
```

# 2.9 Referenzen

## 2.9.1 L- und R-Werte

- Ein *L-Wert* ist ein Objekt, das eine Adresse besitzt (und nicht „im nächsten Moment“ verschwindet) und daher als Ziel einer Zuweisung verwendet werden darf (sofern es nicht `const` deklariert wurde und keine Funktion ist).
- Alle anderen Objekte sind *R-Werte*.
- Folgende Objekte sind L-Werte:
  - Variablen und Parameter
  - Reihenelemente `a[i]`, sofern die Reihe `a` ein L-Wert ist
  - Dereferenzierte Zeiger `*p`, egal der Zeiger `p` ein L-Wert ist oder nicht
  - Strukturelemente `x.m`, sofern das Strukturobjekt `x` ein L-Wert ist
  - Strukturelemente `p->m`, egal ob der Strukturzeiger `p` ein L-Wert ist oder nicht
  - Funktionen

## 2.9.2 Referenztypen

- Darüber hinaus gibt es *Referenztypen*, deren Objekte grundsätzlich L-Werte sind.
- Ein Objekt `r` eines Referenztyps `T&`, das mit einem L-Wert `x` des Typs `T` initialisiert wurde, entspricht konzeptuell einem *implizit dereferenzierten Zeigerwert* `*p`, dessen Zeiger `p` vom Typ `T*` mit der Adresse `&x` des Objekts `x` initialisiert wurde und nicht verändert werden kann.
- Daraus folgt: Die Adresse `&r` der Referenz `r` stimmt mit der Adresse `&x` des referenzierten Objekts `x` überein.
- Beispiel zum Vergleich von Referenzen und Zeigern:

```
int x = 10;
int& r = x;
cout << r << endl;
r = 20;
cout << x << endl;
cout << (&r == &x) << endl;
```

```
int x = 10;
int* p = &x;
cout << *p << endl;
*p = 20;
cout << x << endl;
cout << (&*p == &x) << endl;
```

Ausgabe in beiden Fällen: `10 20 1` (d. h. `true`)

- Achtung: In Java versteht man unter Referenzen das, was man in C++ Zeiger nennt.

## 2.9.3 Initialisierung von Referenzen

- Eine *Variable* eines Referenztyps `T&` muss bei ihrer Deklaration mit einem L-Wert des Typs `T` initialisiert werden.

- Ein *Funktionsparameter* eines Referenztyps `T&` wird beim Aufruf der Funktion mit dem entsprechenden Funktionsargument (aktueller Parameter) initialisiert, bei dem es sich um einen L-Wert des Typs `T` handeln muss.

- Ein *Funktionsresultat* eines Referenztyps `T&` wird quasi durch Ausführung einer Anweisung `return x` im Funktionsrumpf mit einem L-Wert `x` des Typs `T` initialisiert.

- Ein *Strukturelement* eines Referenztyps `T&` muss in jedem Konstruktor der Struktur (bzw. Klasse) mit Hilfe eines Elementinitialisierers mit einem L-Wert des Typs `T` initialisiert werden oder einen entsprechenden Initialisierungsausdruck besitzen (vgl. § 3.2.3).

## 2.9.4 Referenzen auf konstante Objekte

- Im Gegensatz zu allgemeinen Referenzen (vgl. § 2.9.2), dürfen Referenzen auf konstante Objekte (d. h. mit einem Typ `const T&`) auch mit R-Werten des Typs `T` initialisiert werden.

- In diesem Fall erzeugt der Übersetzer implizit ein temporäres Objekt `t` vom Typ `T`, das mit dem gegebenen R-Wert initialisiert wird, und initialisiert dann die Referenz mit dem L-Wert `t`.

- Ausnahme: Wenn es sich bei dem R-Wert um eine R-Wert-Referenz handelt (vgl. § 2.9.5), wird das temporäre Objekt nicht gebraucht, weil die Referenz in diesem Fall direkt mit der R-Wert-Referenz initialisiert werden kann.

## 2.9.5 R-Wert-Referenzen

- Seit C++11 gibt es zusätzlich *R-Wert-Referenzen*.
  Die bisher betrachteten Referenzen heißen zur Unterscheidung *L-Wert-Referenzen*.

- Ein Objekt `r` eines R-Wert-Referenztyps `T&&` kann *nur* mit einem R-Wert `x` des Typs `T` initialisiert werden.

- Genauso wie bei der Initialisierung einer Referenz des Typs `const T&`, wird hierfür ggf. ein temporäres Objekt erzeugt.

- Variablen und Parameter mit Typ `T&&` (d. h. R-Wert-Referenzen, die einen Namen besitzen), sind trotzdem L-Werte.

- Alle anderen Objekte mit Typ `T&&` sind R-Werte.

- Beispiel:

```
// Initialisierung von r1 mit dem R-Wert 1.
int&& r1 = 1;

// r1 ist ein L-Wert -> Zuweisung an r1 erlaubt.
r1 = 2;

// Fehler:
// r1 ist ein L-Wert -> Initialisierung von r2 so nicht erlaubt!
int&& r2 = r1;

// (int)r1 ist ein R-Wert -> Initialisierung so erlaubt.
int&& r3 = (int)r1;

// (int&&)r1 ist ebenfalls ein R-Wert -> Initialisierung erlaubt.
int&& r4 = (int&&)r1;

// Das heißt: Obwohl r1 bereits Typ int&& besitzt,
// besteht ein Unterschied zwischen r1 und (int&&)r1!
```

## 2.9.6 Typische Verwendung von Referenzen

- Parameterübergabe per Referenz
  (*call by reference*; VAR-Parameter in Pascal/Modula/Oberon):

```
void swap (int& a, int& b) {
  int tmp = a; a = b; b = tmp;
}
.......
int x = 1, y = 2;
cout << x << " " << y << endl;          // Ausgabe: 1 2
swap(x, y);
cout << x << " " << y << endl;          // Ausgabe: 2 1
```

- „Variablen“ als Funktionsresultate:

```
char& elem (char* a, int i) { return a[i]; }
.......
char x [10];
elem(x, 5) = 'y';
```

- *Vorsicht:* Wird ein Objekt per Referenz zurückgeliefert, so muss sichergestellt sein, dass es auch nach Beendigung der Funktion noch existiert.
  Insbesondere dürfen keine lokalen Variablen per Referenz zurückgeliefert werden!

- L-Wert-Referenzen auf konstante Objekte werden insbesondere zur effizienten Parameterübergabe an Funktionen verwendet, da ein Parameter vom Typ `const T&` für den Aufrufer der Funktion äquivalent zu einem Parameter vom Typ `T` ist:
  - Man kann *beliebige* Objekte (L- und R-Werte) vom Typ `T` als Argumente übergeben.
  - Übergibt man einen L-Wert, so wird dieser zwar per Referenz übergeben (was u. U. wesentlich effizienter als Übergabe per Wert ist), kann innerhalb der Funktion aber (normalerweise) nicht verändert werden.

  Der Effizienzgewinn kann insbesondere für große Strukturtypen und Typen mit explizitem Kopierkonstruktor (vgl. § 3.4) von Bedeutung sein.

- R-Wert-Referenzen können ebenfalls verwendet werden, um unnötiges Kopieren von Objekten bei Initialisierungen und Zuweisungen zu vermeiden (vgl. § 3.5):
  - Wenn das zu kopierende Objekt ein R-Wert ist – und deshalb an einen Parameter des Typs `T&&` übergeben werden kann –, wird es nach der Initialisierung oder Zuweisung nicht mehr existieren.
  - Deshalb ist es unnötig, seinen Inhalt in das Zielobjekt zu *kopieren*.
  - Es ist effizienter, seinen Inhalt in das Zielobjekt zu *verschieben* bzw. mit dessen Inhalt zu vertauschen.

- Außerdem können R-Wert-Referenzen – im Zusammenspiel mit Schablonen und den „reference collapsing rules“ (vgl. § 2.9.7) – verwendet werden, um Funktionsparameter ohne Änderung ihres L-Wert- oder R-Wert-Status an andere Funktionen weiterzugeben („perfect forwarding“, vgl. § 5.5.3).

## 2.9.7 Referenzen auf Referenzen

- Referenzen auf Referenzen sind (ebenso wie Zeiger auf Referenzen sowie Reihen von Referenzen) nicht zulässig.

- Wenn sie jedoch indirekt durch Typ-Aliase oder Belegung von Schablonen-Parametern entstehen, werden sie nach folgenden Regeln aufgelöst („reference collapsing rules“):
  - `T & &` → `T&`
  - `T & &&` → `T&`
  - `T && &` → `T&`
  - `T && &&` → `T&&`

- Das heißt:
  Die Kombination `&& &&` liefert `&&`, alle anderen Kombinationen von `&` und `&&` liefern `&`.

- Merkregel:
  - L-Wert-Referenz „sticht“ R-Wert-Referenz.
  - Wie bei einer logischen Und-Verknüpfung: faLse (entspricht L-Wert-Referenz) setzt sich gegen tRue (entspricht R-Wert-Referenz) durch.

# 2.10 Funktionstypen, -zeiger und -referenzen

## 2.10.1 Deklaration

- Eine Deklaration `typedef R F (P1, ..., Pn)` oder `using F = R (P1, ..., Pn)` mit einem (nicht ganz) beliebigen Typ `R` sowie beliebig vielen (nicht ganz) beliebigen Typen `P1` bis `Pn` deklariert einen Funktionstyp `F` mit Parametertypen `P1` bis `Pn` und Resultattyp `R`.

- Eine Deklaration `R (*fp) (P1, ..., Pn)` bzw. `R (&fr) (P1, ..., Pn)` deklariert einen Funktionszeiger `fp` bzw. eine Funktionsreferenz `fr` auf eine Funktion des Typs `F`.

## 2.10.2 Verwendung

- Der Name einer Funktion `f` kann implizit in einen Zeiger oder eine Referenz auf diese Funktion umgewandelt werden, so dass Funktionsnamen direkt als Werte von Funktionszeigern und -referenzen verwendet werden können.

- Funktionszeiger und -referenzen können wie Funktionen aufgerufen werden, d. h. Funktionszeiger werden bei Bedarf implizit dereferenziert.

## 2.10.3 Beispiel: Flexible Sortierfunktion

```
#include <cstring>

// sort erhält eine Reihe a mit n Zeichenketten sowie eine Funktion
// (eigentlich einen Funktionszeiger) c mit Parametertypen str, str
// und Resultattyp int zum Vergleich zweier Zeichenketten.
using str = const char*;
void sort (str* a, int n, int c (str, str)) {
  ......
  // Aufruf der Funktion c, um a[i] und a[j] zu vergleichen.
  int x = c(a[i], a[j]);
  ......
}

// Zu sortierende Reihe a mit n Zeichenketten.
str a [] = { ...... };
const int n = sizeof a / sizeof a[0];

// Sortierung von a mit unterschiedlichen Vergleichsfunktionen
// strcmp und strcoll (vgl. Definitionsdatei <cstring>).
sort(a, n, strcmp);
sort(a, n, strcoll);
```

## 2.10.4 Beispiel: Signalbehandlungsfunktionen

```
// sighandler_t ist ein Alias für den Typ:
// Zeiger auf Funktion mit Parametertyp int und Resultattyp void.
using sighandler_t = void (*) (int);

// signal erhält eine Signalnummer s und einen solchen Funktions-
// zeiger h und liefert einen solchen Funktionszeiger als Resultat.
sighandler_t signal (int s, sighandler_t h);

// SIG_IGN ist der Wert 1,
// umgewandelt in den Funktionszeigertyp sighandler_t.
#define SIG_IGN ((sighandler_t)1)

// Definition einer passenden Signalbehandlungsfunktion.
void handler (int s) {
  ......
}

// Typische Verwendung von signal.
sighandler_t prev = signal(SIGINT, handler);
......
signal(SIGINT, prev);
```

## Dasselbe ohne Definition von `sighandler_t`

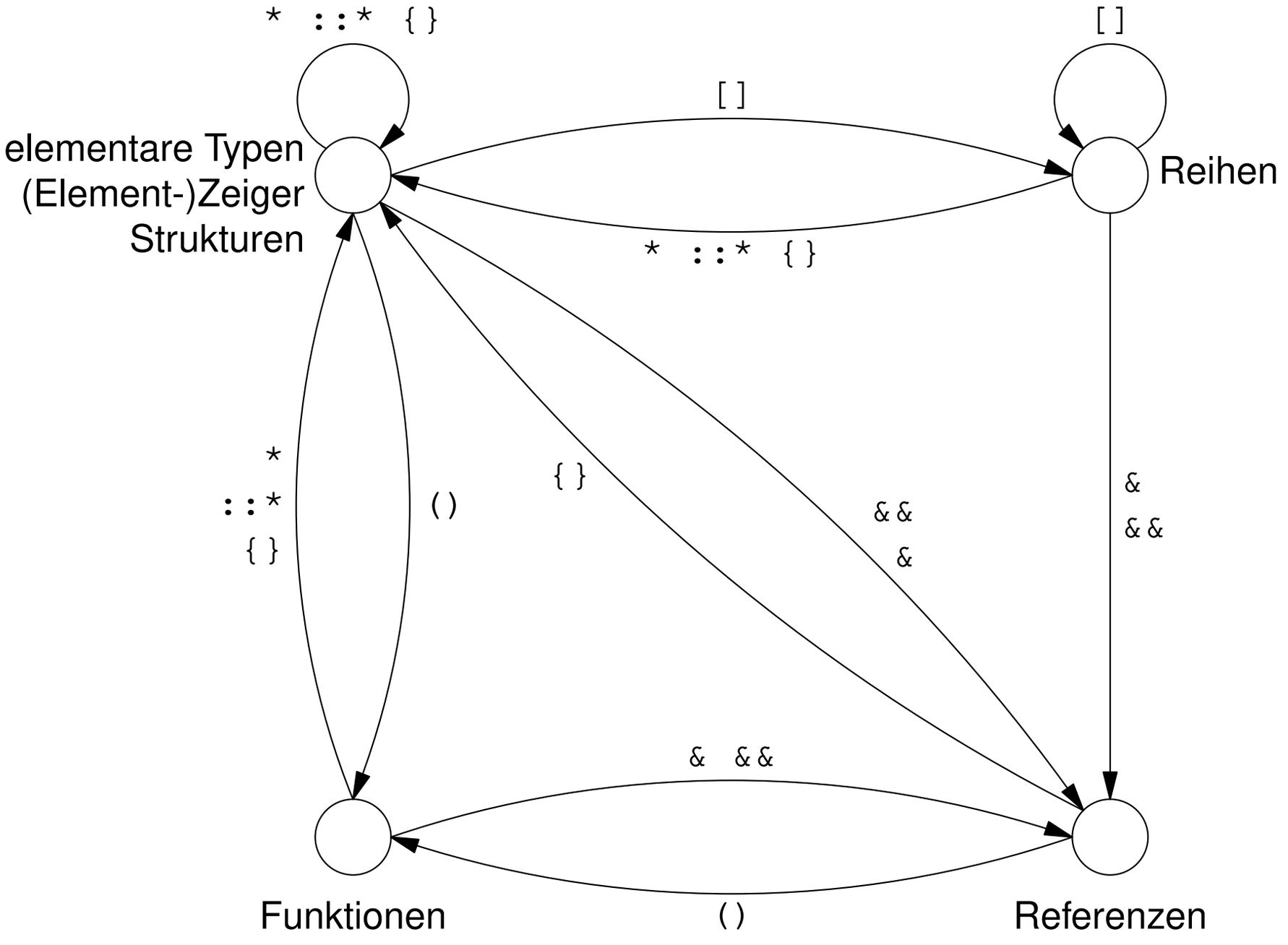
```
// Insbesondere die Definition von signal ist schwer zu verstehen.
void (* signal (int s, void (*h) (int))) (int);

#define SIG_IGN ((void (*) (int)) 1)

void handler (int s) {
  ......
}

void (*prev) (int) = signal(SIGINT, handler);
......
signal(SIGINT, prev);
```

# 2.11 Kombinationsmöglichkeiten der Typkonstruktoren



- Jeder Knoten (Kreis) des Graphen repräsentiert eine bestimmte Menge von Typen.
- Eine Kante (Pfeil) von einem Knoten X zu einem Knoten Y bedeutet je nach Beschriftung:

  Jeder Typ aus der Menge X kann als
  - Zieltyp eines Zeigers (`*`)
  - Zieltyp eines Elementzeigers (`::*`)
  - Zieltyp einer Referenz (`&`, `&&`)
  - Elementtyp einer Reihe (`[]`)
  - Typ eines Strukturelements (`{}`)
  - Parameter- oder Resultattyp einer Funktion (`()`)

  verwendet werden, und der so gebildete Typ gehört zur Menge Y.
- Reihen- und Funktionstypen können zwar syntaktisch als Parametertypen von Funktionen verwendet werden, werden aber implizit durch Zeiger- bzw. Funktionszeigertypen ersetzt.

# 2.12 Deklarationen und Definitionen

## 2.12.1 Unterschied

- Obwohl die Begriffe *Deklaration* und *Definition* von Funktionen, Variablen und Typen häufig (auch in dieser Vorlesung) synonym verwendet werden, besteht ein wichtiger Unterschied, zum Beispiel:

```
// Deklaration der Funktion swap ohne Implementierung.
void swap (int&, int&);

// Deklaration der Variablen x und y mit Schlüsselwort extern.
extern int x, y;

// Deklaration des Strukturtyps Node ohne Inhalt.
struct Node;

// x, y und swap können hier bereits uneingeschränkt
// verwendet werden.
x = 1;
y = 2;
swap(x, y);
```

```
// Node kann aber nur sehr eingeschränkt verwendet werden.
Node* p;                    // Geht.
using NodePtr = Node*;      // Geht.
Node n;                     // Geht nicht.

// Definition der Funktion swap mit Implementierung.
void swap (int& a, int& b) {
  int tmp = a; a = b; b = tmp;
}

// Definition der Variablen x und y (ggf. auch mit
// Initialisierung) ohne Schlüsselwort extern.
int x, y;

// Definition des Strukturtyps Node mit Inhalt.
struct Node {
  const char* name;         // Z. B. der Name einer Person.
  NodePtr left;             // Zeiger auf den linken und ...
  NodePtr right;            // ... rechten Teilbaum (ggf. Nullzeiger).
};
```

## 2.12.2 Regeln

- In einer Übersetzungseinheit darf eine Funktion, eine Variable oder ein Typ beliebig oft deklariert, aber höchstens einmal definiert werden.
- Um denselben Typ in verschiedenen Übersetzungseinheiten verwenden zu können, muss es in jeder Übersetzungseinheit eine (identische) Definition geben.
- Um dieselbe Funktion oder Variable in verschiedenen Übersetzungseinheiten verwenden zu können, muss sie normalerweise in genau einer Übersetzungseinheit definiert und in allen anderen Übersetzungseinheiten (mindestens einmal) deklariert, aber nicht erneut definiert werden (one definition rule, ODR).
- Deshalb dürfen Dateien, die mit `#include` in mehrere Quelldateien eingebunden werden, normalerweise nur Deklarationen, aber keine Definitionen von Funktionen und Variablen enthalten.
- Ausnahme: Bei Verwendung des Schlüsselworts `inline` bei jeder Definition darf dieselbe Funktion oder Variable in beliebig vielen Übersetzungseinheiten je einmal (identisch) definiert werden.

# 3 Klassen

## 3.1 Grundprinzip

- *Klassen* sind verallgemeinerte Strukturtypen:
  - Elemente können *privat*, *halböffentlich* oder *öffentlich* (*private*, *protected*, *public*) sein.
  - Objekte können durch *Konstruktoren* initialisiert und durch *Destruktoren* „aufgeräumt“ werden.
  - Objekte können durch *Elementfunktionen* inspiziert und manipuliert werden.
  - Das Kopieren von Objekten kann durch spezielle Elementfunktionen (*Kopier-* und *Verschiebekonstruktor*, *Kopier-* und *Verschiebezuweisung*) kontrolliert werden.
- Formal ist eine Struktur (`struct`) eine Klasse (`class`), deren Elemente (und Basisklassen, vgl. § 4) standardmäßig öffentlich sind.
- Daher wird in den folgenden Beispielen der Einfachheit halber immer `struct` statt `class` verwendet.

# 3.2 Konstruktoren

## 3.2.1 Beispiel: Rationale Zahlen

```
// Definition:
struct Rational {
  // Datenelemente (numerator, denominator).
  int num, den;

  // Konstruktoren.
  Rational () { num = 0; den = 1; }
  Rational (int n) { num = n; den = 1; }
  Rational (int n, int d) { num = n; den = d; }
};

// Verwendung:
Rational r1 = Rational{};
Rational r2 = Rational{2};
Rational r3 = Rational{1, 2};

// Oder:
Rational r1;
Rational r2 = 2;
```

```
// Oder:
Rational r1{};
Rational r2{2};
Rational r3{1, 2};
```

### 3.2.2 Erläuterungen

- Ein Konstruktor besitzt denselben Namen wie seine Klasse bzw. Struktur.
- Ebenso wie (Element-)Funktionen, können Konstruktoren *überladen* werden.
- Ein Konstruktor „konstruiert“ ein Objekt seines Typs, indem er seine Datenelemente geeignet *initialisiert*. Er kümmert sich *nicht* um die eigentliche *Erzeugung* des Objekts, d. h. um die Bereitstellung *seines* Speicherplatzes.
- Unter Umständen erzeugt ein Konstruktor im Rahmen der Initialisierung *seiner* Datenelemente *weitere* Objekte, für deren Initialisierung *ihrerseits* Konstruktoren aufgerufen werden.
- Konstruktoren können *explizit* oder *implizit* aufgerufen werden:
  - Besitzt ein Typ `T` einen Konstruktor, der ohne Argumente aufgerufen werden kann (entweder, weil er keine Parameter besitzt, oder weil alle Parameter Default-Argumente besitzen), so wird er automatisch bei jeder Deklaration einer Variablen mit Typ `T` aufgerufen, sofern die Variable nicht explizit initialisiert wird.
  - Besitzt `T` einen Konstruktor, der mit einem Argument eines beliebigen Typs `U` aufgerufen werden kann, so wird er bei Bedarf automatisch aufgerufen, um ein Objekt mit Typ `U` in ein Objekt mit Typ `T` umzuwandeln (*implizite Typumwandlung*). Um dies zu verhindern, kann der Konstruktor mit dem Schlüsselwort `explicit` deklariert werden.

- Wenn ein Typ keine Konstruktordeklaration enthält, besitzt er automatisch einen leeren parameterlosen Konstruktor.
  Um dies zu verhindern, könnte man diesen Konstruktor explizit als gelöscht definieren:

```
struct X {
  X () = delete;
  ......
};
```

- Damit dieser parameterlose Standardkonstruktor trotz anderer Konstruktoren automatisch definiert wird, könnte man ihn wie folgt definieren:

```
struct Y {
  Y (int y) { ...... }
  Y () = default;
};
```

- Seit C++11 können für Konstruktoraufrufe, wie im Beispiel, geschweifte Klammern als einheitliche Syntax für Initialisierungen aller Art verwendet werden.

- Anstelle der blauen geschweiften Klammern können aber auch runde Klammern verwendet werden. (Ebenso in den nachfolgenden Beispielen.)
  Ausnahme: `Rational r1();` deklariert `r1` als parameterlose Funktion mit Resultattyp `Rational`.

## 3.2.3 Verwendung von Elementinitialisierern und Initialisierungsausdrücken

### Beispiele

```
struct Rational {
  // Konstante Datenelemente.
  const int num, den;

  // Konstruktoren mit Elementinitialisierern.
  Rational () : num{0}, den{1} {}
  Rational (int n) : num{n}, den{1} {}
  Rational (int n, int d) : num{n}, den{d} {}
};

struct Rational {
  // Konstante Datenelemente mit Initialisierungsausdrücken.
  const int num = 0, den = 1;

  // Konstruktoren mit unvollständigen Elementinitialisierern.
  Rational () {}
  Rational (int n) : num{n} {}
  Rational (int n, int d) : num{n}, den{d} {}
};
```

## Erläuterungen

- Elementinitialisierer stellen Konstruktoraufrufe für die Datenelemente des Typs dar, die vor der Ausführung des Konstruktorrumpfs ausgeführt werden (in der Reihenfolge, in der die Datenelemente deklariert wurden, die von der Reihenfolge der Elementinitialisierer abweichen könnte).

- Wenn ein Konstruktor keinen Elementinitialisierer für ein bestimmtes Datenelement aufruft, aber das Element einen Initialisierungsausdruck besitzt, wird es mit dem Wert dieses Ausdrucks initialisiert.
  Andernfalls wird für dieses Element je nach Typ entweder sein parameterloser Konstruktor ausgeführt (den es in diesem Fall geben muss!), oder das Element bleibt uninitialisiert.

- Durch die explizite Verwendung von Elementinitialisierern kann diese eventuelle Standardinitialisierung von Datenelementen vermieden werden.

- Falls ein Datenelement keinen parameterlosen Konstruktor besitzt, *muss* es entweder mit einem Elementinitialisierer initialisiert werden oder einen Initialisierungsausdruck besitzen.

- Dasselbe gilt für Referenzelemente und konstante Datenelemente, weil ihnen nur auf diese Weise Werte zugeordnet werden können.

## Beispiel: Paare rationaler Zahlen

```
struct RationalPair {
  // Datenelemente.
  Rational x, y;

  // Konstruktoren.
  RationalPair (Rational x, Rational y) : x{x}, y{y} {}
  RationalPair (int a, int b, int c, int d) : x{a, b}, y{c, d} {}
};
```

### 3.2.4 Aufruf eines anderen Konstruktors

- Anstelle von Elementinitialisierern, kann ein Konstruktor einen Aufruf eines anderen Konstruktors desselben Typs enthalten.
- Die daraus resultierenden Aufrufbeziehungen zwischen den Konstruktoren eines Typs dürfen keinen Zyklus bilden.

## Beispiel

```
struct Rational {
  // Konstante Datenelemente.
  const int num, den;

  // Konstruktoren.
  Rational () : Rational{0} {}
  Rational (int n) : Rational{n, 1} {}
  Rational (int n, int d) : num{n}, den{d} {}
};
```

### 3.2.5 Separate Deklaration und Definition von Konstruktoren

- Einfache Konstruktoren können direkt in ihrer Typdefinition definiert (d. h. implementiert) werden. Sie sind in diesem Fall automatisch `inline` deklariert (vgl. § 2.12.2).

- Kompliziertere Konstruktoren werden in der Typdefinition meist nur *deklariert* und später (unter Beachtung der Regeln in § 2.12.2) separat *definiert*.

- Dies ist insbesondere dann sinnvoll, wenn die Typdefinition in einer separaten Definitionsdatei (*header file*) steht.

## Beispiel

```
// Typdefinition.
struct Rational {
  // Konstante Datenelemente.
  const int num, den;

  // Deklaration der Konstruktoren.
  Rational ();
  Rational (int n);
  Rational (int n, int d);
};

......

// Definition der Konstruktoren.
Rational::Rational () : num{0}, den{1} {}
Rational::Rational (int n) : num{n}, den{1} {}
Rational::Rational (int n, int d) : num{n}, den{d} {}
```

## 3.2.6 Beispiel: Dynamische Zeichenketten

```
struct String {
  // Datenelement: Dynamisch erzeugte Reihe von Zeichen
  // mit abschließendem Nullzeichen.
  char* str;

  // Hilfsfunktion: str mit der dynamisch erzeugten Verkettung
  // von s1 und ggf. s2 initialisieren.
  void init (const char* s1, const char* s2 = nullptr) {
    int len = strlen(s1);
    if (s2) len += strlen(s2);
    str = new char [len + 1];
    strcpy(str, s1);
    if (s2) strcat(str, s2);
  }

  // Konstruktor: Erzeugt und füllt die dynamische Reihe str.
  String (const char* s1 = "", const char* s2 = nullptr) {
    init(s1, s2);
  }
```

```
  // Elementfunktion:
  // Alle Kleinbuchstaben in str in Großbuchstaben umwandeln.
  void toupper () {
    // Achtung: In einer Funktion mit einem bestimmten Namen
    // sind andere Bedeutungen dieses Namens verdeckt.
    // Deshalb muss der Name der Bibliotheksfunktion toupper
    // mit std:: qualifiziert werden.
    for (char* s = str; *s; s++) *s = std::toupper(*s);
  }
};

// Globale Funktion: Verkettung der Zeichenketten s1 und s2.
String concat (const String& s1, const String& s2) {
  return String(s1.str, s2.str);
}

// Globale Funktion: Zeichenkette s ausgeben.
void print (const String& s) {
  cout << s.str << endl;
}
```

```
// Testprogramm.
int main (int argc, char* argv []) {
  // Initialisierung von s1 und s2 durch Aufrufe des Konstruktors
  // von String mit jeweils einem Argument.
  String s1 = argv[1];
  String s2 = argv[2];

  // Initialisierung des Resultats von concat durch Aufruf des
  // Konstruktors von String mit zwei Argumenten.
  String s = concat(s1, s2);

  // Kleinbuchstaben in s durch Großbuchstaben ersetzen.
  s.toupper();

  // s ausgeben.
  print(s);
}
```

# 3.3 Destruktoren

## 3.3.1 Problem

- Der Speicher, der vom Konstruktor von `String` beschafft wird, wird nicht wieder freigegeben.

## 3.3.2 Lösung

```
struct String {
  // Wie bisher.
  ......

  // Destruktor: Vernichtet die dynamische Reihe str,
  // die vom zugehörigen Konstruktoraufruf erzeugt wurde.
  ~String () {
    delete [] str;
  }
};
```

## Erläuterungen

- Ein Destruktor ist eine parameterlose (Pseudo-)Elementfunktion, deren Name aus einer Tilde und dem Namen des Typs besteht.

- Ein Destruktor „zerstört“ ein Objekt seines Typs, indem er ggf. erforderliche *Aufräumarbeiten* ausführt. Er kümmert sich *nicht* um die eigentliche *Vernichtung* des Objekts, d. h. um die Freigabe *seines* Speicherplatzes.

- Unter Umständen vernichtet ein Destruktor im Rahmen *seiner* Aufräumarbeiten jedoch *andere* Objekte, die typischerweise von einem Konstruktor seines Typs dynamisch erzeugt wurden.

- Wenn ein Typ keine Destruktordeklaration enthält, besitzt er automatisch einen leeren Destruktor.

- Destruktoren werden in aller Regel *implizit* am Ende der Lebensdauer eines Objekts aufgerufen (vgl. § 3.3.3).

### 3.3.3 Ausführung von Konstruktoren und Destruktoren

- Variablen, die global oder in einem Namensbereich (namespace) definiert sind, sowie statische Elementvariablen von Klassen existieren während der gesamten Programmausführung.
  - Ihr Konstruktor wird entweder irgendwann vor der Ausführung von `main` ausgeführt oder irgendwann, bevor die Variable zum ersten Mal verwendet wird.
  - Für Variablen innerhalb derselben Übersetzungseinheit, die nicht `inline` definiert sind (vgl. § 2.12.2), werden die Konstruktoren in der Reihenfolge der Variablendefinitionen ausgeführt.
  - Für Variablen in verschiedenen Übersetzungseinheiten sowie `inline`-Variablen ist die Reihenfolge nicht festgelegt.
  - Wichtige Ausnahme seit C++20 für Variablen, die nicht `inline` sind:
    Wenn ein Modul ein anderes Modul importiert, werden die Variablen des importierten Moduls garantiert initialisiert, bevor die Variablen des Moduls selbst initialisiert werden, deren Definition sich nach der Import-Deklaration befindet.
  - Die Destruktoren solcher Variablen werden nach der Ausführung von `main` ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.
  - Die Destruktoren werden auch dann noch ausgeführt, wenn das Programm vorzeitig durch Aufruf der Bibliotheksfunktion `exit` beendet wird.

- Wenn derartige Variablen jedoch `thread_local` definiert sind, existieren sie je einmal während der Ausführung jedes Threads.
  - Ihr Konstruktor wird dann entweder vor der Ausführung der „Hauptfunktion“ des jeweiligen Threads ausgeführt oder bevor die Variable zum ersten Mal in diesem Thread verwendet wird.
  - Die Destruktoren solcher Variablen werden am Ende dieses Threads (nach der Ausführung seiner Hauptfunktion) ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.
  - Die Destruktoren werden auch dann noch ausgeführt, wenn dieser Thread `exit` ausführt. (Für die Variablen anderer Threads werden jedoch keine Destruktoren ausgeführt.)
- Variablen, die lokal in einem Block definiert sind, existieren während der Ausführung dieses Blocks.
  - Ihr Konstruktor wird bei der Ausführung ihrer Definition ausgeführt.
  - Die Destruktoren solcher Variablen werden am Ende des Blocks ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.
  - Die Destruktoren werden auch dann noch ausgeführt, wenn der Block vorzeitig mittels einer Sprunganweisung (`break`, `continue`, `return`, `goto`) oder aufgrund einer Ausnahme beendet wird, aber nicht, wenn das Programm mit `exit` beendet wird.

- Wenn derartige Variablen jedoch `static` oder `thread_local` definiert sind, existieren sie entweder während der gesamten Programmausführung oder je einmal während der Ausführung jedes Threads.
  - Ihr Konstruktor wird dann bei der ersten Ausführung ihrer Definition während der Programmausführung bzw. während der Ausführung des jeweiligen Threads ausgeführt.
  - Die Destruktoren solcher Variablen werden nach der Ausführung von `main` bzw. am Ende jedes Threads ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.
- Parameter einer Funktion existieren während der Ausführung dieser Funktion.
  - Ihre Konstruktoren werden beim Aufruf der Funktion (in einer nicht festgelegten Reihenfolge) ausgeführt.
  - Ihre Destruktoren werden am Ende der Funktion ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.

- Objekte, die dynamisch mittels `new` erzeugt werden, existieren bis zu ihrer Vernichtung mittels `delete`.
  - Ihr Konstruktor wird nach der Beschaffung ihres Speicherplatzes durch `new` ausgeführt.
  - Ihr Destruktor wird vor der Freigabe ihres Speicherplatzes durch `delete` ausgeführt.
- Elemente einer Reihe existieren genauso lange wie die Reihe.
  - Ihre Konstruktoren werden bei der Initialisierung der Reihe ausgeführt, und zwar in der Reihenfolge aufsteigender Indizes.
  - Ihre Destruktoren werden bei der Vernichtung der Reihe ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.

- Elementvariablen eines Klassenobjekts existieren genauso lange wie das Klassenobjekt.
  - Ihre Konstruktoren werden bei der Initialisierung des Klassenobjekts durch die Elementinitialisierer (oder die in § 3.2.3 genannten Alternativen) von dessen Konstruktor ausgeführt, und zwar in der Reihenfolge der Deklaration der Elementvariablen.
  - Ihre Destruktoren werden bei der Vernichtung des Klassenobjekts nach der Ausführung von dessen Destruktor ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.

- Damit die obigen Aussagen z. B. auch für elementare Typen wie `int` korrekt sind, besitzen auch solche Typen formal einen parameterlosen Konstruktor sowie einen Destruktor, die beide leer sind.

## 3.3.4 Ressourcenverwaltung durch Konstruktoren und Destruktoren

- Da Destruktoren immer automatisch „im richtigen Moment“ aufgerufen werden, können sie gezielt zur Freigabe von Ressourcen eingesetzt werden, die zuvor (meist in einem zugehörigen Konstruktor) beschafft wurden (resource acquisition/allocation is initialization, RAII).
- Da Anweisungsfolgen auf unterschiedliche Art – manchmal auch unerwartet – vorzeitig beendet werden können, ist dies einfacher und sicherer als eine manuelle Freigabe der Ressourcen am Ende einer Anweisungsfolge.

### Beispiel

```
#include <mutex>
#include <fstream>
#include <memory>
using namespace std;

// Sperre für gegenseitigen Ausschluss (mutual exclusion).
mutex m;

void test () {
  // Der Konstruktor von lock_guard sperrt das übergebene Mutex.
  lock_guard<mutex> g {m};
```

```
  // Der Konstruktor von ifstream öffnet die Datei mit dem
  // übergebenen Namen.
  ifstream f {"input.txt"};

  // Der Konstruktor von unique_ptr speichert den übergebenen
  // Zeiger auf ein dynamisches Objekt oder eine dynamische Reihe.
  unique_ptr<char []> p {new char [100]};

  // Hier könnte die Funktion vorzeitig verlassen werden,
  // z. B. durch return oder eine unerwartete Ausnahme.
  ......

  // Der automatisch aufgerufene Destruktor von unique_ptr gibt
  // das dynamische Objekt oder die dynamische Reihe auf jeden Fall
  // wieder frei.

  // Der automatisch aufgerufene Destruktor von ifstream
  // schließt die Datei auf jeden Fall wieder.

  // Der automatisch aufgerufene Destruktor von lock_guard
  // gibt die Sperre auf jeden Fall wieder frei.
}
```

## 3.3.5 Sichere Initialisierung nicht-lokaler Variablen

- Gemäß § 3.3.3 ist die Initialisierungsreihenfolge von Variablen, die global oder in einem Namensbereich definiert sind, sowie von statischen Elementvariablen in verschiedenen Übersetzungseinheiten nicht festgelegt.

- Deshalb ist das Verhalten eines Programms undefiniert, in dem für die Initialisierung einer derartigen Variablen eine andere derartige Variable aus einer anderen Übersetzungseinheit benötigt wird.

- Dieses Problem kann umgangen werden, indem man anstelle einer Variablen eine Funktion definiert und verwendet, die eine Referenz auf eine lokale statische Variable liefert, zum Beispiel:

```
// Statt einer globalen Variablen ...
String path = "/bin:/usr/bin";

// ... lieber eine globale Funktion.
String& path () {
  static String path = "/bin:/usr/bin";
  return path;
}
```

- Die lokale statische Variable wird garantiert beim ersten Aufruf der Funktion initialisiert.

- Um die Variable `path` zu verwenden, muss dann jeweils `path()` geschrieben werden. Da die Funktion eine Referenz zurückliefert, sind auch Zuweisungen `path() = ...` möglich.

- Alternativ könnte man die Definitionen (nicht unbedingt die Deklarationen) aller nicht-lokalen Variablen in einer sinnvollen Reihenfolge in eine einzige Übersetzungseinheit schreiben.

- Wie bereits in § 3.3.3 erwähnt, lösen Module in C++20 das hier beschriebene Problem, indem sie garantieren, dass die aus anderen Modulen importierten Variablen bereits initialisiert sind, wenn die eigenen Variablen eines Moduls (deren Definition sich nach den Import-Deklarationen befindet) initialisiert werden.

# 3.4 Kopierfunktionen

## 3.4.1 Problem

```
int main (int argc, char* argv []) {
  // Initialisierung von s1 durch Aufruf des Konstruktors
  // von String.
  String s1 = argv[1];

  // Initialisierung von s2 als elementweise Kopie von s1.
  String s2 = s1;

  // Kleinbuchstaben in s1 in Großbuchstaben umwandlen.
  s1.toupper();

  // s1 und s2 ausgeben.
  print(s1);
  print(s2);

  // Automatischer Aufruf des Destruktors von String für s2 und s1.
}
```

- Weil das Objekt `s2` als elementweise „flache“ Kopie von `s1` konstruiert wird, zeigen `s1.str` und `s2.str` auf *dieselbe* dynamische Reihe.

- Damit werden durch den Aufruf `s1.toupper()` auch die Kleinbuchstaben in `s2` in Großbuchstaben umgewandelt, was vermutlich nicht beabsichtigt ist.

- Außerdem wird diese Reihe am Ende des Blocks fälschlicherweise *zweimal* freigegeben – was zu undefiniertem Programmverhalten führt –, weil sowohl für `s2` als auch für `s1` der Destruktor von `String` aufgerufen wird.

### 3.4.2 Lösung: Kopierkonstruktor

```
struct String {
  // Wie zuvor.
  ......

  // Kopierkonstruktor: Erzeugt eine "tiefe" Kopie von that.
  String (const String& that) {
    init(that.str);
  }
};
```

## Erläuterungen

- Ein Kopierkonstruktor eines Typs `T` ist ein Konstruktor mit einem Parameter des Typs `const T&` oder `T&`. (Eventuell vorhandene weitere Parameter müssten Default-Argumente besitzen.)

- Er wird implizit aufgerufen, wenn ein Objekt des Typs `T` durch ein Objekt desselben Typs initialisiert wird, was auch bei der Übergabe eines Objekts als Parameter oder bei der Rückgabe eines Funktionsresultats der Fall ist. (Deshalb darf der Kopierkonstruktor selbst keinen Parameter mit Typ `T` besitzen, weil er sonst bei der Übergabe dieses Parameters ebenfalls aufgerufen werden müsste.)

- Vermeidbare Aufrufe des Kopierkonstruktors dürfen jedoch vom Übersetzer eliminiert werden.

- Außerdem kann man selbst Aufrufe des Kopierkonstruktors vermeiden, indem man Parameter als Referenzen (normalerweise auf Konstanten) deklariert (vgl. § 2.9.6).

- Wenn für einen Typ kein expliziter Kopierkonstruktor definiert ist, besitzt er automatisch einen impliziten Kopierkonstruktor, der alle Elementvariablen des Typs kopiert. Hierfür werden ggf. die Kopierkonstruktoren der Elementvariablen aufgerufen.

- Falls dies aus irgendeinem Grund nicht möglich ist (z. B. weil einer der benötigten Kopierkonstruktoren gelöscht ist), wird der implizite Kopierkonstruktor jedoch als gelöscht definiert. Dann können Objekte des Typs grundsätzlich nicht kopiert werden.

- Wenn für den Typ eine Verschiebefunktion definiert ist (vgl. § 3.5), wird der implizite Kopierkonstruktor ebenfalls als gelöscht definiert.

- Außerdem kann der Kopierkonstruktor auch explizit als gelöscht definiert werden, wenn Objekte des Typs aus irgendeinem Grund nicht kopiert werden sollen (vgl. § 3.4.4).

## 3.4.3 Weiteres Problem

```
int main (int argc, char* argv []) {
  // Initialisierung von s1 und s2 durch Aufrufe des Konstruktors
  // von String.
  String s1 = argv[1];
  String s2 = argv[2];

  // Elementweise Zuweisung von s1 an s2.
  s2 = s1;

  ......

  // Automatischer Aufruf des Destruktors von String für s2 und s1.
}
```

- Durch die elementweise „flache“ Zuweisung von `s1` an `s2` wird `s2.str` durch `s1.str` überschrieben.
- Am Ende des Blocks werden die Destruktoren für `s2` und `s1` aufgerufen, die jetzt beide dieselbe Reihe `s1.str` vernichten (wollen), während die ursprüngliche Reihe `s2.str` für immer als „Speicherleiche“ übrig bleibt.

## 3.4.4 Lösung: Kopierzuweisung

```
struct String {
  // Wie zuvor.
  ......

  // Kopierzuweisung: Vernichtet die eigene Reihe
  // und erstellt eine "tiefe" Kopie der Reihe von that.
  String& operator= (const String& that) {
    // Vorsicht: Die alte Reihe erst vernichten, wenn die neue
    // erfolgreich erzeugt und gefüllt wurde, weil das Erzeugen
    // prinzipiell fehlschlagen kann und weil die alte eventuell
    // noch zum Füllen der neuen gebraucht wird (wenn ein Objekt
    // an sich selbst zugewiesen wird).
    char* s = str;
    init(that.str);
    delete [] s;

    // Selbstreferenz zurückliefern.
    return *this;
  }
};
```

## Erläuterungen

- Eine Kopierzuweisung (oder ein kopierender Zuweisungsoperator) eines Typs `T` ist ein überladener Operator = mit einem Parameter des Typs `const T&`, `T&` oder `T`, der anhand der üblichen Regeln für überladene Operatoren aufgerufen wird.

  (Daraus folgt nebenbei, dass bei Klassen auch R-Werte Ziel einer Zuweisung sein könnten, was eigentlich widersinnig ist! Um dies zu verhindern, kann der Zuweisungsoperator seit C++11 mit `&` qualifiziert werden; vgl. § 3.6.
  Selbst eine Zuweisung an ein konstantes Objekt wäre prinzipiell denkbar, wenn der Zuweisungsoperator mit `const` qualifiziert ist; vgl. ebenfalls § 3.6.)

- Im Gegensatz zu vielen anderen Programmiersprachen, besteht in C++ ein wesentlicher Unterschied zwischen der *Initialisierung* eines Objekts (durch einen Konstruktor) und der *Zuweisung* an ein Objekt (durch einen Zuweisungsoperator).

- Da das Zielobjekt im einen Fall uninitialisiert und im anderen Fall bereits initialisiert ist, müssen Konstruktoren und Zuweisungsoperatoren normalerweise unterschiedliche Anweisungen ausführen.

- Häufig enthält ein Zuweisungsoperator sowohl Teile eines Destruktors als auch Teile eines Konstruktors.

- Wenn die Zuweisung eines Objekts an sich selbst nicht korrekt funktionieren würde, muss dieser Sonderfall abgefangen werden: `if (this != &that) ......`
- Ein Zuweisungsoperator liefert üblicherweise eine Referenz auf das Zielobjekt zurück.
- Um das Kopieren von Objekten eines Typs `T` zu verbieten (z. B. bei Streams, Threads und Mutexes), kann man den Kopierkonstruktor und die Kopierzuweisung von `T` als gelöscht definieren.
- Wenn für einen Typ keine explizite Kopierzuweisung definiert ist, besitzt er automatisch einen impliziten Zuweisungsoperator, der für alle Elementvariablen des Typs eine Kopierzuweisung ausführt. Hierfür werden ggf. die entsprechenden Zuweisungsoperatoren der Elementvariablen aufgerufen.
- Falls dies aus irgendeinem Grund nicht möglich ist (z. B. weil einer der benötigten Zuweisungsoperatoren gelöscht ist oder weil der Typ konstante Elementvariablen enthält), wird der implizite Zuweisungsoperator jedoch als gelöscht definiert.
- Wenn für den Typ eine Verschiebefunktion definiert ist (vgl. § 3.5), wird der implizite Zuweisungsoperator ebenfalls als gelöscht definiert.

# 3.5 Verschiebefunktionen

## 3.5.1 Problem

```
// Alle Kleinbuchstaben in einer Kopie von s
// in Großbuchstaben umwandeln.
String toupper (String s) { // Init. von s durch Kopierkonstruktor.
  s.toupper();
  return s;
}                           // Automatischer Aufruf des Destruktors für s.

int main () {
  // Initialisierung von s durch Aufruf des Konstruktors von String.
  String s = argv[1];

  // Übergabe von s an toupper durch einen (unvermeidbaren) Aufruf
  // des Kopierkonstruktors.
  // Rückgabe des Resultats von toupper in ein temporäres Objekt
  // ebenfalls durch einen Aufruf des Kopierkonstruktors.
  // Zuweisung dieses temporären Objekts an s durch einen Aufruf
  // der Kopierzuweisung.
  s = toupper(s);
}
```

- Das von `toupper` gelieferte Objekt wird hier zweimal unnötig kopiert:
  - Zuerst wird es durch den Kopierkonstruktor in ein temporäres Objekt kopiert.
  - Anschließend wird dieses temporäre Objekt durch die Kopierzuweisung nach `s` kopiert.

## 3.5.2 Lösung: Verschiebekonstruktor und -zuweisung

```
struct String {
  // Wie zuvor.
  ......

  // Verschiebekonstruktor:
  // "Klaut" die Daten von that und hinterlässt that
  // in irgendeinem neuen wohldefinierten Zustand.
  String (String&& that) {
    str = that.str;
    that.init("");
  }

  // Verschiebezuweisung:
  // Vertauscht die Daten von that mit den eigenen und hinterlässt
  // that damit wiederum in einem neuen wohldefinierten Zustand.
  String& operator= (String&& that) {
    char* tmp = str; str = that.str; that.str = tmp;
    return *this;
  }
};
```

## Erläuterungen

- Verschiebekonstruktor und -zuweisung eines Typs `T` sind analog zu Kopierkonstruktor und -zuweisung, allerdings mit einem Parameter des Typs `T&&`. (Alternativ könnte der Parametertyp auch `const T&&` sein, was aber meist nicht sinnvoll ist, weil das übergebene Objekt normalerweise verändert werden soll.)

- Wenn sie vorhanden sind, werden sie vom Übersetzer anstelle von Kopierkonstruktor bzw. -zuweisung verwendet, sofern das zu kopierende bzw. zuzuweisende Objekt ein R-Wert ist (weil ihr R-Wert-Referenz-Parameter nur mit R-Werten initialisiert werden kann, vgl. § 2.9.5; wenn es ein L-Wert ist, werden nach wie vor Kopierkonstruktor bzw. -zuweisung verwendet).

- Weil das an den Parameter gebundene R-Wert-Objekt (normalerweise) „im nächsten Moment“ nicht mehr existieren wird, ist es unnötig, seine Daten zu kopieren. Stattdessen kann man sie einfach „klauen“, d. h. in das Zielobjekt *verschieben*.

- Damit der später für das „bestohlene“ Objekt ausgeführte Destruktor korrekt funktioniert und keinen „Schaden anrichtet“, muss das Objekt aber in einem wohldefinierten neuen Zustand hinterlassen werden, der nichts mit seinem alten Zustand gemeinsam hat.
  Außerdem sollten anschließende (Kopier- und Verschiebe-) Zuweisungen an dieses Objekt noch korrekt funktionieren (vgl. das Beispiel in § 3.5.3).

- Bei der Verschiebezuweisung lässt sich das am einfachsten durch Vertauschen der Daten mit denen des Zielobjekts erreichen (was auch im Sonderfall einer Selbstzuweisung korrekt funktionieren würde).

- Beim Verschiebekonstruktor hinterlässt man das übergebene Objekt häufig im gleichen Zustand wie ein durch den parameterlosen Standardkonstruktor initialisiertes Objekt.

- Im Beispiel von § 3.5.1 wird das temporäre Objekt jetzt durch einen Aufruf des Verschiebekonstruktors anstelle des Kopierkonstruktors mit dem Resultat von `topupper` initialisiert (vgl. die Anmerkungen am Ende von § 3.5.3).
  Für die Zuweisung dieses temporären Objekts an `s` wird jetzt die Verschiebezuweisung anstelle der Kopierzuweisung verwendet, weil die rechte Seite `toupper(s)` ein R-Wert ist.
  Damit sind die beiden „teuren“ Kopieroperationen jeweils durch „billige“ Verschiebeoperationen ersetzt.

- Im Beispiel von § 3.4.3 wird bei der Zuweisung von `s1` an `s2` jedoch weiterhin die Kopierzuweisung aufgerufen – was auch sinnvoll ist –, weil das zuzuweisende Objekt `s2` ein L-Wert ist.
  Aus dem gleichen Grund wird im Beispiel von § 3.4.1 bei der Initialisierung von `s2` durch `s1` weiterhin der Kopierkonstruktor aufgerufen, was ebenfalls sinnvoll ist.

- Auch für Typen wie Streams, Threads und Mutexes, deren Objekte nicht sinnvoll kopiert werden können (vgl. § 3.4.4), lässt sich normalerweise eine sinnvolle Verschiebesemantik implementieren, die z. B. notwendig ist, um derartige Objekte in Containern wie z. B. `vector` speichern zu können.

- Wenn für einen Typ weder Verschiebekonstruktor und -zuweisung noch Kopierkonstruktor und -zuweisung noch Destruktor explizit definiert sind, werden Verschiebekonstruktor und -zuweisung implizit definiert.

- Analog zu implizit definierten Kopierfunktionen, führen diese impliziten Verschiebefunktionen die entsprechende Funktion jeweils für alle Elementvariablen des Typs aus.

- Wenn dies aus irgendeinem Grund nicht möglich ist, wird die entsprechende Funktion jedoch als gelöscht definiert.

## 3.5.3 Verschiebung von L-Werten

### Beispiel

```
int main () {
  // Initialisierung von s1 durch Aufruf des Konstruktors.
  String s1 = argv[1];

  // Initialisierung von s2 durch Aufruf des Kopierkonstruktors.
  String s2 = s1;

  // Initialisierung von s3 durch Aufruf des Verschiebekonstruktors.
  String s3 = static_cast<String&&>(s1);

  // Oder besser so:
  String s3 = std::move(s1);

  // Aber nicht so:
  String s3 = static_cast<String>(s1);

  // Obwohl der Inhalt von s1 nach s3 verschoben wurde,
  // könnte s1 hier immer noch verwendet werden.
}
```

## Erläuterungen

- Um eine Verschiebung anstelle einer Kopie eines L-Werts zu erzwingen – weil der Inhalt des Objekts anschließend nicht mehr benötigt wird –, kann der L-Wert in eine R-Wert-Referenz umgewandelt werden.

- Aus Gründen der Lesbarkeit wird hierfür üblicherweise die Bibliotheksfunktion `move` verwendet, die selbst keinerlei Daten verschiebt, sondern lediglich diese Typumwandlung vornimmt.

- Eine Umwandlung eines L-Werts in seinen eigenen Typ (ohne Referenz) liefert zwar auch einen R-Wert, der aber selbst durch einen Aufruf des Kopierkonstruktors erzeugt wird!

## Beispiel: Vertauschen

```
void swap (T& x, T& y) {
  // Ungünstig wegen Verw.      // Besser wegen Verwendung von
  // von Kopierfunktionen:     // Verschiebefkt. (falls vorhd.):
  T z = x;                     T z = move(x);
  x = y;                       x = move(y);
  y = z;                       y = move(z);
}
```

## Rückgabe lokaler Variablen und Parameter

- Wenn eine lokale Variable (aber kein Parameter) als Funktionsresultat geliefert wird (z. B. `return x`), kann der hierfür theoretisch erforderliche Aufruf des Kopierkonstruktors vom Übersetzer häufig eliminiert werden (sofern die Variable nicht `static` oder `thread_local` ist).

- Deshalb ist die Verwendung von `move` (d. h. `return move(x)`) in solchen Fällen unnötig und oft sogar kontraproduktiv, weil der Ausdruck `move(x)` keine lokale Variable mehr ist und der dann notwendige Aufruf des Verschiebekonstruktors deshalb *nicht* eliminiert werden darf.

- Außerdem werden sowohl lokale Variablen (sofern sie nicht `static` oder `thread_local` sind) als auch Parameter einer Funktion, die als Funktionsresultat geliefert werden, hier ausnahmsweise als R-Werte betrachtet, sodass ein vorhandener Verschiebekonstruktor sowieso automatisch anstelle des Kopierkonstruktors verwendet wird, wenn der Aufruf nicht eliminiert werden kann.

# 3.6 Elementfunktionen (member functions)

- Nichtstatische Elementfunktionen einschließlich Konstruktoren und Destruktor einer Klasse `C` besitzen einen impliziten Parameter `this` mit Typ `C*`, der auf das *Zielobjekt* des Funktionsaufrufs verweist.

- Die Verwendung eines Datenelements `m` der Klasse in einer Elementfunktion ist äquivalent zu `this->m`. Ebenso ist die Verwendung einer anderen Elementfunktion `f` der Klasse äquivalent zu `this->f`.

- Wenn die Deklaration einer Elementfunktion nach der Parameterliste das Schlüsselwort `const` enthält, ist `this` vom Typ `const C*`, d. h. die Funktion darf ihr Zielobjekt nicht verändern. (Analog für `volatile`.)

- Wenn das Zielobjekt eines Funktionsaufrufs konstant ist, *muss* die aufgerufene Elementfunktion `const` sein.
  Andererseits darf eine `const`-Elementfunktion auch für nicht-konstante Zielobjekte aufgerufen werden (vgl. § 2.4.2).
  Daher ist es ratsam, `const` bei der Deklaration von Elementfunktionen so oft wie möglich zu verwenden.

- Elementfunktionen einer Klasse können so überladen werden, dass sie sich nur durch die An- bzw. Abwesenheit von `const` unterscheiden.
  Bei einem Aufruf einer solchen Funktion wird für ein konstantes Zielobjekt die `const`-Funktion, für ein nicht-konstantes Zielobjekt die Nicht-`const`-Funktion verwendet.

- Seit C++11 kann nach der Parameterliste einer Elementfunktion auch `&` oder `&&` stehen, um anzuzeigen, dass das Zielobjekt ein L- bzw. R-Wert sein muss.

- Statische Elementfunktionen sind vergleichbar mit globalen Funktionen, müssen aber beim Aufruf außerhalb der Klasse mit dem Klassennamen qualifiziert werden.
  Sie besitzen keinen impliziten Parameter `this`.

- Datenelemente und Elementfunktionen müssen unterschiedliche Namen besitzen.
  (In Java darf eine Methode den gleichen Namen wie eine Objekt- oder Klassenvariable besitzen.)

## Beispiel

```
// Klasse.
struct User {
  // Datenelemente.
  string username, password;

  // Konstruktor.
  User (const string& un) : username{un} {}

  // Konstante Elementfunktion.
  string get_username () const { return username; }

  // Nicht-konstante Elementfunktion.
  void set_password (const string& pw) { password = pw; }

  // Statische Elementfunktion.
  static bool equal (const User& u1, const User& u2) {
    // Da u1 und u2 konstant sind, muss get_username konstant sein.
    return u1.get_username() == u2.get_username();
  }
};
```

```
// Verwendung der unterschiedlichen Elementfunktionen.
User u{"Dummy"};
cout << u.get_username() << endl;
u.set_password("ymmuD");
if (User::equal(u, u)) ......
```

# 3.7 Statische Datenelemente

- Statische Datenelemente sind vergleichbar mit globalen Variablen, müssen aber außerhalb der Klasse ebenfalls mit dem Klassennamen qualifiziert werden.

- Statische Datenelemente können normalerweise in ihrer Klasse nur *deklariert* werden und müssen zusätzlich außerhalb der Klasse (unter Beachtung der Regeln für Variablen in § 2.12.2) *definiert* werden (sofern sie irgendwo im Programm verwendet werden). Ein eventueller Initialisierungsausdruck wird bei der Definition angegeben.

- Ausnahme: Wenn ein statisches Datenelement konstant ist und einen ganzzahligen oder Aufzählungstyp besitzt, kann es direkt bei seiner Deklaration in der Klasse mit einem *konstanten Ausdruck* initialisiert werden.
  Solange dann nur sein *Wert* (und nicht seine Adresse) verwendet wird, muss keine zusätzliche Definition angegeben werden. (In diesem Fall muss der Übersetzer auch keinen Speicherplatz anlegen.)
  Andernfalls darf bei der Definition kein Initialisierungsausdruck mehr angegeben werden.

- Anstelle von statischen Datenelementen können auch statische Elementfunktionen verwendet werden, die eine Referenz auf eine lokale statische Variable liefern (vgl. § 3.3.5).

## Beispiel

```
struct X {
  // Deklaration statischer Datenelemente.
  static X* p;
  static X* q;
  static int x;
  static const int y = 2;
};

// Definition von X::p und X::x.
X* X::p = nullptr;
int X::x = 1;
```

```
int main () {
  // Verwendung von X::p und X::x.
  X::p = new X;
  cout << X::x << endl;
  cout << &X::x << endl;

  // X::q wird nirgends verwendet
  // und braucht deshalb keine Definition.

  // Der Wert von X::y kann ohne Definition verwendet werden.
  cout << X::y << endl;

  // Fehler: Die Adresse von X::y kann ohne Definition nicht
  // verwendet werden.
  cout << &X::y << endl;

  // Fehler: Um X::y an eine Referenz zu binden, wird implizit
  // ebenfalls seine Adresse verwendet.
  int& r = X::y;
}
```

# 4 Abgeleitete Klassen (derived classes)

## 4.1 Konzept

- Bei der Definition einer Klasse kann man eine oder mehrere *Basisklassen* (base classes) angeben, von denen die neue Klasse *abgeleitet* wird.
- Zusätzlich zu ihren eigenen Elementen, besitzt die abgeleitete Klasse automatisch alle Datenelemente und Elementfunktionen ihrer Basisklassen („Vererbung“).
- Jeder Konstruktor der abgeleiteten Klasse ruft entweder explizit (durch entsprechende Elementinitialisierer) oder implizit Konstruktoren aller Basisklassen auf, bevor die eigenen Elemente der Klasse initialisiert werden.
- Die Reihenfolge dieser Konstruktoraufrufe entspricht der *Reihenfolge der Basisklassen* in der Deklaration der abgeleiteten Klasse, die von der Reihenfolge der Elementinitialisierer abweichen könnte.
- Der Destruktor der abgeleiteten Klasse ruft nach Ausführung seines Rumpfs implizit die Destruktoren aller Elemente und Basisklassen auf.
- Die Reihenfolge dieser Destruktoraufrufe entspricht der *umgekehrten Reihenfolge* der entsprechenden Konstruktoraufrufe.

## Beispiel

```
struct Person {           // Person.
  string name;            // Name.
  Person (string n) : name{n} {}
};

struct Student : Person {       // Student.
  int number;                   // Matrikelnummer.
  Student (string n, int m) : Person{n}, number{m} {}
};

struct Employee : Person {      // Arbeitnehmer.
  int number;                   // Personalnummer.
  Employee (string n, int p) : Person{n}, number{p} {}
};

struct EmployedStudent : Student, Employee { // Teilzeitstudent.
  EmployedStudent (string n, int m, int p)
                          : Student{n, m}, Employee{n, p} {}
};
```

## Erläuterungen

- Der Konstruktor von `Student` ruft den Konstruktor von `Person` auf (der wiederum den Konstruktor von `string` aufruft), bevor er das Datenelement `number` initialisiert. Der Konstruktor von `string` beschafft dynamischen Speicherplatz zur Speicherung des Namens.

- Der implizit definierte Destruktor von `Student` ruft den (ebenfalls implizit definierten) Destruktor von `Person` auf, der wiederum den Destruktor von `string` aufruft. Dieser gibt den dynamischen Speicherplatz frei, den sein Konstruktor zur Speicherung des Namens beschafft hat.

- Analoges gilt für den Konstruktor und Destruktor von `Employee`.

- Der Konstruktor von `EmployedStudent` ruft nacheinander die Konstruktoren von `Student` und `Employee` auf, die ihrerseits beide den Konstruktor von `Person` aufrufen etc. (Tatsächlich besitzt ein `EmployedStudent`-Objekt zwei unabhängige `Person`-Teilobjekte, was vermutlich nicht erwünscht ist; vgl. § 4.4.1 und § 4.4.2.)

# 4.2 Untertyp-Polymorphie

- Ein Objekt einer abgeleiteten Klasse (bzw. ein Zeiger oder eine Referenz auf ein solches Objekt) kann überall verwendet werden, wo ein Objekt einer (eindeutigen und zugänglichen) Basisklasse (bzw. ein Zeiger oder eine Referenz auf ein solches Objekt) erwartet wird, d. h. bei Bedarf findet eine entsprechende implizite *Aufwärtsumwandlung* (up-cast) statt.

- Umgekehrt kann ein Zeiger (bzw. eine Referenz) auf ein Objekt einer (nicht-virtuellen) Basisklasse explizit in einen Zeiger (bzw. eine Referenz) auf ein Objekt einer abgeleiteten Klasse umgewandelt werden (*Abwärtsumwandlung*, down-cast), sofern das referenzierte Objekt tatsächlich zu dieser Klasse gehört. (Andernfalls ist das Verhalten undefiniert.)

- Wenn die Basisklasse *polymorph* (im Sinne der C++-Sprachdefinition) ist, d. h. wenn sie *virtuelle Elementfunktionen* besitzt (vgl. § 4.5), kann zur Laufzeit überprüft werden, ob ein Objekt zur abgeleiteten Klasse gehört (*dynamischer Typtest*, dynamic cast, vgl. `instanceof` in Java).

## Beispiel

```
// Name und ggf. Matrikelnummer von p ausgeben.
void print (const Person& p, bool stud = false) {
  cout << "Name: " << p.name << endl;
  if (stud) {
    const Student& s = static_cast<const Student&>(p);
    cout << "Matrikelnummer: " << s.number << endl;
  }
}

// Aufruf von print mit einem Person-Objekt.
Person p{"Maier"};
print(p);

// Aufruf von print mit einem Student-Objekt.
Student s{"Maier", 123};
print(s, true);

// Aufruf von print mit einem EmployedStudent-Objekt schlägt fehl,
// da Person keine eindeutige Basisklasse ist (vgl. §4.4).
EmployedStudent es{"Maier", 123, 456};
print(es, true); // Fehler!
```

# 4.3 Namenskonflikte

- Elemente einer abgeleiteten Klasse *verbergen* gleichnamige Elemente von Basisklassen.
- Gleichnamige Elemente aus unterschiedlichen Basisklassen führen zu *Mehrdeutigkeiten* in der abgeleiteten Klasse.
- In beiden Fällen können Elementnamen durch ihren Klassennamen *qualifiziert* werden, um den Zugriff zu ermöglichen bzw. eindeutig zu machen.

**Beispiel**

```
cout << "Matrikelnummer: " << es.Student::number << endl;
cout << "Personalnummer: " << es.Employee::number << endl;
```

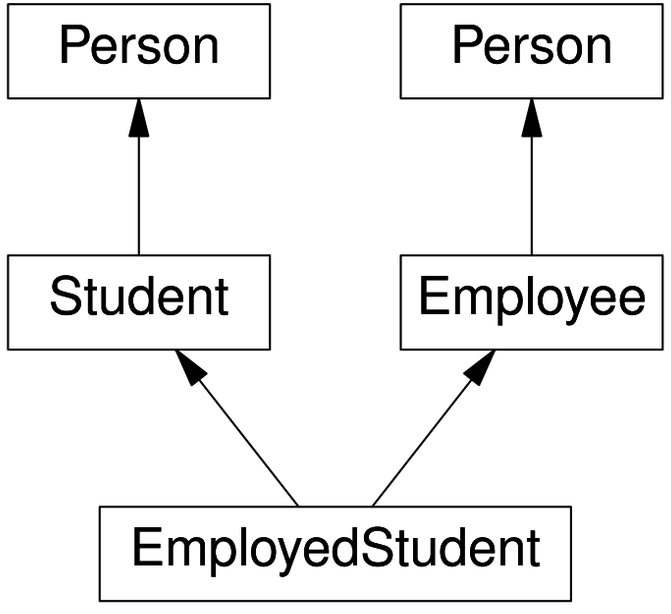
# 4.4 Replizierte und virtuelle Basisklassen

## 4.4.1 Replizierte Basisklassen

- Die *direkten* Basisklassen einer Klasse müssen paarweise verschieden sein.
- Die Menge der *indirekten* Basisklassen kann Klassen jedoch mehrfach enthalten (*replizierte Basisklassen*).
- In diesem Fall enthält ein Objekt der abgeleiteten Klasse mehrere *Teilobjekte* der entsprechenden Basisklassen, und die Elemente dieser Basisklassen sind zwangsläufig *mehrdeutig* (sofern sie nicht durch andere Elemente verborgen sind).
- Um die Elemente verwenden zu können, muss man ihren Namen mit dem Namen einer *eindeutigen* Zwischenklasse qualifizieren.

**Beispiel**

```
cout << es.Student::name << endl;
cout << es.Employee::name << endl;
```




V-förmige Vererbungsstruktur

## 4.4.2 Virtuelle Basisklassen

- Um die Replikation einer Basisklasse zu vermeiden, muss sie überall als *virtuelle Basisklasse* vereinbart werden.

- Wenn eine Klasse (direkte oder indirekte) virtuelle Basisklassen besitzt, ruft jeder Konstruktor dieser Klasse als erstes (explizit oder implizit) Konstruktoren der virtuellen Basisklassen auf, bevor Konstruktoren der (übrigen) direkten Basisklassen und Konstruktoren der eigenen Elemente aufgerufen werden.

- Die Reihenfolge der Konstruktoraufrufe für die virtuellen Basisklassen ergibt sich durch eine *Tiefensuche* im Ableitungsgraphen, wobei auf jeder Ebene die Deklarationsreihenfolge der Basisklassen berücksichtigt wird.

- Konstruktoraufrufe für die virtuellen Basisklassen werden nur von einem Konstruktor der *abgeleitetsten Klasse* (*most derived class*), d. h. vom Konstruktor eines *eigenständigen Objekts* ausgeführt.
  Entsprechende Aufrufe von Zwischenklassen-Konstruktoren werden *ignoriert*.

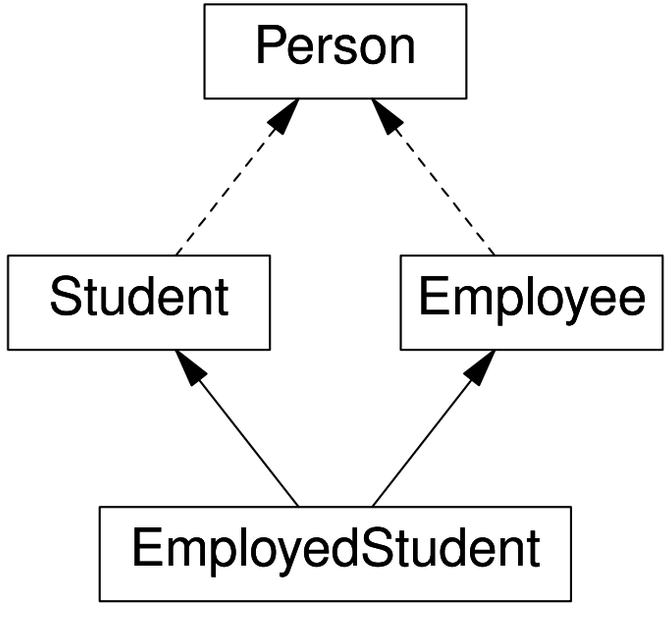
**Beispiel**

```
struct Person {                 // Person.
  string name;                  // Name.
  Person (string n) : name{n} {}
};

struct Student : virtual Person {       // Student.
  int number;                           // Matrikelnummer.
  Student (string n, int m) : Person{n}, number{m} {}
};
```

```
struct Employee : virtual Person {       // Arbeitnehmer.
  int number;                            // Personalnummer.
  Employee (string n, int p) : Person{n}, number{p} {}
};

struct EmployedStudent : Student, Employee { // Teilzeitstudent.
  EmployedStudent (string n, int m, int p)
                  : Person{n}, Student{n, m}, Employee{n, p} {}
}
```




Rautenförmige Vererbungsstruktur (diamond inheritance)

## Erläuterungen

- Die Konstruktoren (und Destruktoren) von `Student` und `Employee` funktionieren wie zuvor.

- Der Konstruktor von `EmployedStudent` ruft als erstes den Konstruktor der (indirekten) virtuellen Basisklasse `Person` auf (der seinerseits den Konstruktor von `string` aufruft), bevor er die Konstruktoren der direkten Basisklassen `Student` und `Employee` aufruft.

- Obwohl diese Konstruktoren Aufrufe des Konstruktors von `Person` enthalten (müssen), werden diese hier ignoriert.
  (Sie werden nur ausgeführt, wenn ein *eigenständiges* `Student`- bzw. `Employee`-Objekt konstruiert wird.)

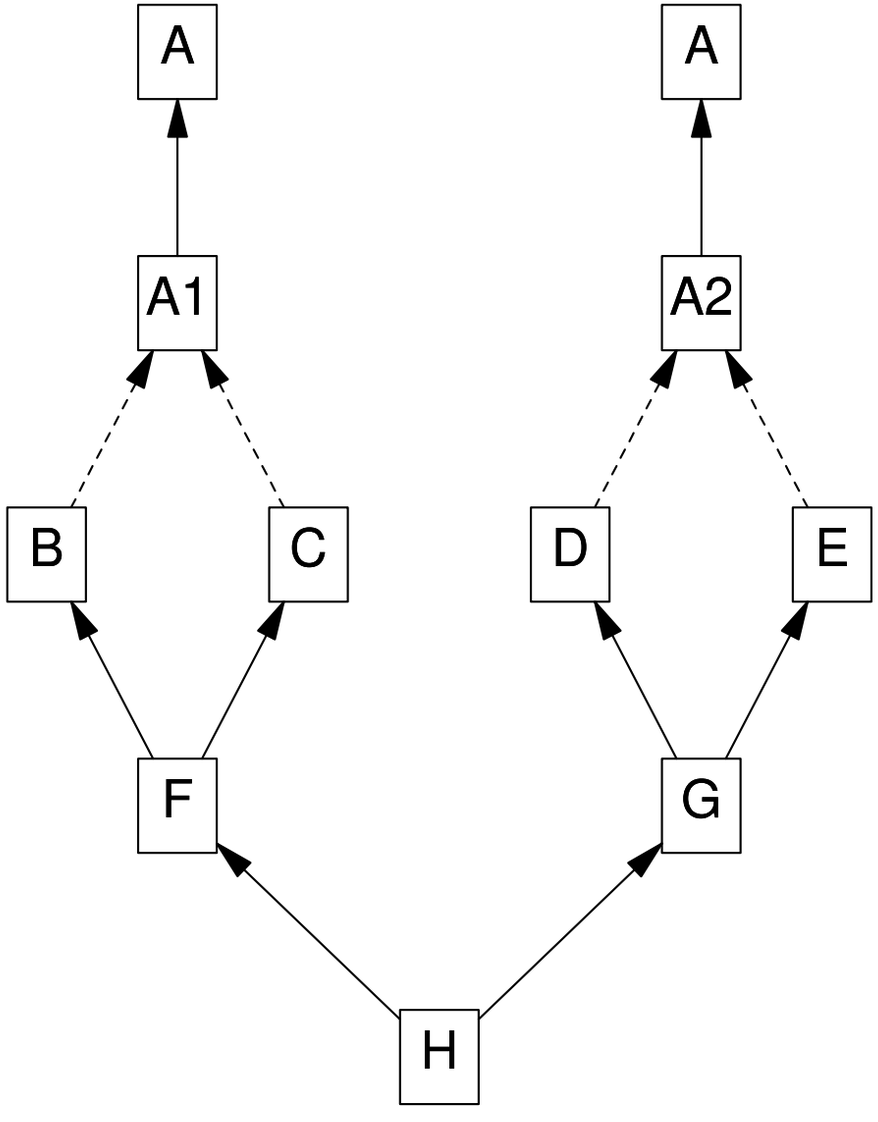
- Der (in diesem Beispiel implizite) Destruktor von `EmployedStudent` ruft (nach der Ausführung seines Rumpfs) die Destruktoren von `Employee`, `Student` und `Person` in dieser Reihenfolge auf.

## 4.4.3 Kombination virtueller und replizierter Basisklassen

- Wenn eine Klasse sowohl als virtuelle als auch als nicht-virtuelle Basisklasse auftritt, werden alle virtuellen Vorkommen zu einem gemeinsamen Teilobjekt zusammengefasst, während jedes nicht-virtuelle Vorkommen zu einem zusätzlichen Teilobjekt führt.

- Um mehrere Teilobjekte einer Klasse *teilweise* gemeinsam zu nutzen, muss man *von Anfang an* künstliche Zwischenklassen einführen.

# Beispiel



## 4.4.4 Anmerkungen

- Das Vererbungskonzept von C++ zeigt, dass mehrfache Vererbung prinzipiell sprachlich beherrschbar ist.
- Replizierte Basisklassen ergeben sich technisch als Normalfall, obwohl sie in Anwendungen eher den Ausnahmefall darstellen.

- Virtuelle Basisklassen haben gewisse konzeptuelle Mängel:
  - Statische Abwärtsumwandlungen (vgl. § 4.2) von einer virtuellen Basisklasse (oder auch von einer ihrer Basisklassen) zu einer abgeleiteten Klasse sind nicht möglich.
  - Virtuelle Basisklassen einer Klasse können vor abgeleiteten Klassen nicht verborgen werden, weil jeder Konstruktor einer abgeleiteten Klasse Konstruktoren der virtuellen Basisklassen aufrufen muss.
    Dies verletzt das Modularitäts- und Geheimnisprinzip.
    (Beispielsweise müssten abgeleitete Klassen von `EmployedStudent` eigentlich nicht wissen, dass `Person` eine virtuelle Basisklasse dieser Klasse ist.)
  - Die Tatsache, dass Konstruktoraufrufe für virtuelle Basisklassen in Zwischenklassen-Konstruktoren ignoriert werden, ist verwirrend.
  - Die Entscheidung für eine virtuelle oder nicht-virtuelle Basisklasse muss an einer Stelle getroffen werden, wo sie zunächst noch keine Auswirkung hat.
    (Für `Student` und `Employee` besteht kein Unterschied, ob `Person` eine virtuelle oder nicht-virtuelle Basisklasse ist. Erst für `EmployedStudent` ist der Unterschied relevant.)
  - Replizierte virtuelle Basisklassen sind nicht direkt möglich.
    Dies verletzt wiederum das Modularitäts- und Geheimnisprinzip, weil man bei der Verwendung einer Basisklasse deren „Vorgeschichte“ kennen muss.
  - Um replizierte virtuelle Basisklassen indirekt zu erreichen, muss man – logisch wiederum an der „falschen“ Stelle – künstliche Zwischenklassen einführen.

# 4.5 Virtuelle Elementfunktionen

## 4.5.1 Konzept

- Wenn eine nichtstatische Elementfunktion `virtual` deklariert ist – und nur dann! –, kann sie in abgeleiteten Klassen *redefiniert* werden (Überschreiben).

- Die *Parametertypen* sowie eventuelle `const`- und `volatile`-*Qualifizierer* einer Redefinition müssen *exakt* mit denen der ursprünglichen Definition übereinstimmen; andernfalls wird eine *andere* gleichnamige Funktion definiert (Überladen).

- Durch das Wort `override` nach der Parameterliste kann explizit zum Ausdruck gebracht werden, dass diese Funktion eine Redefinition darstellt. Wenn es dann keine passende ursprüngliche Definition gibt, erhält man eine Fehlermeldung.

- Durch das Wort `final` nach der Parameterliste kann eine (weitere) Redefinition der Funktion in abgeleiteten Klassen verboten werden.

- Der *Resultattyp* einer Redefinition darf *kovariant* vom Typ der ursprünglichen Definition abweichen: Wenn der ursprüngliche Resultattyp ein Zeiger oder eine Referenz auf eine Klasse ist, darf der Resultattyp der Redefinition ein Zeiger bzw. eine Referenz auf eine davon abgeleitete Klasse sein.

- Eine virtuelle Elementfunktion kann mehrere geerbte virtuelle Funktionen gleichzeitig redefinieren, wenn diese alle den gleichen Namen, die gleichen Parametertypen und die gleichen Qualifizierer besitzen.
  Wenn mehrere dieser geerbten Funktionen selbst (direkte oder indirekte) Redefinitionen derselben ursprünglichen Funktion sind (was bei rautenförmiger Vererbung möglich ist), dann *müssen* sie am „Fußpunkt“ der Raute auf diese Weise gemeinsam redefiniert werden. (Diese Redefinition wird dann als „final overrider“ bezeichnet.)

- Beim Aufruf einer virtuellen Elementfunktion wird zur Laufzeit anhand des *dynamischen Typs* des Zielobjekts die passende Überschreibung der Funktion ausgewählt (*dynamisches Binden*).

- Wenn der Funktionsname beim Aufruf durch einen Klassennamen qualifiziert wird, wird der Aufruf jedoch *statisch* gebunden. Auf diese Weise kann eine Redefinition eine frühere Definition aufrufen (vgl. `super` in Java).

- Wenn eine virtuelle Elementfunktion in einer Klasse nur deklariert und dann außerhalb der Klasse definiert/implementiert wird, darf das Schlüsselwort `virtual` nur bei der Deklaration angegeben werden.

- Virtuelle Elementfunktionen und virtuelle Basisklassen sind voneinander unabhängige Konzepte.

## Beispiel 1

```
// Basisklasse Person.
struct Person {
  string name; // Name.
  Person (string n) : name{n} {}

  // Virtuelle Elementfunktion.
  virtual void print () const {
    cout << "Name: " << name << endl;
  }
};
```

```
// Abgeleitete Klasse Student.
struct Student : virtual Person {
  int number; // Matrikelnummer.
  Student (string n, int m) : Person{n}, number{m} {}

  // Redefinition der virtuellen Funktion.
  virtual void print () const {
    Person::print(); // Aufruf der ursprünglichen Funktion.
    cout << "Matrikelnummer: " << number << endl;
  }
};

// Abgeleitete Klasse Employee.
struct Employee : virtual Person {
  int number; // Personalnummer.
  Employee (string n, int p) : Person{n}, number{p} {}

  // Redefinition der virtuellen Funktion.
  virtual void print () const {
    Person::print(); // Aufruf der ursprünglichen Funktion.
    cout << "Personalnummer: " << number << endl;
  }
};
```

```
// Mehrfach abgeleitete Klasse EmployedStudent.
struct EmployedStudent : Student, Employee {
  EmployedStudent (string n, int m, int p)
                  : Person{n}, Student{n, m}, Employee{n, p} {}

  // Redefinition der mehrfach geerbten virtuellen Funktion.
  virtual void print () const {
    Student::print(); // Aufruf der Funktion von Student.
    // Aufruf von Employee::print
    // würde den Namen noch einmal ausgeben.
    cout << "Personalnummer: " << Employee::number << endl;
  }
};
```

```
int main () {
  // Person.
  Person* p = new Person{...};

  // Polymorphe Verwendung von Student als Person.
  Person* s = new Student{...};

  // Polymorphe Verwendung von EmployedStudent als Student.
  Student* es = new EmployedStudent{...};

  // Aufrufe der Elementfunktion print.
  // Ausgeführte Funktion, wenn print virtual bzw. non-virtual ist:
                          // virtual                  | non-virtual
  p->print();             // Person::print            | Person::print
  s->print();             // Student::print           | Person::print
  es->print();            // EmployedStudent::print   | Student::print
  es->Student::print(); // Student::print             | Student::print
}
```

## Beispiel 2

```
struct A {
  virtual void f1 ()          { cout << "A::f1" << endl; }
  virtual void f2 ()          { cout << "A::f2" << endl; }
  virtual void f3 ()          { cout << "A::f3" << endl; }
};

struct B : virtual A {
  virtual void f2 () override { cout << "B::f2" << endl; }
  virtual void f3 () override { cout << "B::f3" << endl; }
  virtual void f4 ()          { cout << "B::f4" << endl; }
};

struct C : virtual A {
  virtual void f3 () override { cout << "C::f3" << endl; }
  virtual void f4 ()          { cout << "C::f4" << endl; }
};
```

```
struct D : B, C {
  // f3 braucht in D einen "final overrider",
  // f1, f2 und f4 jedoch nicht.
  virtual void f3 () override { cout << "D::f3" << endl; }
};

int main () {
  D* d = new D;
  d->f1();       // A::f1
  d->f2();       // B::f2
  d->f3();       // D::f3
  d->f4();       // Mehrdeutig!
  d->B::f4();    // B::f4
  d->C::f4();    // C::f4
}
```

## 4.5.2 Polymorphe und abstrakte Klassen

- Eine Klasse, die virtuelle Elementfunktionen besitzt (oder erbt), heißt *polymorph* und kann in dynamischen Typtests (`dynamic_cast`) verwendet werden.
- Eine Klasse heißt *abstrakt*, wenn sie *rein virtuelle* Funktionen besitzt, d. h. virtuelle Elementfunktionen, die durch den Zusatz `=0` am Ende ihrer Deklaration gekennzeichnet sind.
- Von einer abstrakten Klasse können keine Objekte erzeugt werden.

## Beispiel

```
// Abstrakte Basisklasse: Arithmetischer Ausdruck.
struct Expr {
  // Rein virtuelle Elementfunktion.
  virtual double eval () const = 0;
};

// Konkrete abgeleitete Klasse: Konstanter Ausdruck.
struct Const : Expr {
  const double val; // Wert des konstanten Ausdrucks.
  Const (double v) : val{v} {}

  // (Re)definition der virtuellen Funktion eval.
  virtual double eval () const {
    return val;
  }
};
```

```
// Abstrakte abgeleitete Klasse: Binärer Ausdruck.
struct Binary : Expr {
  Expr* const left;  // Linker und rechter Teilausdruck
  Expr* const right; // des binären Ausdrucks.
  Binary (Expr* l, Expr* r) : left{l}, right{r} {}

  // Die virtuelle Funktion eval bleibt rein virtuell.
};

// Konkrete abgeleitete Klasse: Addition.
struct Add : Binary {
  Add (Expr* l, Expr* r) : Binary{l, r} {}

  // (Re)definition der virtuellen Funktion eval.
  virtual double eval () const {
    return left->eval() + right->eval();
  }
};

// Analog für Subtraktion, Multiplikation und Division.
......
```

```
int main () {
  Expr* x = new Add{new Const{1}, new Const{2}};

  // Aufruf der virtuellen Elementfunktion.
  cout << x->eval() << endl;

  // Dynamische Typtests.
  if (dynamic_cast<Const*>(x)) { // Nein.
    cout << "constant expression" << endl;
  }
  if (dynamic_cast<Binary*>(x)) { // Ja.
    cout << "binary expression" << endl;
  }
  if (Add* a = dynamic_cast<Add*>(x)) { // Ja.
    cout << "left: " << a->left->eval() << endl;
    cout << "right: " << a->right->eval() << endl;
  }
}
```

## 4.5.3 Dynamische Typtests und -umwandlungen

- Für einen Zeiger `x` auf ein Objekt eines polymorphen Typs `X` und einen Zeigertyp `T` mit Zieltyp `Y` ist `dynamic_cast<T>(x)` *erfolgreich*, wenn eine der folgenden Bedingungen erfüllt ist:
  - Das Objekt `*x` ist ein öffentliches Teilobjekt eines eindeutigen Objekts des Typs `Y` (Abwärtsumwandlung, down-cast).
  - Das Objekt `*x` ist ein öffentliches Teilobjekt seines Gesamtobjekts (most derived object), und dieses besitzt ein eindeutiges öffentliches Teilobjekt des Typs `Y` (Querumwandlung, cross-cast).
- Im Erfolgsfall liefert `dynamic_cast<T>(x)` einen Zeiger auf das genannte Objekt des Typs `Y`, andernfalls einen Nullzeiger. Wenn `x` ein Nullzeiger ist, ist das Resultat ebenfalls ein Nullzeiger.
- Wenn `Y` gleich `void` ist, ist `dynamic_cast<T>(x)` immer erfolgreich und liefert einen Zeiger auf das Gesamtobjekt, zu dem das Objekt `*x` gehört.
- Wenn der Typ `X` `const`- und/oder `volatile`-qualifiziert ist, muss der Typ `Y` ebenso qualifiziert sein, aber nicht umgekehrt.

- Wenn `Y` eine Basisklasse von `X` ist, ist `dynamic_cast<T>(x)` eigentlich ein `static_cast`, der genau dann erfolgreich ist, wenn `Y` eindeutig und zugänglich ist.
  Da dies bereits zur Übersetzungszeit überprüft wird, erhält man ggf. eine entsprechende Fehlermeldung.
  In diesem Fall muss der Typ `X` nicht polymorph sein.

- Wenn `Y` gleich `X` ist, ist `dynamic_cast<T>(x)` immer erfolgreich und liefert einfach `x`.
  Auch in diesem Fall muss der Typ `X` nicht polymorph sein.

- Für einen L-Wert `x` des Typs `X` und einen L-Wert-Referenztyp `T` mit Zieltyp `Y` bzw. für einen beliebigen Wert `x` des Typs `X` und einen R-Wert-Referenztyp `T` mit Zieltyp `Y` ist `dynamic_cast<T>(x)` jeweils analog wie für Zeigertypen definiert.
  Im Erfolgsfall erhält man eine L- bzw. R-Wert-Referenz auf das genannte Objekt des Typs `Y`, andernfalls wird eine Ausnahme des Typs `bad_cast` geworfen.

- Die Begriffe „öffentlich“ und „zugänglich“ werden in § 4.7 definiert.

## Beispiel

```
struct A {
  int a;
  virtual ~A () {} // Virtueller Destruktor, damit A polymorph ist.
};
struct B {
  int b;
};
struct C {
  int c;
  virtual ~C () {} // Virtueller Destruktor, damit C polymorph ist.
};

struct D : A, B, virtual C {};

D* d = new D;
A* a = d;                          // Implizite Aufwärtsumwandlung.

d = dynamic_cast<D*>(a);           // Erfolgreiche Abwärtsumwandlung.
B* b = dynamic_cast<B*>(a);        // Erfolgreiche Querumwandlung.
C* c = dynamic_cast<C*>(a);        // Erfolgreiche Querumwandlung.
d = dynamic_cast<D*>(c);           // Erfolgreiche Abwärtsumwandlung.
```

## Fortsetzung des Beispiels

```
struct D1 : D {};
struct D2 : D {};
struct DD : D1, D2 {};

DD* dd = new DD;
a = (D2*)dd;          // Explizite und implizite Aufwärtsumwandlung.

d = dynamic_cast<D*>(a);  // Erfolgreiche Abwärtsumwandlung.
b = dynamic_cast<B*>(a);  // Nicht erfolgreiche Querumwandlung.
c = dynamic_cast<C*>(a);  // Erfolgreiche Querumwandlung.
d = dynamic_cast<D*>(c);  // Nicht erfolgreiche Abwärtsumwandlung.
```

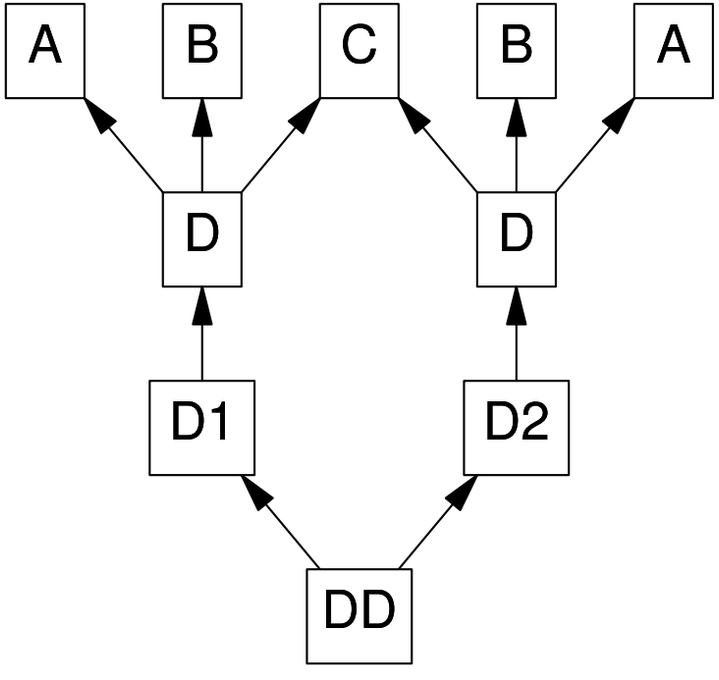
## Erläuterungen

- Die Querumwandlung von `a` nach `B*` ist jetzt nicht erfolgreich, weil das Gesamtobjekt `*dd`, zu dem `*a` gehört, zwei `B`-Teilobjekte enthält und `*a` kein Teilobjekt von einem von ihnen ist.

- Die Querumwandlung von `a` nach `C*` ist jedoch erfolgreich, weil das Gesamtobjekt `*dd` (aufgrund der virtuellen Vererbung) nur ein `C`-Teilobjekt enthält.

- Auch die Abwärtsumwandlung von `a` nach `D*` ist erfolgreich, obwohl das Gesamtobjekt `*dd` zwei `D`-Teilobjekte enthält, weil `*a` ein Teilobjekt von genau einem von ihnen ist.

- Die Abwärtsumwandlung von `c` nach `D*` ist jedoch nicht erfolgreich, weil `*c` ein Teilobjekt von zwei `D`-Teilobjekten ist.



## 4.5.4 Virtuelle Destruktoren

- Wenn mittels `delete` ein Objekt gelöscht wird, dessen dynamischer Typ von seinem statischen Typ abweicht, muss der statische Typ einen *virtuellen Destruktor* besitzen, damit durch dynamisches Binden der Destruktor des dynamischen Typs ausgeführt werden kann.
  (Beim Löschen dynamischer Reihen müssen statischer und dynamischer Typ jedoch immer übereinstimmen.)

- In den Beispielen in § 4.5.3 könnten die dynamischen Objekte `*d` und `*dd` daher auch mittels `delete a` oder `delete c` gelöscht werden (weil die Klassen `A` und `C` jeweils einen virtuellen Destruktor besitzen), aber nicht mittels `delete b` (weil `B` keinen virtuellen Destruktor besitzt).

- Wenn eine Klasse polymorph sein soll, obwohl sie keine virtuellen Elementfunktionen besitzt, kann ihr Destruktor „pro forma“ virtuell definiert werden (vgl. die Beispiele in § 4.5.3).

# 4.6 Vererbung von Konstruktoren

- Wenn eine Klasse die „gleichen“ Konstruktoren wie eine ihrer direkten Basisklassen besitzen soll, können diese mittels `using` geerbt werden.

- Jeder geerbte Konstruktor besitzt die gleichen Parameter wie der entsprechende Konstruktor der Basisklasse und ruft diesen mit den Werten der Parameter auf. Wenn er weitere Konstruktoren (von weiteren direkten Basisklassen, indirekten virtuellen Basisklassen oder eigenen Datenelementen) aufrufen muss, werden diese ohne Parameter aufgerufen. Wenn die zugehörigen Klassen dann keinen entsprechenden Konstruktor besitzen, wird der geerbte Konstruktor gelöscht. (Das heißt, man erhält nur dann eine entsprechende Fehlermeldung, wenn man einen solchen Konstruktor später verwenden will.)

- Das heißt insbesondere: Wenn es indirekte virtuelle Basisklassen gibt, die keine entsprechenden Konstruktoren besitzen, funktioniert die Vererbung von Konstruktoren nicht wie gewünscht.

- Prinzipiell können die Konstruktoren mehrerer Basisklassen geerbt werden sowie zusätzlich weitere Konstruktoren definiert werden.

## Beispiel

```
// Eine beliebige Klasse.
struct A {
  ......
};

// Künstliche Zwischenklassen, damit AA zweimal von A erben kann.
struct A1 : A {
  // A1 erbt die Konstruktoren von A.
  using A::A;
};
struct A2 : A {
  // A2 erbt die Konstruktoren von A.
  using A::A;
};

struct AA : A1, A2 {
  ......
};
```

# 4.7 Zugriffsrechte

## 4.7.1 Private, halböffentliche und öffentliche Elemente und Basisklassen

- Eine Klasse kann mit den Schlüsselwörtern `public`, `proteced` und `private` (jeweils gefolgt von einem Doppelpunkt) in beliebig viele Abschnitte in beliebiger Reihenfolge unterteilt werden, deren Elemente jeweils *öffentlich*, *halböffentlich* bzw. *privat* sind.

- Der Abschnitt vor dem ersten solchen Schlüsselwort ist implizit privat, wenn die Klasse mit dem Schlüsselwort `class` definiert wird, andernfalls (Schlüsselwort `struct` oder `union`) öffentlich.

- Ebenso kann jede direkte Basisklasse einer Klasse öffentlich (Normalfall), halb-öffentlich oder privat deklariert werden.

- Wenn für eine Basisklasse keine Angabe gemacht wird, ist sie implizit privat, wenn die abgeleitete Klasse mit dem Schlüsselwort `class` definiert wird, andernfalls (Schlüsselwort `struct`) öffentlich. (Das heißt: Bei Verwendung des Schlüsselworts `class` muss man normalerweise alle Basisklassen explizit mit dem Schlüsselwort `public` deklarieren.)

## 4.7.2 Zugriffsrechte von Elementen direkter und indirekter Basisklassen

- Die öffentlichen und halböffentlichen Elemente einer öffentlichen direkten Basisklasse sind in der abgeleiteten Klasse ebenfalls öffentlich bzw. halböffentlich.

- Die öffentlichen und halböffentlichen Elemente einer halböffentlichen direkten Basisklasse sind in der abgeleiteten Klasse jeweils halböffentlich.

- Die öffentlichen und halböffentlichen Elemente einer privaten direkten Basisklasse sind in der abgeleiteten Klasse jeweils privat.

- Die privaten Elemente einer beliebigen direkten Basisklasse sind in der abgeleiteten Klasse zwar vorhanden, aber *gesperrt* (oder „eingesperrt“), d. h. nicht direkt zugänglich, ebenso die gesperrten Elemente einer beliebigen direkten Basisklasse.

- Die Elemente einer indirekten Basisklasse werden anhand dieser Regeln schrittweise in die abgeleitete Klasse übernommen, zum Beispiel:
  - Wenn `A` eine öffentliche direkte Basisklasse von `B` und `B` ein private direkte Basisklasse von `C` ist, sind die öffentlichen und halböffentlichen Elemente von `A` in `B` ebenfalls öffentlich bzw. halböffentlich und in `C` privat.
  - Wenn `A` eine private direkte Basisklasse von `B` und `B` ein öffentliche direkte Basisklasse von `C` ist, sind die öffentlichen und halböffentlichen Elemente von `A` in `B` privat und damit in `C` gesperrt.
  - Die privaten und gesperrten Elemente von `A` sind in jedem Fall in `B` und damit auch in `C` gesperrt.
  - Wenn direkte Basisklassen von Klassen immer öffentlich sind (Normalfall), bleiben die Zugriffsrechte von öffentlichen und halböffentlichen Elementen in allen abgeleiteten Klassen unverändert, während private Elemente in abgeleiteten Klassen grundsätzlich immer gesperrt sind.

- Wenn sich für ein Element einer virtuellen Basisklasse auf unterschiedlichen Ableitungswegen unterschiedliche Zugriffsrechte in der abgeleiteten Klasse ergeben, gilt das „Maximum“ dieser Zugriffsrechte, wobei `public` > `protected` > `private` > „gesperrt“ ist.
  Von einem Element einer replizierten Basisklasse gibt es in der abgeleiteten Klasse entsprechend mehrere „Exemplare“, von denen jedes unterschiedliche Zugriffsrechte besitzen kann.

- Eine Deklaration `using B::m` in einem öffentlichen/halböffentlichen/privaten Abschnitt der abgeleiteten Klasse macht das bzw. die Elemente mit dem Namen `m` der Basisklasse `B` jedoch zu entsprechenden Elementen der abgeleiteten Klasse (sofern sie nicht gesperrt sind).

- Ausnahme: Mittels `using B::B` geerbte Konstruktoren (vgl. § 4.6) besitzen in der abgeleiteten Klasse immer die gleichen Zugriffsrechte wie in der Basisklasse `B`.

- Eine indirekte Basisklasse ist öffentlich/halböffentlich/privat/gesperrt, wenn ein gedachtes öffentliches Element dieser Basisklasse in der abgeleiteten Klasse öffentlich/halböffentlich/privat/gesperrt wäre, das heißt:

  a) `B` ist eine öffentliche Basisklasse von `C`, wenn es im Ableitungsgraphen einen Weg von `C` nach `B` gibt, dessen Kanten alle öffentlich sind.

  b) `B` ist eine halböffentliche Basisklasse von `C`, wenn es keinen Weg gemäß a) gibt, aber einen Weg, dessen Kanten alle öffentlich oder halböffentlich sind.

  c) `B` ist eine private Basisklasse von `C`, wenn es keinen Weg gemäß a) und b) gibt, aber einen Weg, bei dem genau die erste Kante privat ist.

  d) Andernfalls (d. h. wenn alle Wege von `C` nach `B` eine private Kante enthalten, die nicht die erste Kante ist) ist `B` eine gesperrte Basisklasse von `C`.

  Wenn `B` eine replizierte Basisklasse von `C` ist, muss jedes „Exemplar“ von `B` separat betrachtet werden.

## 4.7.3 Zugängliche Basisklassen

- Eine öffentliche (direkte oder indirekte) Basisklasse einer Klasse `C` ist überall *zugänglich* (oder *zugreifbar*, engl. *accessible*).
- Eine halböffentliche Basisklasse von `C` ist nur innerhalb der Klasse `C` zugänglich sowie in abgeleiteten Klassen `D`, für die `C` eine öffentliche, halböffentliche oder private (d. h. keine gesperrte) Basisklasse ist.
- Eine private Basisklasse von `C` ist nur innerhalb der Klasse `C` zugänglich.
- Eine gesperrte Basisklasse von `C` ist nirgends zugänglich.
- Außerdem ist eine beliebige Basisklasse `B` von `C` an einer bestimmten Stelle zugänglich, wenn es eine an dieser Stelle zugängliche Basisklasse `Z` von `C` gibt und `B` wiederum eine an dieser Stelle zugängliche Basisklasse von `Z` ist.
- Eine implizite Aufwärtsumwandlung einer Klasse in eine (eindeutige) Basisklasse (vgl. § 4.2) ist an einer bestimmten Stelle nur möglich, wenn die Basisklasse an dieser Stelle zugänglich ist. Dasselbe gilt für explizite Aufwärtsumwandlungen mittels `static_cast`.
- Mit einem „klassischen C-Cast“ ist aber auch eine Aufwärtsumwandlung in eine nicht zugängliche Basisklasse möglich.

### 4.7.4 Zugängliche Elemente

- Ein öffentliches Element einer Klasse `C` ist überall zugänglich.
- Ein halböffentliches Element einer Klasse `C` ist nur innerhalb der Klasse `C` zugänglich sowie in abgeleiteten Klassen `D`, für die `C` eine öffentliche, halböffentliche oder private (d. h. keine gesperrte) Basisklasse ist.
- Ein privates Element einer Klasse `C` ist nur innerhalb der Klasse `C` zugänglich.
- Ein gesperrtes Element einer Klasse `C` ist nirgends zugänglich.
- Außerdem ist ein Element `m` einer Klasse `C` an einer bestimmten Stelle zugänglich, wenn es eine an dieser Stelle zugängliche Basisklasse `Z` von `C` gibt und `m` als Element von `Z` an dieser Stelle zugänglich ist.

### 4.7.5 Verwendung statischer Elemente

- Für ein statisches Element `m` einer Klasse `C` ist die Verwendung `C::m` an einer bestimmten Stelle erlaubt, wenn `m` ein an dieser Stelle zugängliches Element von `C` ist.
- Innerhalb der Klasse `C` ist `m` äquivalent zu `C::m`.

## 4.7.6 Verwendung nichtstatischer Elemente

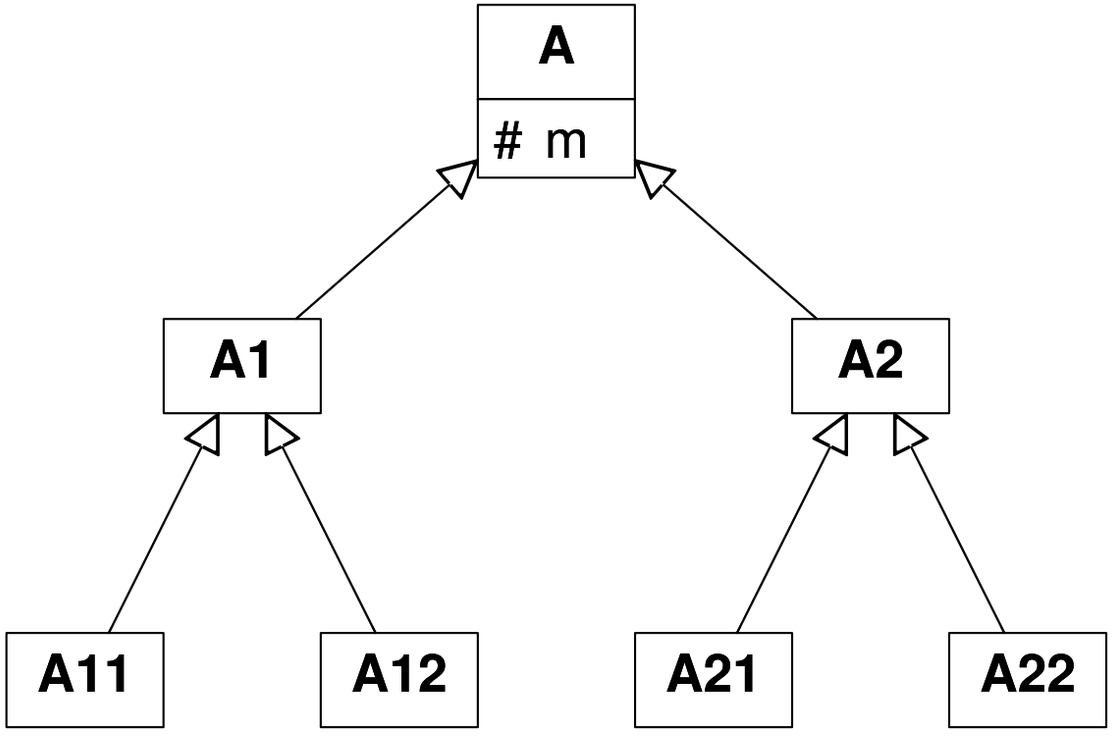
- Für ein nichtstatisches Element `m` einer Klasse `C` sowie ein Objekt `c` dieser Klasse bzw. einen Zeiger `p` auf ein solches Objekt ist die Verwendung `c.m` bzw. `p->m` an einer bestimmten Stelle erlaubt, wenn `m` ein an dieser Stelle zugängliches Element von `C` ist.

- Wenn der Name des Elements `m` mit dem Namen `B` einer Basisklasse qualifiziert ist (`c.B::m` bzw. `p->B::m`), muss sich das Zielobjekt `c` bzw. `p` implizit in diese Basisklasse umwandeln lassen und `m` muss ein an dieser Stelle zugängliches Element von `B` sein.

- Innerhalb einer Klasse ist `m` bzw. `B::m` äquivalent zu `this->m` bzw. `this->B::m`.

- Zusatzregel für halböffentliche nichtstatische Elemente:
  Wenn `m` ein halböffentliches Element von `C` (bzw. `B`) ist, darf es in einer bestimmten Klasse nur für Zielobjekte verwendet werden, die zu dieser Klasse oder einer von ihr abgeleiteten Klasse gehören.

❒ Beispiel:




❍ In der Klasse `A1` darf das halböffentliche nichtstatische Element `m` für Objekte der Klassen `A1...`, aber nicht für Objekte der Klassen `A` und `A2...` verwendet werden.

❍ Die Klassen `A1...` sind gerade diejenigen Klassen, an deren Implementierung die Klasse `A1` „beteiligt“ ist.

## 4.7.7 Befreundete Klassen und Funktionen

- Eine Klasse kann andere Klassen sowie Funktionen aller Art als *Freunde* deklarieren. (Eine Klasse oder Funktion kann sich aber nicht selbst zum Freund einer anderen Klasse machen.)

- Ein Freund einer Klasse besitzt die gleichen „Privilegien“ wie ein Element dieser Klasse, d. h. wenn etwas (ein Element oder eine Basisklasse irgendeiner Klasse) in einer bestimmten Klasse verwendet werden darf, dann darf es auch in den Freunden dieser Klasse verwendet werden. („Darf verwendet werden“ bedeutet aufgrund der in § 4.7.6 genannten Zusatzregel für halböffentliche nichtstatische Elemente u. U. mehr als nur „ist zugänglich“.)

- Ob sich eine Freundschaftsdeklaration in einem öffentlichen, halböffentlichen oder privaten Abschnitt einer Klasse befindet, spielt keine Rolle.

- Freundschaftsbeziehungen werden weder an abgeleitete Klassen vererbt noch sind sie transitiv.

## 4.7.8 Anmerkungen

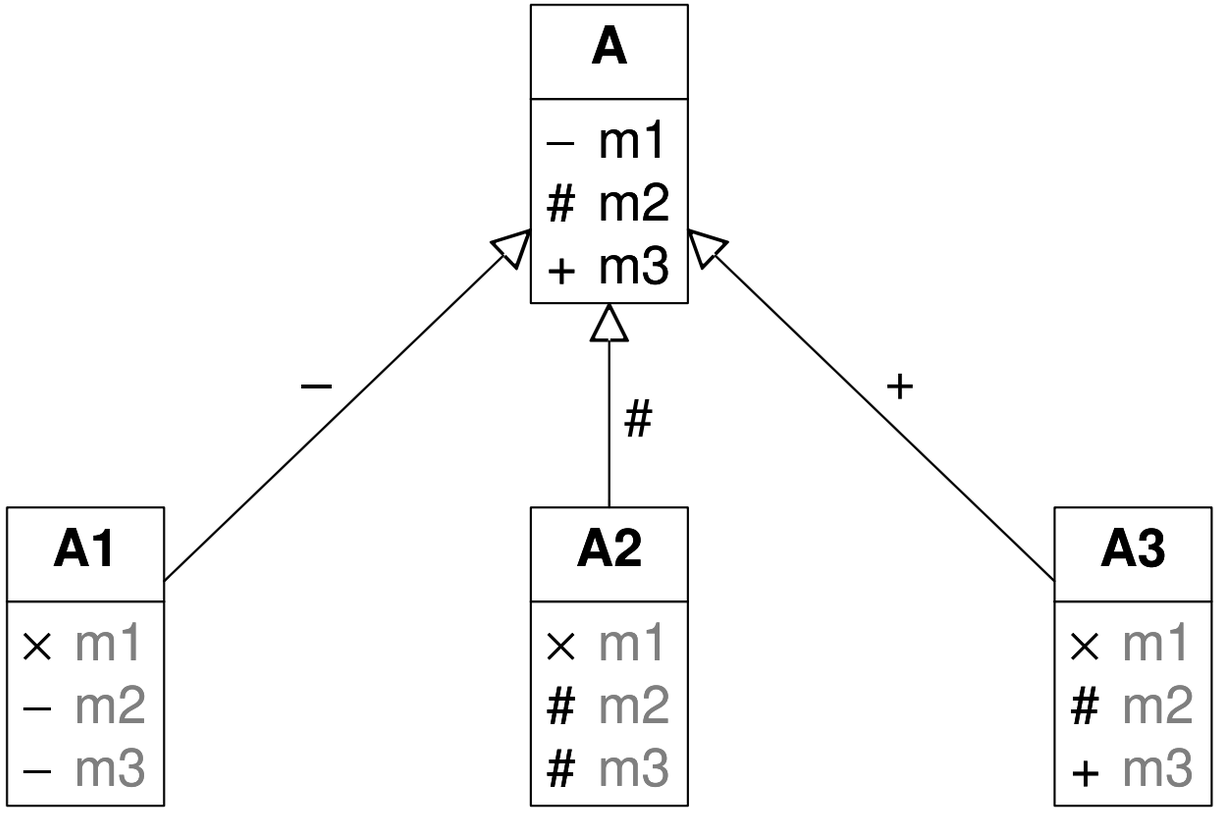
- Nicht zugängliche Elemente einer Klasse sind trotzdem *sichtbar* und werden z. B. bei der Auflösung überladener Funktionen berücksichtigt. Wenn die am besten passende Funktion dann aber nicht zugänglich ist, erhält man eine Fehlermeldung.

- Die Implementierung einer Elementfunktion gehört logisch zu ihrer Klasse, auch wenn sie sich textuell außerhalb der Klassendefinition befindet.

- Die Redefinition einer virtuellen Elementfunktion kann ein anderes Zugriffsrecht haben als die ursprüngliche Definition. Auch private und gesperrte Elementfunktionen können redefiniert werden.

- `c.B::m` bzw. `p->B::m` ist (abgesehen von eventuellen `const`- oder `volatile`-Qualifizierern) bezüglich der Zugriffsrechte äquivalent zu `static_cast<B>(c).m` bzw. `static_cast<B*>(p)->m`.
  Wenn `m` jedoch eine virtuelle Elementfunktion ist, wird der Funktionsaufruf in den Fällen mit `static_cast` dynamisch gebunden (weil der Name `m` unqualifiziert ist), während er bei Verwendung von `B::m` statisch gebunden wird (vgl. § 4.5.1).

## 4.7.9 Beispiel




- Die Abbildung zeigt, welche Zugriffsrechte die Elemente `m1` (privat), `m2` (halböffentlich) und `m3` (öffentlich) der Klasse `A` in den abgeleiteten Klassen `A1` (privat), `A2` (halböffentlich) und `A3` (öffentlich) besitzen.

- Das Kreuz steht hierbei für „gesperrt“, die graue Schrift zeigt an, dass es sich um geerbte Elemente handelt.

```
class A {
  void f ();
  int m1;
protected:
  int m2;
public:
  int m3;
} a;

class A1 : private A {
  void f ();
} a1;

class A2 : protected A {
  void f ();
} a2;

class A3 : public A {
  void f ();
} a3;
```

```
void A::f () {
  a1.m1; a1.m2; a1.m3; // Fehler; Fehler; Fehler;
  a2.m1; a2.m2; a2.m3; // Fehler; Fehler; Fehler;
  a3.m1; a3.m2; a3.m3; // OK;     OK;     OK;
}

void A1::f () {
  a1.m1; a1.m2; a1.m3; // Fehler; OK;     OK;
  a2.m1; a2.m2; a2.m3; // Fehler; Fehler; Fehler;
  a3.m1; a3.m2; a3.m3; // Fehler; Fehler; OK;
}

void A2::f () {
  a1.m1; a1.m2; a1.m3; // Fehler; Fehler; Fehler;
  a2.m1; a2.m2; a2.m3; // Fehler; OK;     OK;
  a3.m1; a3.m2; a3.m3; // Fehler; Fehler; OK;
}

void A3::f () {
  a1.m1; a1.m2; a1.m3; // Fehler; Fehler; Fehler;
  a2.m1; a2.m2; a2.m3; // Fehler; Fehler; Fehler;
  a3.m1; a3.m2; a3.m3; // Fehler; OK;     OK;
}
```

- In der Klasse `A` gilt:
  - `a3.m3` ist natürlich korrekt, weil `m3` in `A3` öffentlich ist.
  - Für alle anderen Kombinationen `aI.mJ` gilt:
    `mJ` ist in `AI` höchstens halböffentlich und deshalb gemäß der ersten drei Regeln in § 4.7.4 nur in `AI` oder eventuell in davon abgeleiteten Klassen zugänglich.
  - Für `Z` gleich `A` gilt jedoch:
    - `Z` ist eine öffentliche und deshalb überall zugängliche Basisklasse von `A3` (aber nicht von `A1` und `A2`).
    - `mJ` ist als Element von `Z` zugänglich.

    Deshalb ist `mJ` gemäß der vierten Regel in § 4.7.4 auch als Element von `A3` (aber nicht von `A1` und `A2`) zugänglich.
  - Die in § 4.7.6 genannte Zusatzregel ist für `a3.m2` ebenfalls erfüllt.

- In den Klassen `A1` und `A2` gilt zum Beispiel für `a3.m2`:
  - `m2` ist in `A3` halböffentlich und deshalb gemäß der ersten drei Regeln in § 4.7.4 nur in `A3` oder davon abgeleiteten Klassen zugänglich.
  - Gemäß der vierten Regel gilt jedoch wiederum mit `Z` gleich `A`:
    - `Z` ist eine öffentliche und deshalb überall zugängliche Basisklasse von `A3` (aber nicht von `A1` und `A2`).
    - `m2` ist als Element von `Z` zugänglich, weil `A1` und `A2` von `Z` abgeleitete Klassen sind.

    Deshalb ist `m2` auch als Element von `A3` zugänglich.
  - Die in § 4.7.6 genannte Zusatzregel ist für `a3.m2` jedoch nicht erfüllt.
  - Deshalb ist `a3.m2` jeweils fehlerhaft (und damit die vorigen Überlegungen, ob `m2` in `A3` zugänglich ist, unnötig).
  - Für `a3.A::m2` anstelle von `a3.m2` gilt (vgl. § 4.7.6):
    - `a3` lässt sich implizit in `A` umwandeln.
    - `m2` ist an dieser Stelle ein zugängliches Element von `A`.

    Trotzdem ist `a3.A::m2` fehlerhaft, weil die in § 4.7.6 genannte Zusatzregel wiederum nicht erfüllt ist.

- Für eine weitere Klasse

```
class A23 : public A2 { ..... };
```

gilt:

- `A` ist eine halböffentliche (indirekte) Basisklasse von `A23` und deshalb aufgrund der ersten drei Regeln in § 4.7.3 nur innerhalb von `A23` und davon abgeleiteten Klassen zugänglich.
- Für `Z` gleich `A2` gilt jedoch innerhalb von `A2`:
  - `Z` ist eine öffentliche und damit überall zugängliche Basisklasse von `A2`.
  - `A` ist eine in `A2` zugängliche Basisklasse von `A2`.

  Deshalb ist `A` in `A2` eine zugängliche Basisklasse von `A23`.

## 4.7.10 Adapter als Beispiel für private Basisklassen

```
#include <vector>
using IntVector = std::vector<int>;

class IntStack : private IntVector {
public:
  using IntVector::size;
  using IntVector::begin;  // begin und end werden für
  using IntVector::end;    // "range-based for loops" benötigt.
  void push (int x) { push_back(x); }
  int top () { return back(); }
  int pop () { int x = top(); pop_back(); return x; }
};

IntStack s;
s.push(5); s.push(7);
for (int x : s) cout << x << endl;  // "range-based for loop"
cout << s.pop() << endl;
```

- Die Tatsache, dass `IntStack` von `IntVector` abgeleitet ist, ist ein Implementierungsdetail, das für Klienten von `IntStack` irrelevant ist.
- Falls es später geändert wird, z. B. indem `list<int>` statt `vector<int>` verwendet wird oder eine Implementierung ohne Basisklasse verwendet wird, sollten Klienten davon nichts „merken“ (außer dass sie neu übersetzt werden müssen).
- Deshalb sollte `IntStack` (außerhalb der Klasse selbst) nicht polymorph als `IntVector` verwendbar sein.
- Außerdem sollen viele öffentliche Elementfunktionen von `IntVector` (z. B. `push_front` und `pop_front`) für Klienten von `IntStack` nicht verwendbar sein.
- Deshalb ist `IntVector` keine öffentliche, sondern eine private Basisklasse von `IntStack`.
  (Falls die Details von `IntVector` auch in einer von `IntStack` abgeleiteten Klasse verwendbar sein sollen, sollte `IntVector` stattdessen eine halböffentliche Basisklasse von `IntStack` sein.)
- Mittels `using` können einzelne Elemente von `IntVector` (konkret `size`, `begin` und `end`) trotzdem für Klienten von `IntStack` verwendbar gemacht werden.

- Die Verwendung eines privaten Datenelements mit Typ `B` anstelle einer privaten Basisklasse `B` zur Implementierung einer Klasse `C` hätte im allgemeinen folgende Nachteile:
  - `using` kann nicht verwendet werden.
    Stattdessen müsste man die gewünschten Elementfunktionen neu schreiben, zum Beispiel (wenn das Datenelement `v` heißt):

    ```
    IntVector::size_type size () { return v.size(); }
    ```

    Hierfür muss man die jeweiligen Funktionen wesentlich genauer kennen, um ihre Parameter- und Resultattypen korrekt angeben zu können.
    Für `begin` und `end` muss man außerdem wissen bzw. daran denken, dass es jeweils zwei derartige Funktionen gibt (eine mit und eine ohne `const`-Qualifizierer).
  - Wenn die Klasse `B` virtuelle Elementfunktionen hätte, dann könnten diese in der Klasse `C` nicht überschrieben werden.
  - Wenn die Klasse `B` abstrakt wäre, dann könnte man sie gar nicht als Typ eines Datenelements verwenden.

# 4.8 Schnittstellen (interfaces)

## 4.8.1 Prinzip

- Da eine Klasse mehr als eine Basisklasse besitzen kann, besteht keine Notwendigkeit für Schnittstellen wie in Java.
- Eine Klasse, die nur rein virtuelle Funktionen enthält, entspricht einer Schnittstelle in Java.
- Die `implements`-Beziehung zwischen einer Klasse und einer Schnittstelle in Java entspricht einer normalen Vererbungsbeziehung.

## 4.8.2 Schnittstellen- und Implementierungshierarchie

### Problemstellung

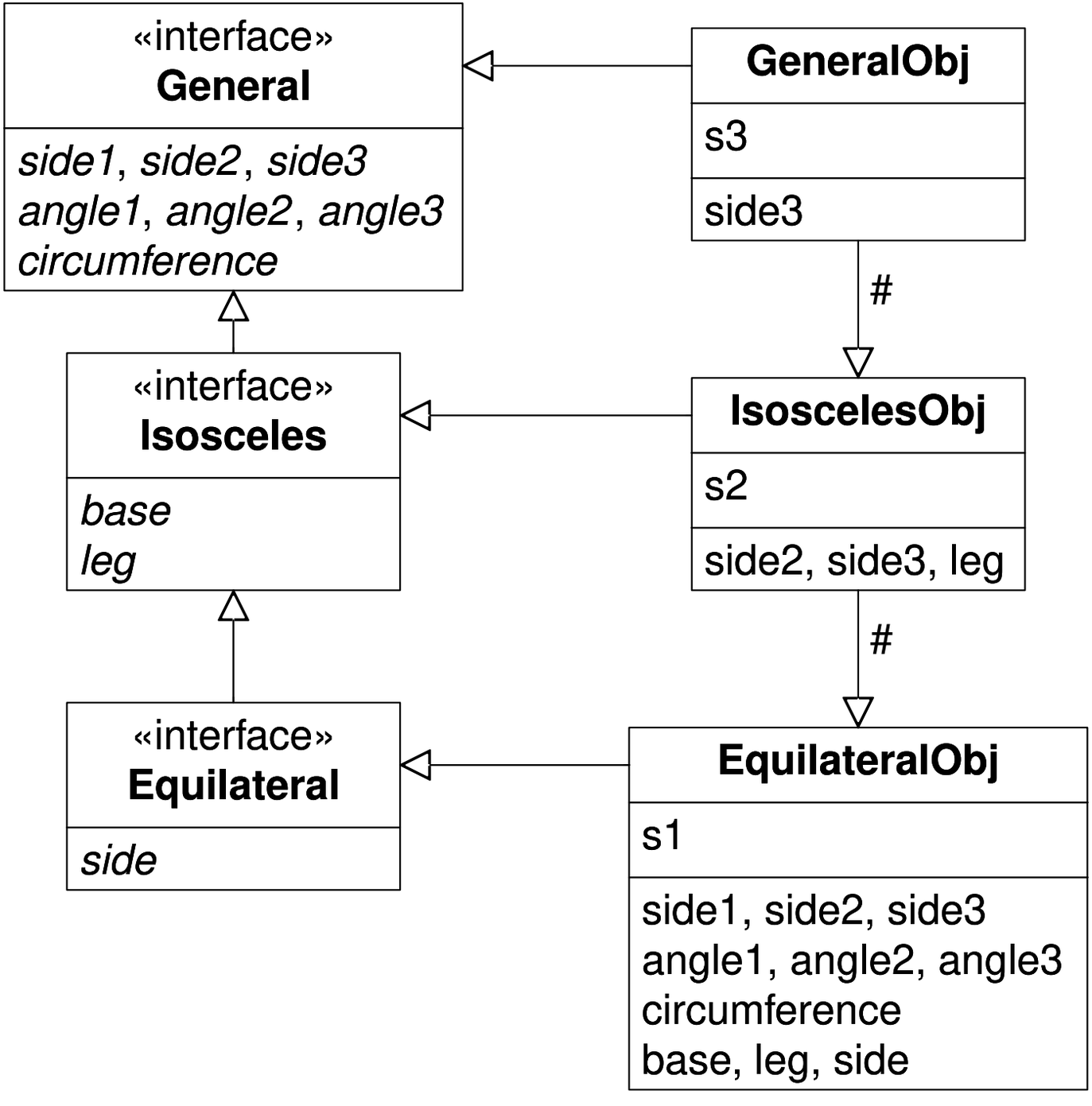
- Es sollen Klassen zur Repräsentation allgemeiner (`General`), gleichschenkliger (`Isosceles`) und gleichseitiger (`Equilateral`) Dreiecke mit sinnvollen Vererbungsbeziehungen definiert werden.

- Da jedes gleichseitige Dreieck auch ein gleichschenkliges und jedes gleichschenklige auch ein allgemeines Dreieck ist, sollten die Objekte der Klassen entsprechend polymorph verwendbar sein, das heißt: `Equilateral` → `Isosceles` → `General`.

- Andererseits werden zur Speicherung eines allgemeinen bzw. gleichschenkligen bzw. gleichseitigen Dreiecks drei bzw. zwei bzw. ein Datenelement benötigt, sodass die umgekehrte Vererbung `General` → `Isosceles` → `Equilateral` aus Implementierungssicht praktischer wäre.

### Lösungsidee

- Es gibt separate Schnittstellen- und Implementierungshierarchien:


«interface»
General
side1, side2, side3
angle1, angle2, angle3
circumference
GeneralObj
s3
side3
#
«interface»
Isosceles
base
leg
IsoscelesObj
s2
side2, side3, leg
#
«interface»
Equilateral
side
EquilateralObj
s1
side1, side2, side3
angle1, angle2, angle3
circumference
base, leg, side

- Die Implementierungen der Elementfunktionen `side1`, `side2`, `side3`, `base`, `leg` und `side` in der Klasse `EquilateralObj` liefern alle den Wert des Datenelements `s1`. Dementsprechend müssen sie in den Klassen `IsoscelesObj` und `GeneralObj` teilweise überschrieben werden, um dort den Wert von `s2` bzw. `s3` zu liefern.

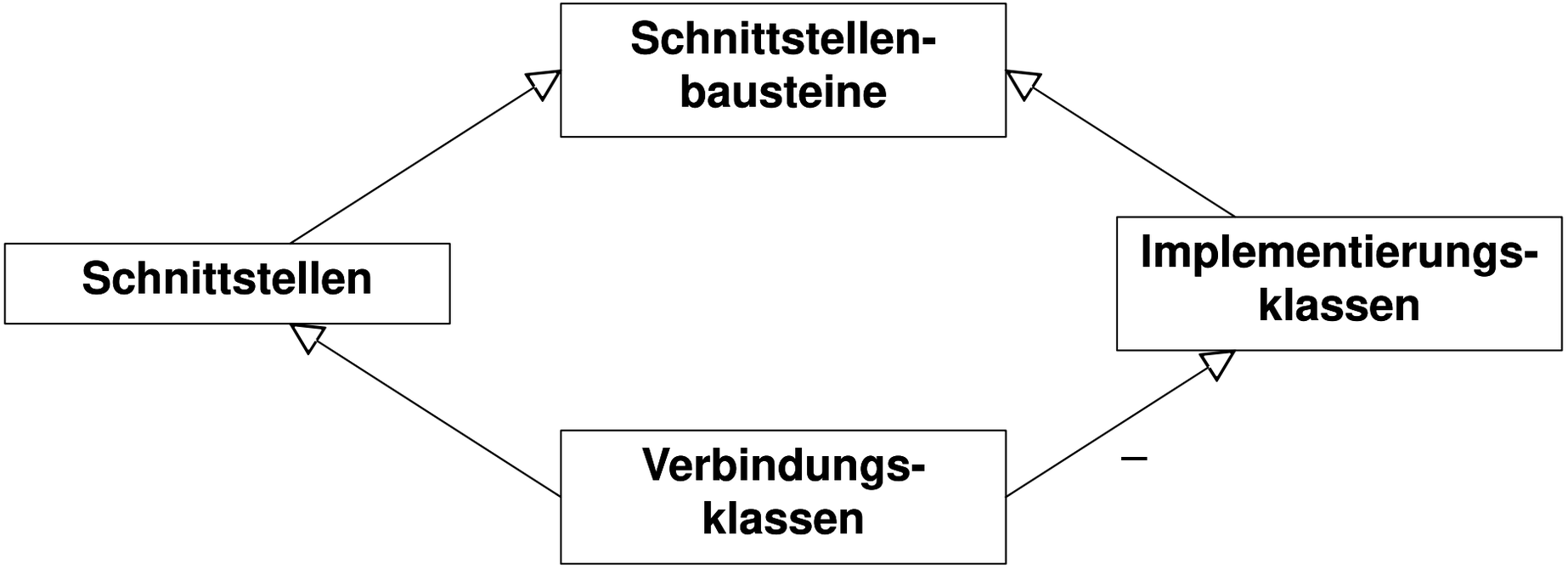
- Die Elementfunktionen `angle1`, `angle2`, `angle3` und `circumference` können in der Klasse `EquilateralObj` unter Verwendung von `side1`, `side2` und `side3` gleich allgemein implementiert werden, sodass sie für alle Arten von Dreiecken korrekt funktionieren.

- Durch die halböffentliche Vererbung zwischen den Implementierungsklassen können entsprechende Objekte immer nur an die „richtigen“ Schnittstellen zugewiesen werden. Beispielsweise kann ein Objekt der Klasse `IsoscelesObj` polymorph als `Isosceles` und als `General` verwendet werden, aber nicht als `Equilateral`.

- Trotzdem besitzt die Lösung noch einen gravierenden Mangel: Anhand der in § 4.5.3 genannten Regeln, wird ein dynamischer Typtest *jede* Art von Dreieck auch als `Isosceles` und `Equilateral` klassifizieren und dementsprechende Umwandlungen erlauben.

## Verbesserung

- Um das zuvor genannte Problem zu vermeiden, darf es zwischen Implementierungsklassen und Schnittstellen *keinerlei* Vererbungsbeziehung geben.
- Stattdessen gibt es „zweiarmige Verbindungsklassen“ zwischen ihnen.
- Damit sich dann Schnittstellen und Implementierungsklassen auf *dieselben* virtuellen Funktionen beziehen, müssen diese in separate „Schnittstellenbausteine“ ausgelagert werden (die dann jeweils als virtuelle Basisklassen verwendet werden).
- Aufgrund der vollständigen Trennung zwischen Schnittstellen und Implementierungsklassen, können Vererbungsbeziehungen zwischen Implementierungsklassen auch wieder öffentlich sein.
- Die Vererbungsbeziehungen zwischen Verbindungs- und Implementierungsklassen sollten aber privat sein, damit Objekte der Verbindungsklassen nur an die zugehörigen Schnittstellen und nicht an die Implementierungsklassen zugewiesen werden können.
- Sowohl die Schnittstellenbausteine als auch die Implementierungsklassen können ggf. in kleinere Bestandteile oder „Komponenten“ zerlegt werden.


Schnittstellen-
bausteine
Schnittstellen
Implementierungs-
klassen
Verbindungs-
klassen

# Umsetzungsmöglichkeit

## *Schnittstellenbausteine*

```
#include <cmath>

#define interface struct

interface Sides {
  virtual double side1 () = 0;
  virtual double side2 () = 0;
  virtual double side3 () = 0;
};

interface Angles {
  virtual double angle1 () = 0;
  virtual double angle2 () = 0;
  virtual double angle3 () = 0;
};

interface Circumference {
  virtual double circumference () = 0;
};
```

```
interface IsoSides {
  virtual double base () = 0;
  virtual double leg () = 0;
};

interface EquSides {
  virtual double side () = 0;
};
```

## *Eigentliche Schnittstellen*

```
interface General
    : virtual Sides, virtual Angles, virtual Circumference {};

interface Isosceles : General, virtual IsoSides {};

interface Equilateral : Isosceles, virtual EquSides {};
```

## Implementierungsklassen

```
class AnglesImp : public virtual Sides, public virtual Angles {
  // Kosinussatz: s1*s1 = s2*s2 + s3*s3 - 2 * s2 * s3 * cos(a1)
  double angle (double s1, double s2, double s3) {
    return acos((s2*s2 + s3*s3 - s1*s1) / (2*s2*s3));
  }
  virtual double angle1 ()
                    { return angle(side1(), side2(), side3()); }
  virtual double angle2 ()
                    { return angle(side2(), side3(), side1()); }
  virtual double angle3 ()
                    { return angle(side3(), side1(), side2()); }
};

class CircumferenceImp
             : public virtual Sides, public virtual Circumference {
  virtual double circumference ()
                    { return side1() + side2() + side3(); }
};
```

```
class Side1Imp : public virtual Sides,
                 public virtual IsoSides, public virtual EquSides {
  double s1;
public:
  Side1Imp (double s) : s1{s} {}
  virtual double side1 () { return s1; }
  virtual double side2 () { return s1; }
  virtual double side3 () { return s1; }
  virtual double base () { return s1; }
  virtual double leg () { return s1; }
  virtual double side () { return s1; }
};

class Side2Imp : public Side1Imp {
  double s2;
public:
  Side2Imp (double b, double l) : Side1Imp{l}, s2{b} {}
  virtual double side2 () { return s2; }
  virtual double base () { return s2; }
};
```

```
class Side3Imp : public Side2Imp {
  double s3;
public:
  Side3Imp (double s1, double s2, double s3)
                                : Side2Imp{s1, s2}, s3{s3} {}
  virtual double side3 () { return s3; }
};
```

## *Verbindungsklassen*

```
class EquilateralObj : public Equilateral, private Side1Imp,
                       private AnglesImp, private CircumferenceImp {
  using Side1Imp::Side1Imp;
};

class IsoscelesObj : public Isosceles, private Side2Imp,
                     private AnglesImp, private CircumferenceImp {
  using Side2Imp::Side2Imp;
};
```

```
class GeneralObj : public General, private Side3Imp,
                   private AnglesImp, private CircumferenceImp {
  using Side3Imp::Side3Imp;
};
```

## *Erzeugungsfunktionen*

```
General* make_general (double s1, double s2, double s3) {
  return new GeneralObj{s1, s2, s3};
}

Isosceles* make_isosceles (double b, double l) {
  return new IsoscelesObj{b, l};
}

Equilateral* make_equilateral (double s) {
  return new EquilateralObj{s};
}
```

## *Anmerkungen*

- Zwischen Schnittstellenbausteinen gibt es keine Vererbungsbeziehungen.
- Eine Implementierungsklasse wird (virtuell) von allen Schnittstellenbausteinen abgeleitet, deren Funktionen sie entweder implementiert oder verwendet.
- Zwischen Implementierungsklassen können beliebige Vererbungsbeziehungen bestehen.
- Bei Verwendung der Erzeugungsfunktionen („Fabrikmuster“) muss man die Implementierungs- und Verbindungsklassen gar nicht kennen. Wenn man ihre Konstruktoren privat oder halböffentlich macht und die Erzeugungsfunktionen als ihre Freunde deklariert, können die Konstruktoren ausschließlich von diesen Funktionen aufgerufen werden; damit bleiben diese Klassen dann vollkommen verborgen.

# 5 Schablonen (templates) und verwandte Themen

## 5.1 Funktionsschablonen

### 5.1.1 Generische Quadratfunktion

- Aus der *Funktionsschablone* (function template)

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

  kann der Übersetzer bei Bedarf für jeden Typ `T` eine entsprechende *Ausprägung* (instantiation) generieren, zum Beispiel:

```
int square<int> (int x) { return x * x; }
double square<double> (double x) { return x * x; }
```

- Allerdings wird diese Generierung nur für Typen `T` gelingen, für die es an der Verwendungsstelle der Funktion eine passende Definition des Multiplikationsoperators `*` gibt.

- Beim Aufruf von `square` wird die Belegung des Schablonenparameters `T` normalerweise aus dem Typ des Funktionsarguments ermittelt (template argument deduction), zum Beispiel:

```
square(1)        // T gleich int
square(1.0)      // T gleich double
square(true)     // T gleich bool
square("abc")    // T gleich const char [4] bzw. const char*
                 // => Fehler
```

- Da es für den Typ `const char*` keinen Multiplikationsoperator gibt, führt `square("abc")` zu einem Übersetzungsfehler.

- Da Wahrheitswerte als ganze Zahlen und umgekehrt verwendet werden können, gibt es für `T` gleich `bool` jedoch einen Multiplikationsoperator, sodass `square(true)` möglich (wenn auch nicht unbedingt sinnvoll) ist.

- Alternativ kann die Belegung des Schablonenparameters auch explizit angegeben werden, zum Beispiel:

  ```
  square<double>(1) // T gleich double
  ```

  Da der Parametertyp `double` damit festgelegt ist, wird der `int`-Wert 1 entsprechend umgewandelt.

- Wenn es für einen benutzerdefinierten Typ einen Multiplikationsoperator gibt, kann er ebenfalls verwendet werden, zum Beispiel:

```
// Rationale Zahl.
struct Rational {
  // Zähler (numerator) und Nenner (denominator).
  const int num, den;

  Rational (int n = 0, int d = 1) : num{n}, den{d} {}
};

// Produkt der rationalen Zahlen x und y.
Rational operator* (Rational x, Rational y) {
  return Rational{x.num * y.num, x.den * y.den};
}

Rational r{3, 4};                  // 3/4
Rational s = square(r);            // 9/16
```

- Anmerkung: Anstelle von `typename` kann auch das Schlüsselwort `class` stehen, obwohl es sich bei den entsprechenden Typen nicht um Klassen handeln muss.

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

## 5.1.2 Generische Maximumfunktion

- Allgemeine Funktionsschablone:

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

- Verwendungsmöglichkeiten:

```
maximum(3, 4)          // T gleich int
maximum(3.0, 4.0)      // T gleich double
maximum(3, 4.0)        // T kann nicht deduziert werden => Fehler
maximum("abc", "def") // T gleich const char*

string s1 = "abc", s2 = "def";
maximum(s1, s2)        // T gleich std::string

Rational r1{1, 2}, r2{3, 4};
maximum(r1, r2)        // T gleich Rational => Fehler
```

- Erläuterungen:
  - Wenn sich aus den Funktionsargumenten unterschiedliche Belegungen für einen Schablonenparameter ergeben, schlägt die template argument deduction fehl.
  - `maximum("abc", "def")` vergleicht lediglich die Adressen der beiden Zeichenketten und liefert die größere von beiden zurück. Es findet also kein inhaltlicher Vergleich der Zeichenketten statt.
  - Für den Typ `std::string` gibt es einen Vergleichsoperator `>`, der einen lexikographischen Vergleich der Zeichenketten durchführt.
  - Für den in § 5.1.1 definierten Typ `Rational` gibt es (momentan) keinen Vergleichsoperator `>`, sodass die template instantiation für `maximum(r1, r2)` fehlschlägt.
  - Mit einer entsprechenden Operatordefinition würde der Aufruf jedoch funktionieren, zum Beispiel:

```
bool operator> (Rational x, Rational y) {
  return double(x.num)/x.den > double(y.num)/y.den;
}
```

- Spezielle Funktion für `const char*`:

```
using str = const char*;
str maximum (str x, str y) {
  return strcmp(x, y) > 0 ? x : y;
}

maximum("abc", "def")       // Aufruf der speziellen Funktion.
maximum<str>("abc", "def") // Aufruf der allgemeinen Funktion.
```

- Allgemeine Regel: Wenn eine normale Funktion und eine Funktionsschablone gleich gut passen, wird die normale Funktion bevorzugt.

- Spezielle Funktionsschablone für Zeigertypen:

```
template <typename 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;
}

int a = 1; int b = 2; int* p = &a; int* q = &b;
maximum(p, q)    // Aufruf der Funktion für Zeiger mit T gleich
                 // int, die ihrerseits die allgemeine Funktion
                 // mit T gleich int aufruft.
maximum(&p, &q) // Aufruf der Funktion für Zeiger mit T gleich
                 // int*, die ihrerseits erneut die Funktion für
                 // Zeiger mit T gleich int aufruft, die wiederum
                 // die allg. Funktion mit T gleich int aufruft.
maximum<int*>(p, q) // Aufruf der allgemeinen Funktion
                 // mit T gleich int*.
```

- Allgemeine Regel: Wenn eine Funktionsschablone durch eine andere implementiert werden könnte, aber nicht umgekehrt, ist die erste spezieller als die zweite und wird bei einem Aufruf ggf. bevorzugt.

- Überprüfung der Regel im obigen Beispiel (die künstlichen Namen `maximum1` und `maximum2` dienen nur zur Unterscheidung der beiden Funktionsschablonen):

```
template <typename T1>  // Notwendige Vorabdeklaration.
T1 maximum1 (T1 x, T1 y);

template <typename T2>
T2* maximum2 (T2* x, T2* y) {
  // Diese Implementierung von maximum2 durch maximum1 ist
  // für beliebige Typen T2 korrekt (es wird T1 gleich T2*
  // deduziert).
  return maximum1(x, y);
}
template <typename T1>
T1 maximum1 (T1 x, T1 y) {
  // Diese Implementierung von maximum1 durch maximum2 ist
  // für beliebige Typen T1 nicht korrekt (es müsste T1 gleich
  // T2* sein, was nur für Zeigertypen T1 der Fall wäre.)
  return maximum2(x, y);
}
```

# 5.2 Typschablonen

## 5.2.1 Containertypen

- Die Containertypen `vector`, `list`, `set`, `map` etc. der C++-Standardbibliothek (vgl. § 7.1.1) sind allesamt Typschablonen, die mit (nicht ganz) beliebigen Elementtypen verwendet werden können.

- Deshalb wird die Bibliothek oft auch als Standard Template Library (STL) bezeichnet.

- Die Bibliothekstypen `string`, `u8string`, `u16string`, `u32string` und `wstring` sind allesamt Ausprägungen einer Typschablone `basic_string` mit Elementtypen `char`, `char8_t`, `char16_t`, `char32_t` bzw. `wchar_t`.

## 5.2.2 Paare

- Typschablone (ähnlich zu `std::pair`):

```
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>.
  template <typename U, typename V>
  Pair (const Pair<U, V>& that)
                          : x(that.getX()), y(that.getY()) {}
};
```

- Funktionsschablone zur bequemeren Verwendung vor C++17 (analog zu `std::make_pair`):

```
template <typename X, typename Y>
Pair<X, Y> mkpair (X x, Y y) {
  return Pair<X, Y>{x, y};
}
```

- Verwendungsmöglichkeit:

```
                                    // Seit C++17:
Pair<int, int> p1{3, 4};            Pair p1{3, 4};
p1 = Pair<int, int>{5, 6};          p1 = Pair{5, 6};
p1 = mkpair(7, 8);                  p1 = {7, 8};

// Implizite Umwandlung von p1 mit Typ Pair<int, int>
// in den Typ Pair<double, double> von p2 und umgekehrt.
Pair<double, double> p2 = p1;
p1 = p2;

// Fehler: Pair<string, string> kann nicht
// in den Typ Pair<int, int> von p1 umgewandelt werden.
p1 = Pair<string, string>{"abc", "def"};
```

- Erläuterungen:
  - Die Konstruktorschablone kann prinzipiell mit Paaren eines beliebigen Typs `Pair<U, V>` aufgerufen werden.
  - Allerdings wird ein entsprechender Aufruf nur gelingen, wenn sich die Elemente `x` und `y` mit Typ `X` bzw. `Y` tatsächlich mit Werten des Typs `U` bzw. `V` initialisieren lassen (was z. B. für `X` gleich `int` und `U` gleich `string` nicht der Fall ist).
  - Für diese Initialisierungen werden bewusst runde statt geschweifte Klammern verwendet (vgl. § 3.2.2), um beliebige implizite Umwandlungen zu erlauben. (Bei Verwendung geschweifter Klammern sind sog. „narrowing conversions“ verboten, zu denen auch Umwandlungen von Gleitkomma- in Ganzzahltypen und umgekehrt gehören.)
  - Bei einer Zuweisung wie z. B. `p1 = p2` wird das Paar `p2`, wenn nötig, ebenfalls mit dieser Konstruktorschablone in den Typ von `p1` umgewandelt.
  - Wenn `X` und `U` und/oder `Y` und `V` verschiedene Typen sind, sind auch `Pair<X, Y>` und `Pair<U, V>` verschiedene Typen, obwohl sie aus derselben Typschablone generiert werden. Dementsprechend darf `Pair<X, Y>` *nicht* auf die privaten Datenelemente von `Pair<U, V>` zugreifen, sondern muss stattdessen die öffentlichen Elementfunktionen `getX` und `getY` verwenden.
  - Bei der Deklaration von Variablen und beim Aufruf von Konstruktoren einer Typschablone muss man vor C++17 die Schablonenparameter explizit angeben. Seit C++17 und bei Aufrufen von Funktionsschablonen werden sie jedoch, wenn möglich, deduziert.

- Explizite Spezialisierung der Typschablone:

```
template <>
class Pair<bool, bool> {
  // Ein Byte zur Speicherung der beiden bool-Werte.
  char xy;
public:
  // y wird im niedrigsten Bit von xy gespeichert,
  // x im nächsthöheren.
  Pair (bool x, bool y) : xy{x << 1 | y} {}
  bool getX () const { return xy & 2; }
  bool getY () const { return xy & 1; }
};
```

- Für `Pair<bool, bool>` wird die Spezialisierung verwendet, für alle anderen Typen `Pair<X, Y>` die allgemeine Typschablone.

- Eine Spezialisierung kann vollkommen anders definiert sein als die allgemeine Typschablone. Beispielsweise hat die obige Spezialisierung keine Konstruktorschablone zur Initialisierung mit Paaren eines anderen Typs.

## 5.2.3 Matrizen

- Typschablone:

```
template <typename T, int M, int N>
class Matrix {
  // Zweidimensionale Reihe zur Speicherung der Elemente.
  T a [M] [N];
public:
  // Lesender und schreibender Zugriff auf die Elemente
  // (mit Indizierung ab 1).
  T get (int i, int k) const { return a[i-1][k-1]; }
  void set (int i, int k, T x) { a[i-1][k-1] = x; }
};
```

- Erläuterungen:
  - Neben Typen sind u. a. auch ganze Zahlen als Schablonenparameter erlaubt.
  - Die zugehörigen Schablonenargumente müssen konstante Werte sein, die bereits zur Übersetzungszeit berechnet werden können.
  - Damit sind z. B. `Matrix<double, 3, 4>` und `Matrix<double, 4, 3>` unterschiedliche Typen.

- Typkorrekte Matrixoperationen:

```
template <typename 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>
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++) {
      sum += a.get(i, j) * b.get(j, k);
    }
    c.set(i, k, sum);
  }
  return c;
}
```

- Verwendungsmöglichkeit:

```
// 3x4-Matrix a und 4x3-Matrix b.
Matrix<double, 3, 4> a;
Matrix<double, 4, 3> b;

// Elemente von a und b zuweisen.
for (int i = 1; i <= 3; i++) for (int k = 1; k <= 4; k++) {
  a.set(i, k, ......);
  b.set(k, i, ......);
}

// a und b multiplizieren
// und das Ergebnis mit Typ Matrix<double, 3, 3> quadrieren,
// wofür wiederum der obige Multiplikationsoperator
// verwendet wird.
Matrix<double, 3, 3> c = square(a * b);

// Seit C++17:
Matrix c = square(a * b);
```

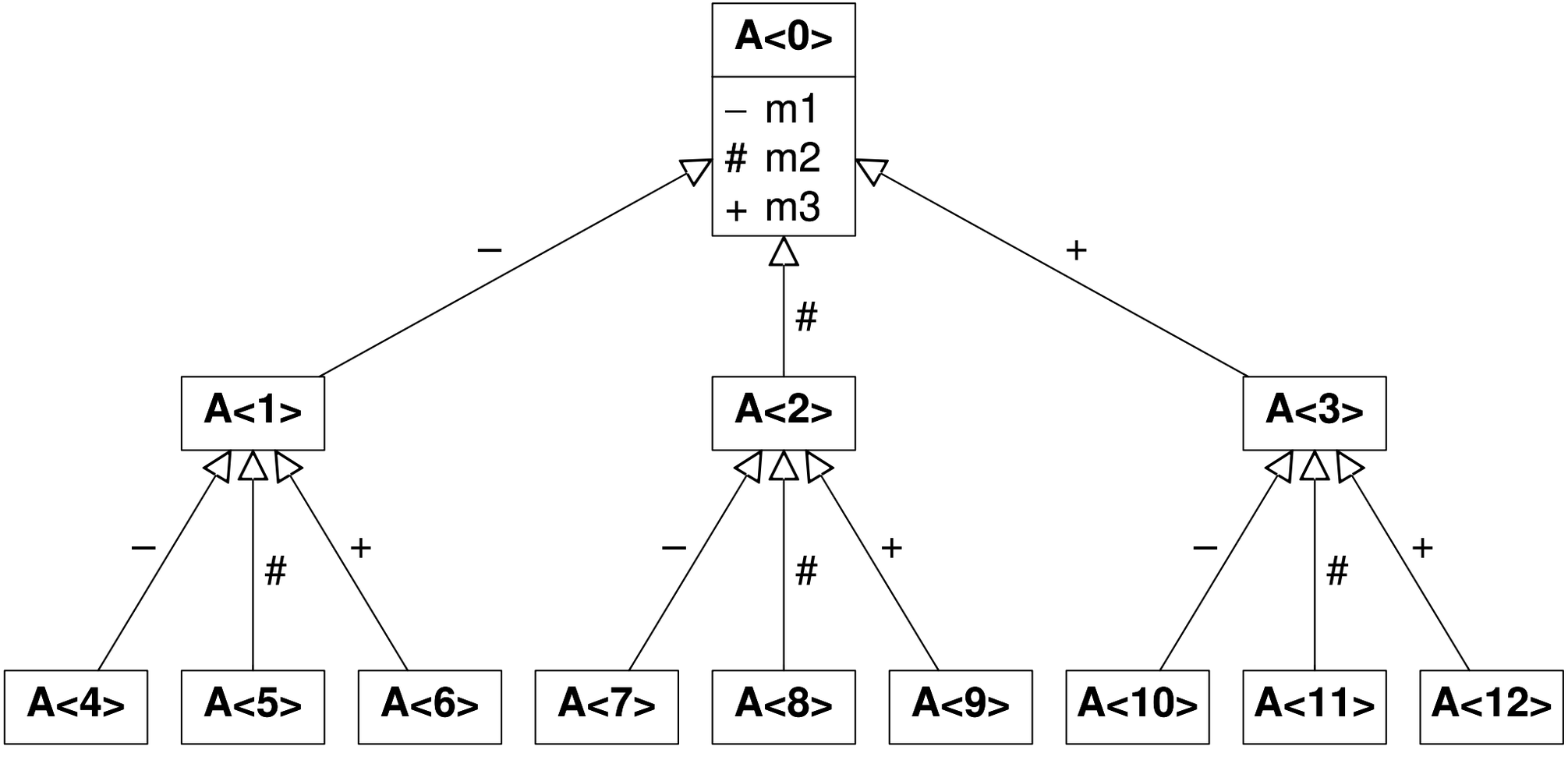
- Partielle Spezialisierung der Typschablone:

```
template <int M, int N>
class Matrix<bool, M, N> {
  // M * N Bits, aufgerundet auf ein Vielfaches von 8.
  unsigned char a [(M * N + 7) / 8];
  // Position des Bits für das Matrixelement (i, k).
  int pos (int i, int k) { return (i-1) * N + (k-1); }
  // Element von a, in dem sich dieses Bit befindet.
  unsigned char& elem (int i, int k) { return a[pos(i, k) / 8]; }
  // Bitmuster zur Selektion dieses Bits in diesem Element.
  unsigned char mask (int i, int k) { return 1 << pos(i, k) % 8; }
public:
  // Lesender und schreibender Zugriff auf die Elemente.
  bool get (int i, int k) {
    return elem(i, k) & mask(i, k);
  }
  void set (int i, int k, bool x) {
    if (x) elem(i, k) |= mask(i, k);
    else elem(i, k) &= ~mask(i, k);
  }
};
```

- Für `Matrix<bool, ..., ...>` wird die partielle Spezialisierung verwendet (die wiederum vollkommen anders definiert sein kann als die allgemeine Typschablone), für alle anderen Typen `Matrix<T, ..., ...>` die allgemeine Typschablone.

## 5.2.4 Rekursive Schablonen

❒ Die nachfolgende Klassenhierarchie soll durch eine Typschablone `A` mit geeigneten Spezialisierungen definiert werden:



- Typschablone mit Spezialisierungen:

```
// Allgemeine Typschablone.
// Der zweite Schablonenparameter ist anonym und besitzt einen
// Defaultwert, der vom ersten Schablonenparameter N abhängt
// (was bei Defaultwerten für Funktionsparameter nicht ginge).
// Er wird nur zur Definition der partiellen Spezialisierungen
// gebraucht.
template <int N, int = N % 3>
class A;

// Explizite Spezialisierung der Wurzelklasse A<0>.
template <>
class A <0> {
  int m1;
protected:
  int m2;
public:
  int m3;
};
```

```
// Partielle Spezialisierung für A<1>, A<4>, A<7>, ...
template <int N>
class A <N, 1> : private A<(N - 1) / 3> {};

// Partielle Spezialisierung für A<2>, A<5>, A<8>, ...
template <int N>
class A <N, 2> : protected A<(N - 1) / 3> {};

// Partielle Spezialisierung für A<3>, A<6>, A<9>, ...
template <int N>
class A <N, 0> : public A<(N - 1) / 3> {};
```

- Erläuterungen:
  - Eine Klasse wie z. B. `A<10>` heißt aufgrund des Defaultwerts des zweiten Schablonenparameters tatsächlich `A<10, 1>`.
  - Dementsprechend wird sie aus der ersten partiellen Spezialisierung generiert und besitzt deshalb `A<3>` gleich `A<3, 0>` als private Basisklasse.
  - Diese wird rekursiv aus der dritten partiellen Spezialisierung generiert und besitzt deshalb `A<0>` (gleich `A<0, 0>`) als öffentliche Basisklasse.
  - Diese steht als explizite Spezialisierung zur Verfügung und beendet damit die rekursive Generierung.

## 5.2.5 Template-Metaprogrammierung

- Beispiel Fibonacci-Zahlen ($f_0 = 0$, $f_1 = 1$, $f_n = f_{n-2} + f_{n-1}$ für $n \geq 2$):

```
// Allgemeine rekursive Schablone.
template <int N>
struct Fib {
  static const int value = Fib<N-2>::value + Fib<N-1>::value;
};

// Spezialisierungen für den Rekursionsabbruch.
template <>
struct Fib<0> {
  static const int value = 0;
};
template <>
struct Fib<1> {
  static const int value = 1;
};

int main () {
  cout << Fib<46>::value << endl;
}
```

- Erläuterungen:
  - Wenn eine statische konstante Elementvariable mit ganzzahligem (oder Aufzählungs-) Typ direkt in der Klassendefinition initialisiert wird, muss der Initialisierungsausdruck ein *konstanter Ausdruck* sein, dessen Wert bereits zur Übersetzungszeit bestimmt werden kann (vgl. § 3.7).
  - Eine solche Elementvariable kann dann selbst in derartigen konstanten Ausdrücken verwendet werden.
  - Deshalb können die Werte aller Elementvariablen `Fib<N>::value` bereits zur Übersetzungszeit berechnet werden.
  - Die Verwendung von `Fib<46>` führt zur rekursiven Generierung von `Fib<44>` und `Fib<45>`, für die wiederum `Fib<42>` und `Fib<43>` bzw. `Fib<43>` und `Fib<44>` generiert wird, usw.
  - Da eine bereits generierte Klasse aber nicht erneut generiert werden muss, wird jede Klasse von `Fib<0>` bis `Fib<46>` jeweils nur einmal generiert, d. h. der Übersetzer führt automatisch eine „tabellengestützte Optimierung“ der Berechnung durch (vgl. „dynamic programming“ in „Algorithmen und Datenstrukturen 2“).
  - Deshalb wird `Fib<46>::value` vom Übersetzer in linearer Zeit berechnet, während die folgende rekursive Funktion exponentielle Zeit benötigt, weil sie einzelne Teilergebnisse immer wieder neu berechnet:

```
int fib (int n) {
  if (n <= 1) return n;
  else return fib(n-2) + fib(n-1);
}
```

- Anmerkungen:
  - $f_{46}$ ist die größte Fibonacci-Zahl, die als 32-Bit-`int`-Wert dargestellt werden kann.
  - Man kann zeigen, dass Template-Metaprogrammierung Turing-vollständig ist, obwohl dies von den „Erfindern“ vermutlich nicht beabsichtigt war.
  - Seit C++11 gibt es mit `constexpr` eine einfachere und natürlichere Möglichkeit, komplexe Berechnungen zur Übersetzungszeit durchzuführen.
    Dazu muss die obige Funktion `fib` lediglich mit dem Schlüsselwort `constexpr` definiert werden. (Allerdings kann man dann nicht davon ausgehen, dass der Übersetzer bei Aufrufen der Funktion tabellengestützte Optimierung verwendet.)
  - Auch Variablenschablonen ermöglichen zum Teil einfachere Formulierungen derartiger Berechnungen (vgl. § 5.4.1).

## 5.2.6 Aliasschablonen

❒ Beispiel:

```
// Knoten einer verketteten Liste mit Elementtyp T.
template <typename T>
struct Node {
  // Das in diesem Knoten gespeicherte Element.
  T head;

  // Zeiger auf den nächsten Knoten oder Nullzeiger.
  Node* tail;
};

// Für jeden Typ T ist List<T> ein Alias für Node<T>*.
template <typename T>
using List = Node<T>*;
```

## 5.2.7 Freunde von Typschablonen

- Klassen als Freunde:

```
template <typename T>
class C;

template <typename T>
class A {
  // Klasse B ist ein Freund aller Klassen A<T>.
  friend class B;

  // Jede Klasse C<T> ist ein Freund der entspr. Klasse A<T>.
  // C muss schon vorher als Klassenschablone deklariert sein.
  friend class C<T>;

  // Jede Klasse D<U> ist ein Freund aller Klassen A<T>.
  template <typename U>
  friend class D;
};
```

- Funktionen als Freunde:

```
template <typename T>
void h (T);

template <typename T>
class A {
  // Funktion f ist ein Freund aller Klassen A<T>.
  friend void f ();

  // Jede Funktion g mit Parametertyp T ist ein Freund
  // der entsprechenden Klasse A<T>.
  // (g ist keine Funktionsschablone!)
  friend void g (T);

  // Jede Funktion h<T> (mit Parametertyp T) ist ein Freund
  // der entsprechenden Klasse A<T>.
  // h muss schon vorher als Funktionsschablone deklariert sein.
  friend void h<T> (T);

  // Jede Funktion i<U> ist ein Freund aller Klassen A<T>.
  template <typename U>
  friend void i (U);
};
```

# 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);
```

# 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;
```

# 6 Überladene Operatoren

## 6.1 Allgemeines Prinzip

### 6.1.1 Definition von Operatoren durch gewöhnliche Funktionen

```
// Rationale Zahl (vgl. §5.1.1).
struct Rational {
  // Zähler (numerator) und Nenner (denominator).
  const int num, den;

  // Konstruktor.
  explicit Rational (int n = 0, int d = 1) : num{n}, den{d} {}
};
```

```
// Differenz der rationalen Zahlen x und y (binärer Operator).
Rational operator- (Rational x, Rational y) {
  return Rational{x.num * y.den - y.num * x.den, x.den * y.den};
}

// Negation der rationalen Zahl x (unärer Operator).
Rational operator- (Rational x) {
  return Rational{-x.num, x.den};
}

// Mögliche Verwendung.
Rational r1{1, 2};
Rational r2{2, 3};
Rational r3 = -r1 - r2;
```

## 6.1.2 Definition von Operatoren durch Elementfunktionen

```
// Rationale Zahl.
struct Rational {
  // Zähler (numerator) und Nenner (denominator).
  const int num, den;

  // Konstruktor.
  explicit Rational (int n = 0, int d = 1) : num{n}, den{d} {}

  // Differenz der rationalen Zahlen *this und y (binärer Operator).
  Rational operator- (Rational y) const {
    return Rational{num * y.den - y.num * den, den * y.den};
  }

  // Negation der rationalen Zahl *this (unärer Operator).
  Rational operator- () const {
    return Rational{-num, den};
  }
};

// Verwendung wie zuvor.
```

### 6.1.3 Erläuterungen

- Das Schlüsselwort `operator`, gefolgt von einem Operatorsymbol, entspricht syntaktisch einem Funktionsnamen.

- Ein Ausdruck wie z. B. `-r1 - r2` mit Teilausdrücken `r1` und `r2` des Typs `Rational` wird vom Übersetzer entweder in gewöhnliche Funktionsaufrufe oder in Aufrufe von Elementfunktionen (oder eine passende Mischform) transformiert:

```
// Wenn Operatoren durch gewöhnliche Funktionen definiert wurden:
operator-(operator-(r1), r2)

// Wenn Operatoren durch Elementfunktionen definiert wurden:
r1.operator-().operator-(r2)
```

- Normalerweise ist die Definition durch gewöhnliche Funktionen und die Definition durch Elementfunktionen äquivalent.

- Die Operatoren = (Zuweisung), `[]` (Indexoperation), `()` (Funktionsaufruf) und `->` (Elementzugriff über Zeiger) können jedoch nur als Elementfunktionen definiert werden.

- Der Operator `->` wird hierbei als unärer Postfix-Operator interpretiert, auf dessen Resultat der Operator dann erneut angewandt wird.

- Die Operatoren `::` (Qualifizierung von Namen), `.` (Elementzugriff), `.*` (Elementzugriff über sog. Elementzeiger) und `?:` (Verzweigung) können grundsätzlich nicht überladen werden.

- Man kann weder neue Operatorsymbole einführen noch die Regeln für Vorrang und Assoziativität der vorhandenen Operatoren verändern.

- Mindestens ein Operand eines überladenen Operators muss ein benutzerdefinierter Typ (d. h. eine Klasse/Struktur/Überlagerung oder ein Aufzählungstyp) sein, d. h. die Bedeutung der vordefinierten Operatoren kann nicht verändert werden.

- Bei den Funktionen zur Definition überladener Operatoren kann es sich auch um Schablonen handeln.

- Ebenso wie bei gewöhnlichen Funktionsaufrufen ist es undefiniert, ob vor dem Aufruf eines binären Operators zuerst sein linker oder sein rechter Operand ausgewertet wird.

# 6.2 Vergleichsoperatoren

- Seit C++20 gelten die im folgenden beschriebenen zusätzlichen Regeln, um den Aufwand zur Definition von Vergleichsoperatoren zu reduzieren.

## 6.2.1 Gleichheitsoperatoren mit vertauschten Operandentypen

- Bei der Interpretation eines Vergleichs `x == y` werden zusätzlich zu den vordefinierten und explizit definierten (gewöhnlichen oder Element-) Funktionen `operator==` mit Operandentypen `U` und `V` auch die entsprechenden *synthetisierten* Funktionen mit den vertauschten Operandentypen `V` und `U` betrachtet (sofern `U` und `V` verschieden sind).

- Wenn die am besten passende Funktion dann eine dieser synthetisierten Funktionen ist, wird der Ausdruck `x == y` durch den Funktionsaufruf `operator==(y, x)` bzw. `y.operator==(x)` ersetzt, wobei `operator==` die zugehörige vordefinierte oder explizit definierte Funktion bezeichnet.

- Zum Beispiel:

```
// Vergleich der rationalen Zahl x mit der ganzen Zahl y
// oder umgekehrt.
bool operator== (Rational x, int y) {
    return x.num == y * x.den;
}

// Mögliche Verwendung.
Rational r{6, 3};
int i = 2;

// Ausdruck:    // Bedeutung:              // Oder (falls der obige
                                          // Operator als Element-
                                          // funktion von Rational
                                          // definiert ist):
r == i;         // operator==(r, i)       // r.operator==(i)
i == r;         // operator==(r, i)       // r.operator==(i)
```

- Vor C++20 hätte man zusätzlich `operator== (int, Rational)` definieren müssen, damit der Vergleich `i == r` vom Übersetzer akzeptiert wird.

## 6.2.2 Implizit definierte Ungleichheitsoperatoren

- Bei der Interpretation eines Vergleichs `x != y` werden zusätzlich zu den vordefinierten und explizit definierten Funktionen `operator!=` auch *umgeschriebene* Funktionen betrachtet, die sowohl aus den vordefinierten und explizit definierten als auch aus den gemäß § 6.2.1 synthetisierten Funktionen `operator==` gebildet werden.

- Wenn die am besten passende Funktion dann eine dieser umgeschriebenen Funktionen ist, wird der Ausdruck `x != y` durch `!(x == y)` ersetzt, was gemäß § 6.1.3 wiederum durch `!operator==(x, y)` bzw. `!x.operator==(y)` oder gemäß § 6.2.1 durch `!operator==(y, x)` bzw. `!y.operator==(x)` ersetzt wird.

- Zum Beispiel:

```
// Ausdruck:     // Bedeutung:              // Oder:
r != i;          // !operator==(r, i)      // !r.operator==(i)
i != r;          // !operator==(r, i)      // !r.operator==(i)
```

- Vor C++20 hätte man zusätzlich `operator!=(Rational, int)` und `operator!=(int, Rational)` definieren müssen, damit die Vergleiche `r != i` und `i != r` vom Übersetzer akzeptiert werden. (Das heißt, man hätte insgesamt vier statt nur einer Funktion definieren müssen, um rationale und ganze Zahlen in beiden Richtungen auf Gleichheit und Ungleichheit testen zu können.)

## 6.2.3 Dreiwegvergleich

### Grundprinzip

- Seit C++20 gibt es neben den sechs gewohnten Vergleichsoperatoren <, <=, ==, !=, >=, > einen weiteren Operator <=>, der als *Dreiwegvergleich* bezeichnet wird.
- Sein Resultattyp ist normalerweise einer der Typen `std::strong_ordering`, `std::weak_ordering` oder `std::partial_ordering`.
- Jeder dieser Typen besitzt die Werte `less`, `equivalent` und `greater` (z. B. `strong_ordering::less` oder `partial_ordering::equivalent`), die ausdrücken, dass der linke Operand eines Dreiwegvergleichs kleiner bzw. äquivalent bzw. größer als der rechte Operand ist.
- `strong_ordering` besitzt zusätzlich den Wert `equal`, der hier aber gleichbedeutend mit `equivalent` ist.
- `partial_ordering` besitzt zusätzlich den Wert `unordered`, der ausdrückt, dass die beiden Operanden nicht vergleichbar sind, d. h. dass der linke Operand weder kleiner noch äquivalent noch größer als der rechte Operand ist.
- Jeder dieser Werte kann mit jedem Vergleichsoperator in beiden Richtungen mit dem `int`-Wert 0 verglichen werden, wobei `less` kleiner, `equivalent` und `equal` gleich, `greater` größer und `unordered` weder kleiner noch gleich noch größer als 0 ist. (Wenn ein solcher Vergleich wiederum ein Dreiwegvergleich ist, erhält man einen Wert desselben Typs.)

## Beispiele

```
// Vergleich:                          // Resultatwert:
1 <=> 2                                strong_ordering::less
1 <=> 1                                strong_ordering::equal/equivalent
2 <=> 1                                strong_ordering::greater

1.0 <=> 2.0                            partial_ordering::less
1.0 <=> 1.0                            partial_ordering::equivalent
2.0 <=> 1.0                            partial_ordering::greater
-0.0 <=> +0.0                          partial_ordering::equivalent
double nan = 0.0/0.0;
1.0 <=> nan                            partial_ordering::unordered
nan <=> 1.0                            partial_ordering::unordered
nan <=> nan                            partial_ordering::unordered

1 <=> 2 < 0                            true
1 <=> 2 <= 0                           true
1 <=> 2 == 0                           false
1 <=> 2 != 0                           true
1 <=> 2 >= 0                           false
1 <=> 2 > 0                            false
1 <=> 2 <=> 0                          strong_ordering::less
```

## Erläuterungen

- Die `double`-Werte `-0.0` und `+0.0` sind äquivalent, obwohl sie nicht exakt gleich sind. Beispielsweise liefert die Division `1.0/+0.0` einen positiven unendlichen Wert, während `1.0/-0.0` einen negativen unendlichen Wert liefert.

- Die Division `0.0/0.0` liefert einen NaN-Wert (not a number), der weder kleiner noch äquivalent noch größer als jeder `double`-Wert (einschließlich NaN-Werten selbst!) ist.

- Der Dreiwegvergleich `<=>` ist linksassoziativ und bindet stärker als die relationalen Operatoren `<`, `<=`, `>=` und `>`, die wiederum stärker als die Gleichheitsoperatoren `==` und `!=` binden.

- Dementsprechend ist `1 <=> 2 < 0` z. B. gleichbedeutend mit `(1 <=> 2) < 0` und `1 <=> 2 <=> 0` gleichbedeutend mit `(1 <=> 2) <=> 0`.

- Ein typisches Beispiel für `weak_ordering` ist ein lexikographischer Vergleich von Zeichenketten ohne Berücksichtigung von Groß- und Kleinschreibung, wo beispielsweise `"abc"` und `"ABC"` äquivalent, aber nicht exakt gleich sind.

# Benutzerdefinierte Dreiwegvergleiche

```
// Vergleich der rationalen Zahlen x und y.
strong_ordering operator<=> (Rational x, Rational y) {
  // Jeden Zähler mit dem jeweils anderen Nenner multiplizieren
  // und die Ergebnisse vergleichen.
  strong_ordering r = x.num * y.den <=> y.num * x.den;

  // Wenn beide Nenner dasselbe Vorzeichen besitzen,
  // bleibt die Relation durch die Multiplikationen erhalten,
  // andernfalls dreht sie sich um.
  return (x.den > 0) == (y.den > 0) ? r : 0 <=> r;
}
```

### 6.2.4 Dreiwegvergleich mit vertauschten Operandentypen

- Bei der Interpretation eines Dreiwegvergleichs `x <=> y` werden zusätzlich zu den vordefinierten und explizit definierten Funktionen `operator<=>` mit Operandentypen `U` und `V` auch die entsprechenden synthetisierten Funktionen mit den vertauschten Operandentypen `V` und `U` betrachtet (sofern `U` und `V` verschieden sind).

- Wenn die am besten passende Funktion dann eine dieser synthetisierten Funktionen ist, wird der Ausdruck `x <=> y` durch `0 <=> (y <=> x)` ersetzt, wobei `<=>` innerhalb der Klammer den zugehörigen vordefinierten oder explizit definierten Operator bezeichnet.

### 6.2.5 Implizit definierte relationale Operatoren

- Bei der Interpretation eines Vergleichs `x @ y`, wobei `@` einer der vier relationalen Operatoren `<`, `<=`, `>=` oder `>` ist, werden neben den jeweiligen vordefinierten und explizit definierten Operatoren auch umgeschriebene Funktionen betrachtet, die sowohl aus den vordefinierten und explizit definierten als auch aus den gemäß § 6.2.4 synthetisierten Funktionen `operator<=>` gebildet werden.

- Wenn die am besten passende Funktion dann eine dieser umgeschriebenen Funktionen ist, wird der Ausdruck `x @ y` durch `x <=> y @ 0` ersetzt, wobei `<=>` den zugehörigen vordefinierten, explizit definierten oder synthetisierten Operator bezeichnet (der dann ggf. wieder gemäß § 6.2.4 interpretiert wird).

- Zum Beispiel:

```
// Ausdruck:                    // Bedeutung:
r < r                           // (r <=> r) < 0
r >= r                          // (r <=> r) >= 0
```

- Vor C++20 hätte man vier statt nur einer Funktion definieren müssen, damit die vier relationalen Vergleiche `r @ r` vom Übersetzer akzeptiert werden.

- Fortsetzung des Beispiels:

```
// Vergleich der rationalen Zahl x mit der ganzen Zahl y
// oder umgekehrt.
strong_ordering operator<=> (Rational x, int y) {
    return x <=> Rational{y, 1};
}

// Ausdruck:                    // Bedeutung:
r < i                           // r <=> i < 0
i >= r                          // 0 <=> (r <=> i) >= 0 bzw.
                                // r <=> i <= 0
```

- Vor C++20 hätte man zusätzlich acht statt nur einer Funktion definieren müssen, damit die relationalen Vergleiche `r @ i` und `i @ r` vom Übersetzer akzeptiert werden.

### 6.2.6 Allgemeine Regeln

- Wenn eine vordefinierte oder explizit definierte und eine synthetisierte oder umgeschriebene Funktion gleich gut passen (aber nur dann), wird die vordefinierte bzw. explizit definierte Funktion bevorzugt.

- Wenn zwei umgeschriebene Funktionen gleich gut passen, von denen eine direkt aus einer vordefinierten oder explizit definierten Funktion und die andere aus einer synthetisierten Funktion mit vertauschten Operandentypen gebildet wurde, wird erstere bevorzugt.

### 6.2.7 Anmerkungen

- Dreiwegvergleiche werden nur bei der Interpretation der vier relationalen Vergleiche <, <=, >= und > berücksichtigt, aber nicht bei Tests auf Gleichheit und Ungleichheit.

- Obwohl die neuen Regeln von C++20 die Definition von Vergleichsoperatoren zum Teil erheblich vereinfachen, führen sie gelegentlich auch dazu, dass Code, der bis C++17 korrekt funktioniert hat, mit C++20 nicht mehr funktioniert, weil Vergleiche aufgrund der neuen Regeln plötzlich anders als zuvor interpretiert werden.

## Beispiel

```
struct A { ..... };
struct B : A { ..... };

bool operator== (A x, A y) { ..... }   // AA
bool operator== (A x, B y) { ..... }   // AB

// Ausdruck: // Aufruf von Operator AA oder AB?
A a; B b;    // Bis C++17:          // Ab C++20:

a == b;      // AB                  // AB

b == a;      // AA                  // AB mit vertauschten Operanden
             // (AB passt nicht)    // (passt besser als AA)

b == b;      // AB                  // AB, aber eigentlich mehrdeutig:
                                    // AB oder
                                    // AB mit vertauschten Operanden
```

# 7 Container und Iteratoren

## 7.1 Standardcontainer

### 7.1.1 Containertypen

#### Sequentielle Containertypen

- `array<T, N>` (seit C++11)
  Lediglich eine Verpackung des Reihentyps `T [N]`, die analog zu anderen Containertypen verwendet werden kann.

- `vector<T>`
  Eine Reihe, die bei Bedarf automatisch vergrößert wird.

  `vector<bool>` ist eine Spezialisierung, die die Elemente kompakt in einem Bitvektor speichert.

- `deque<T>` (double-ended queue)
  Typischerweise eine Reihe von Reihen, die bei Bedarf automatisch vergrößert wird und bei der Elemente an beiden „Enden“ effizient hinzugefügt und entfernt werden können.

- `forward_list<T>` (seit C++11)
  Eine einfach verkettete Liste.

- `list<T>`
  Eine doppelt verkettete Liste.

### Adapter für sequentielle Containertypen

- `stack<T>`
  Ein LIFO-Container (last in, first out).

- `queue<T>`
  Ein FIFO-Container (first in, first out).

- `priority_queue<T>`
  Eine Vorrangwarteschlange.

### Weitere sequenzartige Typen

- `bitset<N>`
  Logisch eine Verpackung des Reihentyps `bool [N]`, dessen Elemente aber (wie bei `vector<bool>`) kompakt in einem Bitvektor gespeichert werden.

- `basic_string<T>`
  Eine Sequenz von Elementen mit zusätzlichen Funktionen wie z. B. `substr` und `operator+`.

  `string` ist lediglich ein Alias für `basic_string<char>`, `wstring` ein Alias für `basic_string<wchar_t>` usw.

## Assoziative Containertypen

- `set<Key>`
  Eine (typischerweise als Baum implementierte) geordnete Menge, die jedes Element nur einmal enthalten kann.

- `multiset<Key>`
  Eine (typischerweise als Baum implementierte) geordnete Menge, die Elemente auch mehrfach enthalten kann.

- `unordered_set<Key>` (seit C++11)
  Eine als Streuwerttabelle (hash table) implementierte Menge, die jedes Element nur einmal enthalten kann.

- `unordered_multiset<Key>` (seit C++11)
  Eine als Streuwerttabelle (hash table) implementierte Menge, die Elemente auch mehrfach enthalten kann.

- `map<Key, T>`
  Eine (typischerweise als Baum implementierte) geordnete Tabelle von Schlüssel-Wert-Paaren, die jeden Schlüssel nur einmal enthalten kann.

- `multimap<Key, T>`
  Eine (typischerweise als Baum implementierte) geordnete Tabelle von Schlüssel-Wert-Paaren, die Schlüssel auch mehrfach enthalten kann.

- `unordered_map<Key, T>` (seit C++11)
  Eine als Streuwerttabelle (hash table) implementierte Tabelle von Schlüssel-Wert-Paaren, die jeden Schlüssel nur einmal enthalten kann.

- `unordered_multimap<Key, T>` (seit C++11)
  Eine als Streuwerttabelle (hash table) implementierte Tabelle von Schlüssel-Wert-Paaren, die Schlüssel auch mehrfach enthalten kann.

**Anmerkungen**

- Anders als in Java und anderen Sprachen, gibt es keine gemeinsame (abstrakte) Basisklasse wie `Collection` o. ä.

- Alle Operationen sind aus Effizienzgründen bewusst nicht-virtuell, d. h. es findet kein dynamisches Binden statt.

- Die Palette der angebotenen Operationen kann je nach Containertyp sehr unterschiedlich sein.
- Die gleiche Operation kann je nach Containertyp sehr unterschiedliche Laufzeit besitzen.
- Deshalb ist die Wahl eines bestimmten Containertyps eine weitreichende Entwurfsentscheidung, die oft nur mit großem Aufwand geändert werden kann.
- Die Adaptertypen `stack`, `queue` und `priority_queue` bieten lediglich angepasste und eingeschränkte Schnittstellen zu einem anderen sequentiellen Containertyp an.
- Eine Menge entspricht logisch und implementierungstechnisch einer Tabelle, die nur Schlüssel und keine zugehörigen Werte speichert.
- Die Namen sind zum Teil historisch begründet:
  - Weil die als Streuwerttabellen implementierten Mengen und Tabellen erst 2011 zur Standardbibliothek hinzugefügt wurden und zu diesem Zeitpunkt bereits viele andere Bibliotheken Typen wie `hash_set` und `hash_map` anboten, wurden vorsichtshalber andere Namen gewählt. (Und die geordneten Container wurden aus Kompatibilitätsgründen natürlich nicht in `ordered_set` etc. umbenannt.)
  - Da es vor 2011 nur doppelt verkettete Listen gab, wurden diese einfach `list` (und nicht etwa `bidirectional_list` o. ä.) genannt (und später nicht umbenannt).

## 7.1.2 Wichtige Operationen auf Containern

- `c.size()` liefert die Größe des Containers `c`, d. h. die Anzahl der Elemente, die er momentan enthält.
  (Nicht zu verwechseln mit der momentanen *Kapazität* eines Vektors, die größer sein kann.)

- `c.empty()` liefert genau dann `true`, wenn der Container `c` momentan leer ist, d. h. keine Elemente enthält.

- `c.front()` bzw. `c.back()` liefert eine Referenz auf das erste bzw. letzte Element des Containers `c`.
  Wenn `c` leer ist, ist das Verhalten undefiniert.

- Für `i` zwischen 0 einschließlich und `c.size()` ausschließlich liefern `c[i]` und `c.at(i)` eine Referenz auf das `i`-te Element des sequentiellen Containers `c`.
  Wenn `i` größer oder gleich `c.size()` ist, ist das Verhalten von `c[i]` undefiniert, während `c.at(i)` eine Ausnahme des Typs `out_of_range` wirft.
  (Da der Parametertyp `size_t` vorzeichenlos ist, kann der Fall `i` kleiner 0 prinzipiell nicht auftreten, weil negative Werte in positive umgewandelt werden.)

- Für einen Schlüssel `k` liefern `c[k]` und `c.at(k)` eine Referenz auf den zu `k` gehörenden Wert der Tabelle (`map` oder `unordered_map`) `c`.
  Wenn die Tabelle keinen Eintrag mit Schlüssel `k` enthält, erstellt `c[k]` einen entsprechenden Eintrag mit Wert `T()` und liefert eine Referenz auf diesen Wert, während `c.at(k)` eine Ausnahme des Typs `out_of_range` wirft.

- `c.push_front(x)` bzw. `c.push_back(x)` fügt das Element `x` am Anfang bzw. am Ende des sequentiellen Containers `c` hinzu.

- `c.pop_front()` bzw. `c.pop_back()` entfernt das erste bzw. letzte Element des sequentiellen Containers `c`.
  Wenn `c` leer ist, ist das Verhalten undefiniert.

- `c.insert(i, x)` (bzw. `c.insert_after(i, x)` bei einer `forward_list` `c`) fügt das Element `x` vor (bzw. nach) der durch den Iterator `i` (vgl. § 7.2.1) bezeichneten Position in den sequentiellen Container `c` ein und liefert einen Iterator auf das eingefügte Element.

- `c.insert(x)` fügt das Element `x` in den assoziativen Container `c` ein, sofern es noch nicht enthalten ist oder der Container Elemente mehrfach enthalten kann.
  Wenn `c` eine Tabelle ist, muss `x` ein `std::pair<const Key, T>` sein.

- `c.erase(i)` (bzw. `c.erase_after(i)` bei einer `forward_list` `c`) entfernt das durch den Iterator `i` bezeichnete Element (bzw. das Element danach) aus dem Container `c`.

- `c.erase(x)` bzw. `c.erase(k)` entfernt das Element `x` bzw. das Element mit Schlüssel `k` aus der Menge bzw. Tabelle `c`, sofern ein solches Element enthalten ist. Falls mehrere solche Elemente enthalten sind, werden alle entfernt.

## Verfügbarkeit und Laufzeit der Operationen

| | array | vector | deque | list | forward_list | [multi] set/map | unordered_ [multi]set/map |
|---|---|---|---|---|---|---|---|
| front | kon. | kon. | kon. | kon. | kon. | | |
| back | kon. | kon. | kon. | kon. | | | |
| []<br>at | kon. | kon. | kon. | | | log.[4] | kon.[4] |
| push_front<br>pop_front | | | kon.[1] | kon. | kon. | | |
| push_back<br>pop_back | | kon.[1] | kon.[1] | kon. | | | |
| insert<br>erase | | lin.[2] | lin.[2] | kon. | kon.[3] | log. | kon. |

- „kon.“ bedeutet konstante Laufzeit.
- „lin.“ bedeutet lineare Laufzeit bezogen auf die momentane Größe des Containers.
- „log.“ bedeutet logarithmische Laufzeit bezogen auf die die momentane Größe des Containers.
- Ein leerer Eintrag bedeutet, dass es die entsprechende Operation nicht gibt.

## Fußnoten zur Tabelle

1. Die Laufzeit der `push`-Operationen bei `vector` und `deque` ist *im Durchschnitt* konstant.
   Einzelne Operationen, die eine Vergrößerung des internen Speichers erfordern, können jedoch lineare Laufzeit besitzen.

2. Die Laufzeiten von `insert` und `erase` bei `vector` und `deque` beziehen sich auf den *allgemeinen* Fall.
   Wenn eine solche Operation gleichbedeutend mit einer (bei diesem Containertyp verfügbaren) `push`- bzw. `pop`-Operation ist, ist ihre Laufzeit (im Durchschnitt) konstant.
   Bei `vector` ist die Laufzeit von `insert` und `erase` allgemein proportional zur Anzahl der *nachfolgenden* Elemente.

3. Bei `forward_list` heißen die Operationen nicht `insert` und `erase`, sondern `insert_after` und `erase_after`.

4. Von den insgesamt acht assoziativen Containertypen besitzen nur `map` und `unordered_map` die Operationen `[]` und `at`.

## 7.1.3 Verwendung geordneter Mengen und Tabellen

- Die Containertypen `set`, `multiset`, `map` und `multimap` verwenden standardmäßig den Kleiner-Operator des Element- bzw. Schlüsseltyps `Key` zur Sortierung der Einträge.

- Die Gleichheit zweier Elemente bzw. Schlüssel `x` und `y` wird indirekt mittels `!(x < y) && !(y < x)`, d. h. durch zwei Aufrufe des Kleiner-Operators überprüft.
(Obwohl die Gleichheit zweier Elemente seit C++20 mit nur einem Aufruf des Dreiwegvergleichs `<=>` effizienter überprüft werden könnte, wird nach wie vor der Kleiner-Operator verwendet.)

- Für einen benutzerdefinierten Typ muss ggf. ein passender Kleiner-Operator definiert werden, zum Beispiel (vgl. § 5.1.2):

```
bool operator< (Rational x, Rational y) {
  return double(x.num)/x.den < double(y.num)/y.den;
}
```

- Um ein anderes Vergleichskriterium zu verwenden, kann als zusätzlicher Schablonenparameter eine Klasse angegeben werden, die eine konstante Elementfunktion `operator()` mit zwei Parametern des Element- bzw. Schlüsseltyps `Key` und Resultattyp `bool` besitzt, die dann anstelle des Kleiner-Operators verwendet wird.

## Beispiel

```
using str = const char*;

set<str> ss1;

struct StrLess {
  bool operator() (str s1, str s2) const {
    return strcmp(s1, s2) < 0;
  }
};

set<str, StrLess> ss2;
```

- Die Menge `ss1` verwendet zum Vergleich von Elementen den Kleiner-Operator für Zeiger, was vermutlich nicht sinnvoll ist, weil inhaltlich gleiche Zeichenketten mit verschiedenen Adressen dann als „ungleich“ behandelt werden.

- Die Menge `ss2` hingegen verwendet die in der Klasse `StrLess` definierte Vergleichsfunktion, die einen inhaltlichen Vergleich der Zeichenketten durchführt.

## 7.1.4 Verwendung ungeordneter Mengen und Tabellen

- Die Containertypen `unordered_set`, `unordered_multiset`, `unordered_map` und `unordered_multimap` verwenden standardmäßig den Gleichheitsoperator des Element- bzw. Schlüsseltyps `Key` zum Vergleich von Einträgen sowie die Elementfunktion `operator()` der Bibliotheksklasse `std::hash<Key>` (sofern diese existiert) zur Berechnung von Streuwerten.

- Für einen benutzerdefinierten Typ müssen diese Funktionen ggf. passend definiert werden, zum Beispiel (vgl. § 5.1.2):

```
bool operator== (Rational x, Rational y) {
  return x.num * y.den == y.num * x.den;
}

template <>
struct std::hash<Rational> {
  size_t operator() (Rational x) const {
    return x.num << 16 | x.den;
  }
}
```

(Sowohl `std::` als auch `const` ist notwendig!)

- Um andere Funktionen zu verwenden, können als zusätzliche Schablonenparameter zwei Klassen `Hash` und `Pred` angegeben werden, die jeweils eine passende Elementfunktion `operator()` besitzen.

- In jedem Fall müssen die Funktionen zum Vergleich von Elementen und zur Berechnung von Streuwerten semantisch zusammenpassen, d. h. Objekte, die gemäß der Vergleichsfunktion „gleich“ sind, müssen den gleichen Streuwert besitzen.

**Beispiel (vgl. § 7.1.3)**

```
using str = const char*;

unordered_set<str> ss1;

struct StrEqual {
  bool operator() (str s1, str s2) const {
    return strcmp(s1, s2) == 0;
  }
};
```

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

unordered_set<str, StrHash, StrEqual> ss2;
```

- Die Menge `ss1` verwendet zum Vergleich von Elementen den Gleichheitsoperator für Zeiger und zur Berechnung von Streuwerten die entsprechende Spezialisierung von `std::hash`, was – ähnlich wie in § 7.1.3 – vermutlich beides nicht sinnvoll (aber zusammen semantisch korrekt) ist.

- Die Menge `ss2` hingegen verwendet die in den Klassen `StrEqual` und `StrHash` definierten Funktionen, die jeweils den Inhalt der vorliegenden Zeichenketten berücksichtigen.

# 7.2 Iteratoren

## 7.2.1 Grundprinzip

- *Iteratoren* sind eine Verallgemeinerung von Zeigern: Obwohl es sich um Objekte beliebiger Typen handeln kann, können sie mehr oder weniger wie Zeiger verwendet werden, weil Operatoren wie `*` und `++` bei Bedarf entsprechend überladen sind.

- Ein Iterator `i` verweist normalerweise auf ein Element irgendeiner *Folge* von Objekten, beispielsweise eines Containers.
  In diesem Fall liefert `*i` dieses Element (je nach Typ des Iterators als Referenz, `const`-Referenz oder als Wert).

- Ein Iterator kann aber z. B. auch auf das Ende einer Folge verweisen, d. h. auf die Position *nach dem letzten Element* der Folge (vgl. § 2.3.7).
  In diesem Fall ist das Verhalten von `*i` undefiniert.

- Analog zu einem Zeiger, kann ein *Vorwärtsiterator* (forward iterator) `i`, der auf ein Element einer Folge verweist, mittels `++i` oder `i++` inkrementiert werden, damit er auf das nächste Element der Folge verweist. (Das heißt, ein Vorwärtsiteratortyp muss sowohl Präfix- als auch Postfix-Inkrement unterstützen. `++i` liefert den Iterator, der auf das nächste Element der Folge verweist, `i++` den ursprünglichen Iterator.)

- Außerdem können zwei Iteratoren `i` und `j` mittels `i == j` und `i != j` verglichen werden. (Das heißt, jeder Iteratortyp muss sowohl Gleichheit als auch Ungleichheit unterstützen, wofür seit C++20 aber die Definition des Gleichheitsoperators ausreicht, vgl. § 6.2.2). Zwei Iteratoren sind genau dann gleich, wenn sie auf das gleiche Element bzw. auf die gleiche Position verweisen.

- Damit ermöglichen Iteratoren u. a. die Iteration über alle Elemente einer Folge, ohne dass man deren interne Datenstruktur kennen muss:
  Wenn eine Folge `s` Elementfunktionen `begin` und `end` besitzt, die Iteratoren auf das erste Element bzw. auf das o. g. Ende der Folge liefern, dann kann man z. B. wie folgt alle Elemente der Folge ausgeben:

  ```
  for (auto i = s.begin(); i != s.end(); i++) cout << *i;
  ```

  (Wenn die Folge leer ist, muss `s.begin()` den gleichen Iterator liefern wie `s.end()`.)

- Ein *bidirektionaler Iterator* (bidirectional iterator) `i`, der nicht auf das erste Element einer Folge verweist, kann mittels `--i` oder `i--` auch dekrementiert werden, damit er auf das vorige Element der Folge verweist. (Das heißt, ein bidirektionaler Iteratortyp muss – zusätzlich zu Präfix- und Postfix-Inkrement sowie Gleichheit und Ungleichheit – sowohl Präfix- als auch Postfix-Dekrement unterstützen. `--i` liefert den Iterator, der auf das vorige Element der Folge verweist, `i--` den ursprünglichen Iterator.)

- Damit könnte eine Iteration in umgekehrter Richtung wie folgt implementiert werden (siehe aber auch § 7.2.2):

```
auto i = s.end();
while (i != s.begin()) cout << *--i;
```

(Beachte die Asymmetrie von `begin` und `end`: `begin` liefert einen Iterator auf das erste Element der Folge, `end` jedoch auf die Position *nach* dem letzten Element.)

- *Iteratoren mit wahlfreiem Zugriff* (random access iterators) erlauben zusätzlich beliebige „Adressarithmetik“ analog zu Zeigern (vgl. § 2.3.7) sowie beliebige Vergleiche, d. h. für derartige Iteratoren `i` und `j` und eine ganze Zahl `n` müssen zusätzlich alle folgenden Ausdrücke möglich sein:

```
i + n          i += n          n + i
i - n          i -= n          i - j
i < j          i <= j
i > j          i >= j
```

Außerdem ist `i[n]` wie bei Zeigern äquivalent zu `*(i+n)`.

- Daraus folgt: Jeder Zeiger ist ein random access iterator (und damit auch ein bidirektionaler Iterator und ein Vorwärtsiterator).

## 7.2.2 Iteratoren der Standardcontainer

- Jeder in § 7.1.1 genannte Containertyp der Standardbibliothek (außer `bitset` und den Adaptertypen `stack`, `queue` und `priority_queue`) besitzt Elementfunktionen `begin` und `end` sowie `cbegin` und `cend` mit folgender Bedeutung:

- `begin` und `cbegin` liefern jeweils einen Iterator auf das erste Element des Containers, sofern dieser nicht leer ist; andernfalls wird der gleiche Iterator wie bei `end` bzw. `cend` geliefert.
- `end` und `cend` liefern jeweils einen Iterator auf das Ende des Containers, d. h. auf die Position nach dem letzten Element.
- Die von `cbegin` und `cend` gelieferten Iteratoren `i` sind *konstant*, d. h. `*i` liefert jeweils eine Referenz auf ein *konstantes* Element, sodass Zuweisungen an `*i` nicht möglich sind.
- Seit C++11 sind auch die von `begin` und `end` gelieferten Iteratoren `i` von Mengen konstant, weil eine Zuweisung an `*i` die „innere Ordnung“ des Containers zerstören könnte.
  (Dass dies vor C++11 nicht so festgelegt war, war offensichtlich ein Fehler. In gängigen Implementierungen war es aber trotzdem schon so. Die Iteratoren von Tabellen verweisen auf Elemente des Typs `std::pair<const Key, T>`, an deren erste Komponente grundsätzlich nicht zugewiesen werden kann.)
- Außerdem liefern `begin` und `end` für einen konstanten Container eines beliebigen Typs immer konstante Iteratoren.
- Ansonsten liefern `begin` und `end` nicht-konstante Iteratoren `i`, d. h. Zuweisungen an `*i` sind möglich.

- Alle Containertypen außer `forward_list` und den ungeordneten Mengen und Tabellen besitzen zusätzlich Elementfunktionen `rbegin` und `rend` sowie `crbegin` und `crend`, die „spiegelbildlich“ zu den zuvor genannten Funktionen funktionieren, das heißt:
  - `rbegin` und `crbegin` liefern jeweils einen Iterator auf das *letzte* Element des Containers (*nicht* auf die Position danach) bzw. den gleichen Iterator wie `rend` bzw. `crend`.
  - `rend` und `crend` liefern jeweils einen Iterator auf den *Anfang* des Containers, d. h. auf die Position *vor* dem ersten Element.
  - Wenn ein solcher *reverse iterator* inkrementiert bzw. dekrementiert wird, verweist er anschließend auf das *vorige* bzw. *nächste* Element des Containers.
  - Damit erlauben diese Funktionen die Implementierung von *Rückwärtsiterationen* nach demselben Schema wie Vorwärtsiterationen, zum Beispiel:

```
for (auto i = c.rbegin(); i != c.rend(); i++) cout << *i;
```

  - „Normale“ und „umgekehrte“ Iteratoren besitzen potentiell unterschiedliche Typen, d. h. sie können nicht miteinander kombiniert werden.

- `forward_list` bietet stattdessen zusätzliche Elementfunktionen `before_begin` sowie `cbefore_begin`, die jeweils einen (normalen) Iterator auf den Anfang der Liste liefern, d. h. auf die Position vor dem ersten Element.
  Diese sind nützlich, um mittels `insert_after` bzw. `erase_after` Elemente am Anfang einer Liste einfügen bzw. entfernen zu können.

- Die Iteratoren der Typen `array` und `vector` erlauben wahlfreien Zugriff.

- Die Iteratoren der Typen `deque` und `list` sowie der geordneten Mengen und Tabellen (`set`, `multiset`, `map`, `multimap`) sind bidirektional.

- Die Iteratoren des Typs `forward_list` sowie der ungeordneten Mengen und Tabellen (`unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`) sind lediglich Vorwärtsiteratoren.

- Das Einfügen und Entfernen von Elementen kann je nach Typ des Containers dazu führen, dass auch Iteratoren (sowie Zeiger und Referenzen) auf *andere* Elemente ungültig werden.

### 7.2.3 Operationen mit Iteratoren

Iteratoren treten an unzähligen Stellen der Standardbibliothek auf, zum Beispiel:

- Jeder Standardcontainertyp, der eine Elementfunktion `insert` (bzw. `insert_after`) zum Einfügen eines einzelnen Elements besitzt, besitzt eine weitere gleichnamige Elementfunktion, die anstelle eines Elements zwei beliebige Iteratoren `first` und `last` als Parameter erhält und alle Elemente von `first` einschließlich bis `last` ausschließlich in den Container einfügt.
- Außerdem besitzt jeder solche Containertyp einen Konstruktor, der ebenfalls zwei Iteratoren `first` und `last` als Parameter erhält und ebenfalls alle Elemente von `first` einschließlich bis `last` ausschließlich in den neuen Container einfügt. Damit kann ein Container bequem z. B. mit allen Elementen eines beliebigen anderen Containers initialisiert werden.
- Die Elementfunktion `find` von Mengen und Tabellen liefert als Resultat einen Iterator auf das gesuchte Element, sofern es gefunden wurde, bzw. den gleichen Iterator wie die Elementfunktion `end`, wenn es nicht gefunden wurde.
- Generische Algorithmen wie z. B. `find`, `count`, `copy`, `remove` und `sort` (vgl. Definitionsdatei `<algorithm>`) sowie reguläre Ausdrücke (vgl. Definitionsdatei `<regex>`) arbeiten immer auf Folgen (oder Teilfolgen) von Elementen, die durch zwei Iteratoren `first` und `last` begrenzt sind. (Für reguläre Ausdrücke gibt es zusätzlich „Bequemlichkeitsfunktionen“, die direkt auf Zeichenketten arbeiten.)

## 7.2.4 Beispiele

```
// Eine Liste von Zeichen.
list<char> ls;
......

// Alle in der Liste enthaltenen Ziffern ausgeben.
// (Die Deklaration von isdigit macht den überladenen
// Funktionsnamen im aktuellen Block eindeutig, sodass er
// an die generische Funktion find_if übergeben werden kann.
int isdigit (int);
for (auto i = ls.begin(), e = ls.end();
                          (i = find_if(i, e, isdigit)) != e; i++) {
  cout << *i << endl;
}

// Überprüfen, ob die Liste mindestens zwei
// aufeinanderfolgene Ziffern enthält.
regex r ("[0-9]{2,}");
cout << regex_search(ls.begin(), ls.end(), r) << endl;
```

## 7.2.5 Beispiel eines selbstdefinierten Iteratortyps

### Anforderungsdefinition

- Für zwei Werte `first` und `last` eines beliebigen Typs `T`, dessen Werte mittels Addition oder Inkrementoperatoren inkrementiert werden können, soll `range(first, last)` die Folge der Werte von `first` bis `last` (jeweils einschließlich) repräsentieren, ohne all diese Werte explizit zu speichern.

- Stattdessen sollen die Elemente einer solchen Folge indirekt über Iteratoren abfragbar sein.

- Zum Beispiel:

```
auto letters = range('a', 'z');
for (auto i = letters.begin(), e = letters.end(); i != e; i++) {
  cout << *i << endl;
}
```

- Oder kürzer (vgl. § 7.4):

```
for (char c : range('a', 'z')) {
  cout << c << endl;
}
```

## Mögliche Lösung

```
// Hilfstyp zur Speicherung eines Bereichs.
template <typename T>
struct Range {
  // Bereichsgrenzen.
  T first, last;
  Range (T first, T last) : first{first}, last{last} {}

  // Zugehöriger Iteratortyp.
  // Die Basisklasse iterator<forward_iterator_tag, T>
  // kennzeichnet den Typ als Vorwärtsiterator mit Elementtyp T
  // und ermöglicht die Verwendung von iterator_traits<Iter>,
  // was für viele Verwendungen notwendig ist.
  struct Iter : iterator<forward_iterator_tag, T> {
    // Element des Bereichs, auf das der Iterator verweist.
    T current;
    Iter (T current) : current{current} {}

    // Operator zur Abfrage dieses Elements.
    T operator* () const {
      return current;
    }
```

```
// Operatoren zum Inkrementieren/Weitersetzen des Iterators.
// Der Präfixoperator ist (wie erwartet) parameterlos und
// liefert den weitergesetzten Iterator *this per Referenz
// zurück.
// Der Postfixoperator muss zur Unterscheidung einen Dummy-
// Parameter mit Typ int besitzen und liefert den
// ursprünglichen Iterator i als Wert zurück.
// (Er verwendet sinnvollerweise den Präfixoperator.)
Iter& operator++ () {
  ++current;
  return *this;
}
Iter operator++ (int) {
  Iter i = *this;
  ++*this;
  return i;
}
```

```
    // Operatoren zum Vergleich mit einem anderen Iterator that.
    // (Auch hier verwendet ein Operator sinnvollerweise den
    // anderen. Ab C++20 muss der Ungleichoperator nicht mehr
    // explizit definiert werden.)
    bool operator== (const Iter& that) const {
      return current == that.current;
    }
    bool operator!= (const Iter& that) const {
      return !(*this == that);
    }
  };

  // Iteratoren auf das erste Element bzw. auf das Ende der Folge
  // (d. h. logisch auf die Position nach dem letzten Element).
  Iter begin () { return Iter{first}; }
  Iter end () { return Iter{last + 1}; }
};

// Die für den Benutzer sichtbare Funktionsschablone.
template <typename T>
Range<T> range (T first, T last) {
  return Range<T>{first, last};
}
```

# 7.3 Initialisiererlisten (initializer lists)

## 7.3.1 Prinzip

- An zahlreichen Stellen der C++-Grammatik können *Initialisiererlisten* verwendet werden, die aus beliebig vielen Ausdrücken in geschweiften Klammern bestehen.
- Der Übersetzer ersetzt eine solche Initialisiererliste durch ein Objekt `x` des Typs `initializer_list<T>`, wobei sich der Typ `T` meist aus dem Kontext ergibt.
- Die Typen der Ausdrücke müssen dann implizit in `T` umwandelbar sein, wobei Umwandlungen verboten sind, die einen Wert möglicherweise verfälschen könnten (sog. „narrowing conversions“, z. B. von `int` nach `char`, von `double` nach `int`, aber auch von `int` nach `double`).
- Ausnahme: Entsprechende Umwandlungen konstanter Werte sind erlaubt, wenn sich die umgewandelten Werte im Zieltyp exakt darstellen lassen (was zur Übersetzungszeit geprüft werden kann).
- Wenn sich der Typ `T` nicht aus dem Kontext ergibt, wird der Typ der Ausdrücke verwendet, der in diesem Fall für alle Ausdrücke gleich sein muss.

- Zur Laufzeit werden die Ausdrücke einer Initialisiererliste von links nach rechts ausgewertet und ihre Werte in eine anonyme, temporäre Reihe kopiert, die indirekt über das o. g. Objekt `x` verfügbar ist:
  - `x.size()` liefert die Anzahl der Elemente der Reihe bzw. der Initialisiererliste.
  - `x.begin()` liefert einen Zeiger (und damit auch einen random access iterator) mit Typ `const T*` auf das erste Element der Reihe.
  - `x.end()` liefert einen Zeiger (und damit auch einen random access iterator) mit Typ `const T*` auf das Ende der Reihe, d. h. `x.begin() + x.size()`.

## 7.3.2 Verwendung in der Standardbibliothek

Initialisiererlisten werden ebenfalls an zahlreichen Stellen der Standardbibliothek verwendet, zum Beispiel:

- Jeder Standardcontainertyp, der eine Elementfunktion `insert` (bzw. `insert_after`) zum Einfügen eines einzelnen Elements besitzt, besitzt eine weitere gleichnamige Elementfunktion, die anstelle eines Elements eine Initialisiererliste des Elementtyps als Parameter erhält und alle Elemente dieser Liste in den Container einfügt (vgl. auch § 7.2.3).
- Außerdem besitzt jeder solche Containertyp einen Konstruktor, der – auch implizit – mit einer Initialisiererliste des Elementtyps aufgerufen werden kann und alle Elemente dieser Liste in den neuen Container einfügt.
- Zum Beispiel:

```
vector<int> v = { 1, 2, 3, 4 };
v.insert(v.end(), { 5, 6, 7 });
list<vector<int>> ls = { { 1, 2 }, { 3, 4 } };

void f (const set<string>& names) { ..... }
f({ "Anton", "Berta", "Christian" });
```

- Die Funktionen `min`, `max` und `minmax` können entweder mit zwei Werten eines beliebigen Typs `T` oder mit einer entsprechenden Initialisiererliste aufgerufen werden.

## 7.3.3 Beispiele

- Summe und Durchschnittswert beliebig vieler Zahlen (vgl. § 5.5.1):

```
double sum (initializer_list<double> il) {
  double s = 0;
  for (double x : il) s += x;
  return s;
}

double avg (initializer_list<double> il) {
  if (il.size() == 0) throw invalid_argument("empty list");
  return sum(il) / il.size();
}
```

- Verwendungsmöglichkeiten:

```
sum({ 1.5, 2.7, -3.8 });

int x, y, z;
......
sum({ x, y, z });        // Fehler: narrowing conversions.
sum({ 1, 2, 3 });        // OK: 1, 2, 3 lassen sich exakt
                         // als double-Werte darstellen.
```

# 7.4 Schleifen über Bereiche (range-based for loops)

## 7.4.1 Schleifen über Reihen

- Für eine Reihe `a` mit `n` Elementen ist eine Schleife der Gestalt

```
for (T x : a) ......
```

  gleichbedeutend mit

```
for (auto i = a, e = i + n; i != e; ++i) {
  T x = *i;
  ......
}
```

  d. h. `x` durchläuft nacheinander alle Elemente von `a`.

- Um eventuelle Namenskonflikte zu vermeiden, bleiben die Namen der Hilfsvariablen `i` und `e` jedoch verborgen.

- Durch die Verwendung des Schlüsselworts `auto` kann der Code unabhängig vom konkreten Elementtyp der Reihe formuliert werden. (Der Typ von `i` und `e` ist dann der entsprechende Zeigertyp.)

## 7.4.2 Schleifen über andere Objekte

- Wenn der Typ von `a` eine Klasse mit parameterlosen Elementfunktionen `begin` und `end` ist (typischerweise ein Containertyp oder eine Initialisiererliste), ist eine Schleife der Gestalt

  ```
  for (T x : a) ......
  ```

  gleichbedeutend mit

  ```
  for (auto i = a.begin(), e = a.end(); i != e; ++i) {
    T x = *i;
    ......
  }
  ```

  d. h. wenn die Elementfunktionen die „übliche“ Bedeutung haben (vgl. § 7.2.2 und § 7.3.1), durchläuft `x` der Reihe nach alle Elemente von `a`.

- Obwohl der Ausdruck `a` in der „Übersetzung“ zweimal auftritt, wird er zur Laufzeit jedoch nur einmal ausgewertet.

- Wenn der Typ von `a` keine Klasse ist oder die Klasse keine parameterlosen Elementfunktionen `begin` und `end` besitzt, werden `a.begin()` und `a.end()` ersetzt durch `begin(a)` und `end(a)`.

- Wenn es dann keine passenden globalen Funktionen `begin` und `end` (in sog. „associated namespaces“ des Typs von `a`) gibt, erhält man einen Übersetzungsfehler.

### 7.4.3 Anmerkungen

- Nach den üblichen Regeln (vgl. § 3.3.3), wird am Anfang bzw. Ende jedes Schleifendurchlaufs der Konstruktor bzw. Destruktor von `x` ausgeführt.
- Die Variable `x` muss auf diese Weise lokal in der Schleife deklariert werden, d. h. es kann keine zuvor deklarierte Variable verwendet werden.
- Der Typ `T` kann auch ein ggf. `const`-qualifizierter Referenztyp sein.
- Anstelle von `T` kann auch `auto` verwendet werden.

### 7.4.4 Beispiele

- Ausgabe einer Menge:

```
set<int> s;
......

// Formulierung mit explizitem Iterator in C++98.
for (set<int>::iterator i = s.begin(); i != s.end(); ++i) {
  int x = *i;
  cout << x << endl;
}
```

```
// Vereinfachung durch auto in C++11.
for (auto i = s.begin(); i != s.end(); ++i) {
  int x = *i;
  cout << x << endl;
}

// Kompakte Formulierung mit verborgenem Iterator.
for (int x : s) {
  cout << x << endl;
}
```

- Verdopplung aller Elemente eines Vektors:

```
vector<int> v;
......

for (int& x : v) x *= 2;
```

- Ausgabe eines Containers mit unbekanntem Typ:

```
template <typename C>
void print (const C& c) {
  for (auto& x : c) cout << x << endl;
}
```

# 8 Ausblick

- Implizite Typumwandlungen
- Ausnahmen
- Ein- und Ausgabe
- `constexpr`
- `shared_ptr`, `weak_ptr`, `unique_ptr`
- SFINAE (substitution failure is not an error),
  ADFINAE (argument deduction failure is not an error)
- Threads, Mutexes, Condition variables
- Module
- Koroutinen
- u. v. a. m.