# *lib*CH

*C++-Bibliothek*
*von*
*Prof. Dr. Christian Heinlein*

26. Oktober 2022

# 1 Einleitung

Die hier beschriebene Bibliothek befindet sich in Dateien mit der Endung `.ch`, und ihr gesamter Code befindet sich im Namensraum `CH`.

Da es für Makros keine Namensräume gibt, beginnen alle Makronamen vorsichtshalber mit dem Präfix `CH_`, z. B. `CH_TYPE`. Wenn beim Einbinden einer Definitionsdatei das Makro `USING_CH` definiert ist, werden aber zusätzlich „Aliase“ ohne dieses Präfix definiert, z. B. `TYPE`. Bei der nachfolgenden Beschreibung werden die Präfixe `CH::` und `CH_` der Einfachheit halber weggelassen.

Damit durch `using namespace CH` in einem Anwendungsprogramm keine internen Namen des Namensraums `CH` sichtbar werden (die dann zu Konflikten führen können), beginnen interne Namen ebenfalls mit dem Präfix `CH_`. (Die Verlagerung interner Namen in einen untergeordneten Namensraum würde dieses Problem nicht vollständig lösen, weil dann zumindest der Name dieses Unternamensraums sichtbar wäre.)

Die Implementierung verwendet zahlreiche Sprachmittel von C++11, wie z. B. range-based for loops, variadic templates, perfect forwarding usw., sowie einige von C++14 und C++17, z. B. fold expressions und deduction guides. Deshalb braucht man einen Compiler, der C++17 ausreichend fehlerfrei unterstützt, z. B. `g++` ab Version 9 oder `clang++` ab Version 7.

`g++` Version 7 und 8 akzeptieren deduction guides mit leerer Parameterliste nicht. `clang++` Version 5 hat möglicherweise ein Problem mit variadic deduction guides. Deshalb funktionieren diese Compiler-Versionen nicht.

`clang++` Version 6 wurde nicht getestet.

Der Code wurde konkret getestet mit:

- `g++` 7.5.0 (funktioniert nicht)
- `g++` 8.2.1 (funktioniert nicht)
- `g++` 9.3.1 (funktioniert)
- `g++` 10.2.1 (funktioniert)
- `g++` 12.1.1 (funktioniert)
- `clang++` 5.0.2. (funktioniert nicht)
- `clang++` 7.0.1. (funktioniert)
- `clang++` 9.0.1. (funktioniert)
- `clang++` 13.0.1. (funktioniert)
- `msvc` 19.28 auf `godbolt.org` (funktioniert)

# 2 Hilfsmittel für Template-Metaprogrammierung (`meta.ch`)

Siehe Kommentare in der Implementierung.

# 3 Weitere Hilfsmittel (`util.ch`)

## 3.1 Nilwerte

Die Konstante `nil` repräsentiert einen generischen Nilwert, der prinzipiell in jeden Typ `T` umgewandelt werden kann. Diese Umwandlung liefert den Wert `nil_v<T>`, der standardmäßig gleich `T()` ist. Wenn es für einen Typ `T` keinen parameterlosen Konstruktor gibt oder dieser wider Erwarten nicht den gewünschten Wert liefert, kann die Variablenschablone `nil_v` geeignet spezialisiert werden.

## 3.2 `bool`-artige Werte

Der Typ `boollike` ist ein Synonym für einen elementaren Typ wie z. B. `void*`, der implizit nach `bool` umgewandelt werden kann und dessen Werte deshalb wie `bool`-Werte verwendet werden können. Andererseits kann dieser Typ in (möglichst) keinen anderen Typ umgewandelt werden, und er besitzt (möglichst) keine anwendbaren Operationen (wie z. B. arithmetische Operatoren).

Wenn ein benutzerdefinierter Typ eine implizite Umwandlung nach `boollike` statt nach `bool` besitzt, können seine Objekte zwar wie `bool`-Werte (z. B. in Schleifenbedingungen) verwendet werden, aber nicht als numerische Werte, was oft zu unerwünschten Mehrdeutigkeiten führen würde.

Für einen `bool`-Wert `b` liefert `boollikeval(b)` den zu `b` korrespondierenden Wert des Typs `boollike`.

*Anmerkung:* C++11 bietet mit `explicit operator bool` anstelle von `operator bool` eine ähnliche, aber eingeschränktere Möglichkeit: Laut en.cppreference.com/w/cpp/language/cast_operator wird eine durch `explicit operator bool` definierte Umwandlung nur bei „direct initializations“ und „explicit conversions“ verwendet, wobei die Bedingungen von `if`, `while` und `for` sowie die Operanden der vordefinierten Operatoren `!`, `&&` und `||` und der erste Operand von `?:` sozusagen direkt initialisiert werden (contextual conversion to bool). Bei der Übergabe von Parameterwerten und der Rückgabe von Funktionswerten, die als „copy initializations“ gelten, werden solche Umwandlungen jedoch nicht verwendet. Das bedeutet zum Beispiel, dass Aufrufe `implies(x, y)` der folgenden Funktion nicht korrekt sind, wenn der Typ von `x` und `y` eine Umwandlung `explicit operator bool` besitzt, aber dass sie korrekt sind, wenn er stattdessen eine Umwandlung `operator boollike` besitzt, obwohl der logisch äquivalente Ausdruck `!x || y` in beiden Fällen korrekt ist:

```
bool implies (bool x, bool y) { return !x || y; }
```

## 3.3 Dreiwertige Logik

Der Typ `bool3` realisiert eine dreiwertige Logik mit den Wahrheitswerten `false` (sicher nicht wahr), `maybe` (eventuell wahr) und `true` (sicher wahr), wobei `false` kleiner als `maybe` und `maybe` kleiner als `true` ist.

`bool3` ist kompatibel mit `bool`, d. h. die `bool`-Werte `false` und `true` können auch als `bool3`-Werte verwendet werden. Umgekehrt kann ein `bool3`-Wert auch als `bool`-Wert verwendet werden, der genau dann `true` ist, wenn der `bool3`-Wert `true` ist, d. h. `false` und `maybe` werden beide auf den `bool`-Wert `false` abgebildet.

Die Und- bzw. Oder-Verknüpfung (Operatoren `&` und `&&` bzw. `|` und `||`) zweier Werte des Typs `bool3` (oder auch `bool`) liefert das Minimum bzw. Maximum der beiden Werte.

Die Negation (Operatoren `~` und `!`) von `true` bzw. `false` ist wie gewohnt `false` bzw. `true`, die Negation von `maybe` ist `maybe`.

## 3.4 Unveränderte Weitergabe von Makroargumenten

Wenn ein Text `xyz` Kommas enthält (z. B. `std::pair<int, int>`), wird er bei der Übergabe an ein Makro in mehrere Makroargumente zerlegt. Um dies zu verhindern, kann er mit dem Makro `LIT` geklammert werden: `LIT(xyz)`.

## 3.5 Garantiert initialisierte Variablen

`SIGVAR(T, name, ...)` bzw. `SITVAR(T, name, ...)` definiert faktisch eine globale bzw. Thread-lokale Variable mit dem Typ `T` und dem Namen `name`, die garantiert bei ihrer ersten Verwendung initialisiert ist, was bei normalen derartigen Variablen nicht garantiert ist, wenn sie in unterschiedlichen Übersetzungseinheiten definiert sind. (Hier garantiert der Standard nur, dass sie innerhalb derselben Übersetzungseinheit in der Reihenfolge ihrer Definition und ansonsten *entweder* vor ihrer ersten Verwendung *oder* vor der Ausführung von `main` initialisiert werden. GCC und vermutlich auch Clang implementieren wohl die zweite Variante. Wenn während der Initialisierung einer solchen Variablen direkt oder indirekt eine andere solche Variable verwendet wird, kann es tatsächlich passieren, dass die andere Variable noch nicht initialisiert ist, was dann zu sehr merkwürdigen und schwer zu findenden Fehlern führt.)

Tatsächlich wird jeweils eine parameterlose Funktion mit dem Namen `name` und Resultattyp `T&` definiert, die eine entsprechende Variable `static` bzw. `thread_local` definiert und per Referenz zurückgibt.

Weitere optionale Argumente `...` werden zur Initialisierung der Variablen mit geschweiften Klammern verwendet.

`SIGVAR` bzw. `SITVAR` steht für „surely initialized global/thread-local variable“.

Durch eine vorangestellte `template`-Klausel können entsprechende Variablenschablonen definiert werden.

## 3.6 Verhüllen von Typen

`wrap<T>::type` sowie `wrap_t<T>` sind Synonyme für `T`, vgl. `type_identity` in C++20.

Die Verwendung von `wrap_t<T>` anstelle von `T` ist nützlich, wenn `T` an bestimmten Stellen nicht direkt verwendet werden kann, z. B. `wrap_t<void (*) ()> p`, und wenn aus `T` bewusst keine Schablonenparameter deduziert sollen, zum Beispiel:

```
template <typename X>
vector<X>& operator+= (vector<X>& v, const wrap_t<X>& x) {
    v.push_back(x);
    return v;
}
```

Hier genügt es, wenn der Typ `X` aus dem Vektor `v` deduziert wird, während der Typ von `x` nicht exakt `X` sein muss.

Die erste oben genannte Verwendung ist insbesondere in Makros sinnvoll, wenn man `T` nicht genau kennt.

# 4 Ablaufverfolgung

Die Klasse `Trace` ermöglicht eine einfache Ablaufverfolgung, die zur Übersetzungszeit durch Definition des Makros `CH_TRACE` (z. B. mittels `-D CH_TRACE`) für ein Programm oder eine Quelldatei aktiviert werden kann.

Die Klasse verwaltet eine aktuelle Einrückung, die anfangs leer ist und für jede Ebene um 2 Leerzeichen erhöht wird.

Der Konstruktor der Klasse kann mit beliebig vielen beliebigen Parametern aufgerufen werden. Er gibt eine Zeile mit der aktuellen Einrückung aus, die aus einer öffnenden geschweiften Klammer und den Werten der Parameter, jeweils getrennt durch ein Leerzeichen, besteht. (Für die Typen der Parameter muss es hierfür geeignete Definitionen des Ausgabeoperators << geben.) Anschließend wird die aktuelle Einrückung um eine Ebene erhöht.

Der Destruktor der Klasse (der üblicherweise implizit aufgerufen wird) reduziert die aktuelle Einrückung um eine Ebene und gibt dann eine Zeile mit der neuen aktuellen Einrückung aus, die aus einer schließenden geschweiften Klammer besteht.

Der Klammeroperator der Klasse kann ebenfalls mit beliebig vielen beliebigen Parametern aufgerufen werden. Er gibt eine Zeile mit der aktuellen Einrückung aus, die aus den Werten der Parameter, jeweils getrennt durch ein Leerzeichen, besteht.

Alle Ausgaben gehen auf `std::clog`, das genauso wie `std::cerr` dem `stderr` von C und damit Filedeskriptor 2 entspricht.

Das Makro `TRACE_BLOCK` kann mit beliebig vielen beliebigen Parametern aufgerufen werden. Es deklariert eine Variable des Typs `Trace` mit einem internen Namen und gibt seine Parameter an den aufgerufenen Konstruktor weiter.

Das Makro `TRACE_LINE` kann ebenfalls mit beliebig vielen beliebigen Parametern aufgerufen werden. Es ruft den Klammeroperator auf der zuvor mittels `TRACE_BLOCK` im selben Anweisungsblock definierten Variablen auf und gibt seine Parameter an ihn weiter.

Diese beiden Makros werden allerdings nur wie beschrieben definiert, wenn beim Einbinden von `trace.ch` das Makro `CH_TRACE` definiert ist. Andernfalls ist die Definition beider Makros leer und ihre Verwendung damit wirkungslos.

Eine typische Verwendung sieht daher wie folgt aus:

```
#include <iostream>
#include "trace.ch"

void countevenodd (int a [], int n) {
    CH_TRACE_BLOCK("countevenodd", n)
    int even = 0, odd = 0;
    for (int i = 0; i < n; i++) {
        CH_TRACE_BLOCK("for", i, a[i])
        if (a[i] % 2 == 0) {
            even++;
```

```
            CH_TRACE_LINE("even", even);
        }
        else {
            odd++;
            CH_TRACE_LINE("odd", odd);
        }
    }
    std::cout << even << " " << odd << std::endl;
}
```

# 5 Generische Funktionsobjekte (`func.ch`)

Siehe Kommentare in der Implementierung.

# 6 Sequenzen (`seq.ch`)

Für einen Elementtyp `T` stellt ein Objekt des Typs `seq<T>` eine Sequenz von Objekten des Typs `T` dar.

Als Sonderfall stellt ein Objekt des Typs `seq<void>` immer eine „generische“ leere Sequenz dar, die implizit in jeden anderen Typ `seq<T>` umgewandelt werden kann.

Im folgenden bezeichnet `s` immer eine Sequenz eines solchen Typs `seq<T>`.

## 6.1 Konstruktoren und einfache Abfragen

Der Typ `seq<T>` besitzt folgende Konstruktoren:

- `seq<T>()` liefert eine leere Sequenz.
  Dieser Konstruktor kann auch implizit aufgerufen werden.
  Wenn man den Typparameter `T` weglässt, erhält man eine generische leere Sequenz mit Typ `seq<void>`.
- Für einen Wert `x` mit Typ `seq<void>` liefert `seq<T>(x)` ebenfalls eine leere Sequenz.
  Dieser Konstruktor kann auch implizit aufgerufen werden.
- Für einen oder mehrere Werte `x` mit Typ `T` (bzw. beliebigen Typen, die jeweils implizit in `T` umgewandelt werden können) liefert `seq<T>(x ...)` eine Sequenz mit den Elementen `x`.
  Dieser Konstruktor muss explizit aufgerufen werden.
  Wenn man den Typparameter `T` weglässt, müssen die Werte `x` einen „gemeinsamen“ Typ gemäß `std::common_type` besitzen, der dann als Elementtyp der Sequenz verwendet wird.
- Für eine Initialisiererliste `xs` mit Typ `std::initializer_list<T>` liefert `seq<T>(xs)` eine Sequenz mit den Elementen von `xs`.
  Dieser Konstruktor kann auch implizit aufgerufen werden.
- Für `xs` mit einem beliebigen anderen Typ, der nicht implizit in `T` umgewandelt werden kann, liefert `seq<T>(xs)` eine Sequenz mit den Elementen `x`, die sich aus der Anweisung

  ```
  for (auto x : xs) ......
  ```

  ergeben. Falls diese Anweisung nicht korrekt ist, führt die Verwendung dieses Konstruktors zu einem Übersetzungsfehler.
  Insbesondere kann `xs` ein beliebiger STL-Container sein.
  Dieser Konstruktor muss explizit aufgerufen werden. Aus diesem Grund gibt es zusätzlich den Konstruktor für eine Initialisiererliste, obwohl er nur ein Spezialfall dieses Konstruktors ist.

- Für Iteratoren `b` und `e` eines beliebigen Typs, der nicht implizit in `T` umgewandelt werden kann, liefert `seq<T>(b, e)` eine Sequenz mit den Elementen `x`, die sich aus der Anweisung

  ```
  for (auto i = b; i != e; i++) {
      auto x = *i;
      ......
  }
  ```

  ergeben. Falls diese Anweisung nicht korrekt ist, führt die Verwendung dieses Konstruktors zu einem Übersetzungsfehler.
  Insbesondere können `b` und `e` beliebige STL-Iteratoren sein.

Für eine Sequenz `s` liefert `*s` die Länge von `s`, d. h. die Anzahl ihrer Elemente.

Eine Sequenz `s` kann implizit als `bool`-Wert verwendet werden, der genau dann `true` ist, wenn die Sequenz nicht leer ist.

## 6.2 Verkettung von Sequenzen und Elementen

Für zwei Sequenzen `s1` und `s2` desselben Typs `seq<T>` liefert `s1+s2` eine neue Sequenz mit allen Elementen von `s1` und `s2` (in dieser Reihenfolge). Wenn die Elementtypen von `s1` und `s2` verschieden sind, ist der Elementtyp der Resultatsequenz gemäß `std::common_type` definiert.

Für eine Sequenz `s` und einen Wert `x` mit einem beliebigem Typ, der nach `T` umwandelbar ist, ist `s+x` bzw. `x+s` gleichbedeutend mit `s+seq<T>(x)` bzw. `seq<T>(x)+s`.

`s1+=s2` bzw. `s+=x` ist äquivalent zu `s1=s1+s2` bzw. `s=s+x`.

## 6.3 Positionen

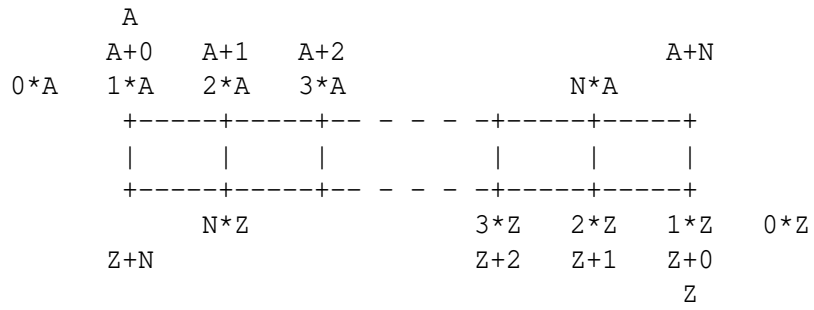
Eine Position bezeichnet logisch eine Stelle zwischen zwei Elementen (oder unmittelbar vor dem ersten bzw. nach dem letzten Element). Das bedeutet umgekehrt, dass sich ein Element immer zwischen zwei benachbarten Positionen befindet.

Die Konstante `A` mit Typ `posA` bezeichnet den Anfang einer beliebigen Sequenz, d. h. die Position vor dem ersten Element.

Für eine Position `p` mit Typ `posA` und einen ganzzahligen Wert `n` besitzen `p+n` (oder `n+p`) und `p-n` ebenfalls Typ `posA` und bezeichnen die Position, die sich `n` Elemente rechts bzw. links von Position `p` (bzw. `-n` Elemente links bzw. rechts von `p`, falls `n` negativ ist) befindet. `n*A` (oder `A*n`) ist gleichbedeutend mit `A+n-1`. Damit sind z. B. `A`, `A+0` und `1*A` gleichbedeutend, ebenso `A+3` und `4*A`. Damit können, je nach persönlicher Vorliebe, Positionen entweder ab 0 oder ab 1 gezählt werden, je nachdem, ob man `A+n` oder `n*A` verwendet.

Vollkommen symmetrisch zu `A`, bezeichnet die Konstante `Z` mit Typ `posZ` das Ende einer Sequenz, d. h. die Position nach dem letzten Element. Entsprechend bezeichnen `p+n` (oder `n+p`) und `p-n` für eine Position `p` mit Typ `posZ` die Position, die sich `n` Elemente links bzw. rechts (!) von `p` (bzw. `-n` Elemente rechts bzw. links von `p`, falls `n` negativ ist) befindet. `n*Z` (oder `Z*n`) ist gleichbedeutend mit `Z+n-1`.

Die nachfolgende Abbildung zeigt die Bedeutung verschiedener Positionen für eine Sequenz mit `N` Elementen:




Für Positionen `p` und `q` desselben Typs liefert `q-p` die vorzeichenbehaftete Anzahl der Elemente, die sich zwischen diesen Positionen befinden, sodass `p+(q-p)` immer gleich `q` ist. Dementsprechend ist der Wert von `q-p` für Positionen des Typs `posA` bzw. `posZ`

- positiv, wenn sich `q` rechts bzw. links von `p` befindet;
- negativ, wenn sich `q` links bzw. rechts von `p` befindet;
- null, wenn `q` und `q` dieselbe Position bezeichnen.

Für Positionen unterschiedlicher Typen ist `q-p` jedoch nicht definiert, weil die entsprechende Anzahl der Elemente ohne Kenntnis der konkreten Sequenz nicht ermittelt werden kann.

Außerdem stehen für Positionen (ggf. desselben Typs) alle Operatoren zur Verfügung, die sich mit der „üblichen“ Bedeutung auf die oben genannten Operatoren zurückführen lassen, zum Beispiel:

- `p+=n` entspricht `p=p+n`.
- `++p` entspricht `p+=1`.
- `p<q` entspricht `p-q<0`.
- usw.

Eine nicht explizit initialisierte Variable des Typs `posA` bzw. `posZ` besitzt implizit den Wert `0*A` bzw. `0*Z`.

Eine Position `p` des Typs `posA` bzw. `posZ` kann implizit als `bool`-Wert verwendet werden, der genau `false` ist, wenn `p<A` bzw. `p<Z` ist, weil eine solche Position garantiert außerhalb jeder Sequenz (vor dem Anfang bzw. nach dem Ende) liegt.

## 6.4 Bereiche

Ein Bereich bezeichnet einen zusammenhängenden Teil einer beliebigen Sequenz, der entweder durch eine Anfangs- und eine Endposition (die unabhängig voneinander Typ `posA` oder `posZ` besitzen können) oder eine Anfangsposition und eine ganzzahlige Länge (Anzahl von Elementen) oder eine ganzzahlige Länge und eine Endposition definiert ist.

Prinzipiell sind beliebige Positionen und Längen erlaubt, das heißt:

- Jede Position kann auch vor dem Anfang oder hinter dem Ende der Sequenz liegen.
- Die Anfangsposition kann auch rechts von der Endposition liegen.
- Die Länge kann auch negativ sein.

Die Bedeutung dieser Sonderfälle wird weiter unten erklärt.

Für Positionen p und q und einen ganzzahligen Wert n bezeichnet:

- p|q den Bereich mit Anfangsposition p und Endposition q;
- p|n den Bereich mit Anfangsposition p und Länge n, was äquivalent zu p|p+n bzw. p|p-n ist, wenn p Typ posA bzw. posZ besitzt;
- n|q den Bereich mit Länge n und Endposition q was äquivalent zu q-n|q bzw. q+n|q ist, wenn q Typ posA bzw. posZ besitzt.

## 6.5 Bildung von Teilsequenzen

Für eine Sequenz s und einen Bereich p|q mit Anfangsposition p und Endposition q liefert s(p|q) eine neue Sequenz mit allen Elementen von s außer denen, die sich vor Position p oder nach Position q befinden.

Das sind „normalerweise“ die Elemente, die sich zwischen den Positionen p und q befinden. Die vorstehende Formulierung erfasst jedoch auch alle oben genannten Sonderfälle von Bereichen:

- Wenn die Anfangsposition p hinter dem Ende der Sequenz liegt, liegen alle Elemente vor dieser Position, d. h. die Teilsequenz ist dann leer.
- Ebenso, wenn die Endposition q vor dem Anfang der Sequenz liegt.
- Wenn die Anfangsposition p vor dem Anfang der Sequenz liegt, gibt es keine Elemente vor dieser Position, d. h. die Teilsequenz beginnt dann mit dem ersten Element (sofern sie nicht leer ist).
- Wenn die Endposition q nach dem Ende der Sequenz liegt, gibt es keine Elemente nach dieser Position, d. h. die Teilsequenz endet dann mit dem letzten Element (sofern sie nicht leer ist).
- Wenn die Anfangsposition p rechts von der Endposition q liegt, ist die Teilsequenz leer, weil sich dann jedes Element entweder vor Position p oder nach Position q befindet.

Für Bereiche der Art p|n und n|q ist s(p|n) bzw. s(n|q) jeweils äquivalent zu s(p|q), wenn p|q jeweils den zu p|n bzw. n|q äquivalenten Bereich gemäß der Definition weiter oben bezeichnet.

## 6.6 Ersetzung von Teilsequenzen

Für eine Sequenz s und einen Bereich p|q mit Anfangsposition p und Endposition q ist s(p|q, xs) äquivalent zu s(A|p) + seq<T>(xs) + s(q|Z), sofern dieser Ausdruck typkorrekt ist. Dementsprechend kann xs z. B. ein einzelner Wert mit Typ T oder eine Sequenz mit Typ seq<T> sein.

Das bedeutet „normalerweise“, dass in einer (gedachten) Kopie von s die Elemente zwischen den Positionen p und q durch die Elemente von xs ersetzt werden, womit auch das Einfügen und Entfernen von Elementen jeweils als Spezialfall enthalten ist. (Wenn Anfangs- und Endposition des Bereichs gleich sind, werden die Elemente von xs an dieser Position eingefügt. Wenn xs z. B. eine leere Sequenz ist, werden die Elemente des Bereichs entfernt.) Aber auch hier werden durch die vorstehende Formulierung wieder alle Sonderfälle von Bereichen berücksichtigt:

- Wenn die Anfangsposition p vor dem Anfang der Sequenz liegt, ist die Teilsequenz s(A|p) leer und fehlt somit in der Resultatsequenz.
- Wenn die Endposition q nach dem Ende der Sequenz liegt, ist die Teilsequenz s(q|Z) leer und fehlt somit in der Resultatsequenz.
- Wenn die Anfangsposition p rechts von der Endposition q liegt, überlappen sich die Teilsequenzen s(A|p) und s(q|Z) teilweise, was dazu führt, dass ihre gemeinsame Teilsequenz s(q|p) in der Resultatsequenz zweimal auftritt, einmal vor und einmal nach den Elementen von xs.

Obwohl dies etwas ungewöhnlich ist, ist diese Definition zumindest konsistent und „symmetrisch“. Da der Bereich zwischen den Positionen `p` und `q` leer ist (`s(p|q)` liefert in diesem Fall tatsächlich eine leere Sequenz), müssten die Elemente von `xs` eigentlich nur an der richtigen Stelle eingefügt werden. Aber die Frage nach der „richtigen“ Stelle lässt sich vermutlich nicht sinnvoll beantworten. Eine denkbare, aber noch etwas kompliziertere Alternative wäre, `s(p|q, xs)` grundsätzlich äquivalent zu `s(A|l) + seq<T>(xs) + s(r|Z)` zu definieren, wobei `l` bzw. `r` das „Minimum“ bzw. „Maximum“ der Positionen `p` und `q` bezeichnet.

- Wenn die Anfangsposition `p` hinter dem Ende der Sequenz oder die Endposition `q` vor dem Anfang der Sequenz liegt, enthält die Teilsequenz `s(A|p)` bzw. `s(q|Z)` alle Elemente von `s`.

Für Bereiche der Art `p|n` und `n|q` ist `s(p|n, xs)` bzw. `s(n|q, xs)` jeweils äquivalent zu `s(p|q, xs)`, wenn `p|q` jeweils den zu `p|n` bzw. `n|q` äquivalenten Bereich bezeichnet.

Für eine einzelne Position `p` ist `s(p, xs)` äquivalent zu `s(p|p, xs)`, d. h. die Elemente von `xs` werden an Position `p` eingefügt.

Anstelle eines einzelnen Arguments `xs` können bei allen o. g. Aufrufen auch mehrere Argumente angegeben werden, die dann jeweils einzeln mittels `seq<T>(xs)` in Sequenzen umgewandelt und dann miteinander verkettet werden.

## 6.7 Abfrage einzelner Elemente

Für eine Sequenz `s` und eine Position `p` mit Typ `posA` bzw. `posZ` liefert `s[p]` das Element von `s`, das sich rechts bzw. links von Position `p` befindet, als R-Wert. Falls es kein solches Element gibt, wird der Wert `nil` bzw. `nil_v<T>` geliefert, wobei die Variablenschablone `nil_v` wie folgt definiert ist:

```
template <typename T>
const T nil_v = T();
```

Durch Spezialisierungen dieser Schablone können bei Bedarf andere Ersatzwerte definiert werden.

## 6.8 Hilfstypen

Die Typen `boolA` und `boolZ` sind jeweils „Repliken“ von `bool`, d. h. sie besitzen jeweils implizite Umwandlungen von und nach `bool` (deren Hintereinanderausführung jeweils die identische Funktion ist) und können deshalb im wesentlichen wie `bool` verwendet werden. Ihre Unterscheidung ist jedoch im folgenden wichtig. (Deshalb ist es auch wichtig, dass es sich nicht nur um Synonyme oder Aliase von `bool` handelt, die mit `typedef` oder `using` definiert sind, sondern jeweils um eigenständige Typen.)

Die Typen `bool12A` und `bool12Z` sind ebenfalls derartige Repliken von `bool`, die zusätzlich noch einen zweiten optionalen `bool`-Wert enthalten können. Tatsächlich sind sie von `std::pair<bool, std::optional<bool>>` abgeleitet, sodass die in ihnen enthaltenen `bool`-Werte mit Hilfe der Elementvariablen `first` und `second` von `pair` und der verschiedenen Elementfunktionen von `optional` abgefragt werden können. Zusätzlich zu den impliziten Umwandlungen von und nach `bool`, die den Wert `first` setzen bzw. liefern und `second` auf `std::nullopt` setzen bzw. unbeachtet lassen, besitzen die Typen noch folgende Elementfunktionen:

- Einen Konstruktor mit zwei `bool`-Werten als Parameter, die in `first` bzw. `second` gespeichert werden.
- Einen Präfixoperator `*`, der den Wert `second` liefert.

Da `optional` ebenfalls einen Präfixoperator `*` sowie eine implizite Umwandlung nach `bool` besitzt,

können die in einem Objekt `x` des Typs `bool12A` oder `bool12Z` enthaltenen `bool`-Werte auch wie folgt abgefragt werden:

- Die implizite Verwendung von `x` als `bool`-Wert liefert den ersten `bool`-Wert `x.first`.
- Die implizite Verwendung von `*x` als `bool`-Wert liefert den Wert `x.second.has_value()`, der genau dann `true` ist, wenn `x` einen zweiten `bool`-Wert enthält.
- In diesem Fall liefert `**x` diesen zweiten `bool`-Wert `x.second.value()`. (Andernfalls ist `**x` undefiniert.)

Obwohl auch diese beiden Typen „funktional identisch“ sind, ist ihre Unterscheidung im folgenden wichtig.

## 6.9 Suchen, Filtern und Transformieren von Elementen

Für eine Sequenz `s` und eine Funktion `f` (bei der er sich auch um ein Funktionsobjekt oder einen Lambda-Ausdruck handeln kann), die mit einem Wert des Typs `T` aufgerufen werden kann und ein Resultat eines beliebigen Typs `R` zurückliefert, liefert `s(f)` entweder einen Wert des Typs `posA` oder `posZ` oder eine neue Sequenz, deren Elemente wie folgt aus den Elementen `x` von `s` gebildet werden:

- Wenn `R` gleich `boolA` oder `boolZ` ist, hat der Resultatwert von `s(f)` den Typ `posA` bzw. `posZ` und bezeichnet „normalerweise“ die Position `p` des ersten Elements von vorn bzw. von hinten, für das `f(s[p])` gleich `true` ist. Falls es kein solches Element gibt, ist der Resultatwert `0*A` bzw. `0*Z`. (Das heißt, der Resultatwert ist genau implizit `true`, wenn es ein entsprechendes Element gibt.)
  Das heißt, die Funktion `f` wird zum Suchen des ersten bzw. letzten Elements mit einer bestimmten Eigenschaft verwendet.
- Wenn `R` gleich `bool` ist, besitzt die Resultatsequenz den Typ `seq<T>` und enthält diejenigen Elemente `x`, für die `f(x)` gleich `true` ist.
  Das heißt, die Funktion `f` wird zum Filtern der Elemente `x` verwendet.
  Dieses Verhalten kann prinzipiell genauso leicht auch mit dem nachfolgenden Fall erreicht werden. Die Verwendung von `bool` anstelle von `bool12A` oder `bool12Z` als Resultattyp verdeutlicht jedoch zum einen, dass die Reihenfolge, in der die Elemente durchlaufen werden, hier prinzipiell unwichtig ist; zum anderen lässt sich dieser Fall etwas effizienter implementieren, weil die Resultatwerte der Funktionsaufrufe `f(x)` nur auf `true` oder `false` getestet und nicht genauer untersucht werden müssen.
- Wenn `R` gleich `bool12A` oder `bool12Z` ist, besitzt die Resultatsequenz ebenfalls den Typ `seq<T>` und enthält zunächst diejenigen Elemente `x`, für die `f(x).first` gleich `true` ist. Sobald `f(x).second.has_value()` bei einem Aufruf von `f` (unabhängig vom Wert `f(x).first`) gleich `true` ist, wird die Funktion für die verbleibenden Elemente `x` nicht mehr aufgerufen; wenn `f(x).second.value()` dann ebenfalls `true` ist, werden diese verbleibenden Elemente noch zur Resultatsequenz hinzugefügt, andernfalls nicht.
  Wenn `R` gleich `bool12A` ist, werden die Elemente `x` hierfür in ihrer „natürlichen“ Reihenfolge von vorn nach hinten durchlaufen. Wenn `R` gleich `bool12Z` ist, werden sie jedoch in umgekehrter Reihenfolge durchlaufen und die Resultatsequenz entsprechend von hinten nach vorn aufgebaut. Dieser Unterschied im Resultattyp der Funktion ist genau dann wichtig, wenn der Fall `f(x).second.has_value()` eintritt. (Wenn bekannt ist, dass er nie eintreten wird, kann statt `bool12A` oder `bool12Z` einfach Resultattyp `bool` verwendet werden, wie weiter oben bereits erläutert wurde.)
  Das heißt, die Funktion `f` wird hier ebenfalls zum Filtern einzelner Elemente verwendet und bietet gleichzeitig die Möglichkeit, alle verbleibenden Elemente ab einer bestimmten Stelle auf einmal ein- oder auszuschließen. Damit lassen sich insbesondere Funktionen zum bequemen Löschen des ersten (Resultattyp `bool12A`) oder letzten (Resultattyp `bool12Z`) Elements mit einer bestimmten Eigenschaft schreiben (siehe unten).

- Wenn `R` gleich `std::optional<U>` für einen beliebigen Typ `U` ist, besitzt die Resultatsequenz den Typ `seq<U>` und enthält alle Elemente `f(x).value()`, für die `f(x).has_value()` gleich `true` ist.
  Das heißt, die Funktion `f` wird hier sowohl zum Filtern als auch zum Transformieren von Elementen verwendet.
- Für alle anderen Typen `R` besitzt die Resultatsequenz den Typ `seq<R>` und enthält alle Elemente `f(x)`.
  Das heißt, die Funktion `f` wird hier nur zum Transformieren von Elementen verwendet.
  Wenn eine solche Transformationsfunktion als Resultattyp `R` eigentlich einen der zuvor genannten besonderen Typen (z. B. `bool` oder `optional<U>`) besitzen soll, kann man stattdessen Resultattyp `optional<R>` (also z. B. `optional<bool>` oder `optional<optional<U>>`) verwenden und immer ein `optional`-Objekt zurückliefern, das tatsächlich einen Wert enthält.

Neben `s(f)` ist auch die allgemeinere Form `s(f, args)` mit beliebigen weiteren Argumenten `args` möglich, die dann jeweils zusätzlich zum aktuellen Element `x` an die Funktion `f` weitergegeben werden.

## 6.10 Funktionen zum Suchen und Löschen bestimmter Elemente

Für eine Funktion `f` mit Resultattyp `bool` (z. B. `eq(5)` oder `(rem(2), eq(0))`) liefert:

- `s(searchA(f, n))` bzw. `s(searchZ(f, n))` die Position mit Typ `posA` bzw. `posZ` des `n`-ten Elements `x` von `s` von vorn bzw. von hinten, für das `f(x)` gleich `true` ist. Wenn es kein solches Element gibt, ist der Resultatwert `0*A` bzw. `0*Z`.
  Wenn das Argument `n` fehlt, wird `n=1` verwendet.
- `s(remove(f))` eine Kopie von `s`, in der alle Elemente `x` fehlen, für die `f(x)` gleich `true` ist.
- `s(removeA(f, n))` bzw. `s(removeZ(f, n))` eine Kopie von `s`, in der die ersten bzw. letzten `n` Elemente `x` fehlen, für die `f(x)` gleich `true` ist.
  Wenn das Argument `n` fehlt, wird `n=1` verwendet.

## 6.11 STL-Iteratoren

Für eine Sequenz `s` und eine Position `p` mit Typ `posA` bzw. `posZ` liefert `s(p)` einen konstanten STL-Iterator der Kategorie Random Access, der logisch auf die Position `p` von `s` zeigt (die sich zwischen zwei benachbarten Elementen befindet). Aus STL-Sicht zeigt er jedoch immer auf das Element `s[p]`, egal ob es dieses Element tatsächlich gibt oder nicht, sodass `*s(p)` immer äquivalent zu `s[p]` ist. Des weiteren ist auch `*(s(p)+n)` für jeden ganzzahligen Wert `n` äquivalent zu `s[p+n]`. Daraus folgt indirekt, dass die von `s(p)` gelieferten Iteratoren für `p` vom Typ `posA` bzw. `posZ` vorwärts bzw. rückwärts (reverse) durch die Sequenz `s` iterieren.

Für einen derartigen Iterator `p` liefert `~p` einen korrespondierenden Reverse-Iterator, der logisch auf dieselbe Position von `s` wie `p` zeigt. Aus STL-Sicht zeigt `~p` jedoch auf das Element auf der jeweils anderen Seite dieser Position.

Insbesondere gilt (logisch, da es die entsprechenden Elementfunktionen nicht gibt):

- `s(A)` entspricht `s.cbegin()`.
- `s(Z)` entspricht `s.crbegin()`.
- `~s(Z)` entspricht `s.cend()`.
- `~s(A)` entspricht `s.crend()`.

Darüber hinaus gibt es die für die Anweisung `for (x : s) .....` benötigten Elementfunktionen `begin` und `end` (die ebenfalls konstante Iteratoren liefern).

### 6.12 Vergleichsoperatoren

Für zwei Sequenzen `s1` und `s2` desselben Typs `seq<T>` liefert `diff(s1, s2)` das Ergebnis des „Dreiwegvergleichs“ von `s1` und `s2`, d. h. einen ganzzahligen Wert kleiner bzw. gleich bzw. größer als 0, wenn `s1` lexikographisch kleiner bzw. gleich bzw. größer als `s2` ist.

Zwei Werte `x1` und `x2` des Typs `T` werden dabei wiederum mittels `diff(x1, x2)` verglichen. Hierfür ist `diff` als Funktionsschablone für beliebige Typen `T` vordefiniert und liefert jeweils die Differenz `x1-x2`. Durch Spezialisierungen dieser Schablone kann dieser „Dreiwegvergleich“ für bestimmte Typen `T` aber auch anders definiert werden.

Für zwei Sequenzen `s1` und `s2` desselben Typs `seq<T>` und einen Vergleichsoperator `op` aus der Menge <, >, <=, >=, ==, != ist `s1 op s2` gleichbedeutend mit `diff(s1, s2) op 0`.

### 6.13 Zeichenketten

`str` ist ein Synonym bzw. Alias für `seq<char>`.

Dieser Typ besitzt einen zusätzlichen impliziten Konstruktor mit Parametertyp `const char*`, sodass „C-Strings“ (insbesondere String-Literale) bei Bedarf implizit in `str` umgewandelt werden und deshalb (vermutlich) überall verwendet werden können, wo ein Wert des Typs `str` erwartet wird, z. B. bei Verkettungsoperationen oder Vergleichen.

Außerdem ist `std::hash<str>` sowie der übliche Ausgabeoperator << für `str` definiert.

### 6.14 Sonstiges

Für eine Sequenz `s` liefert `~s` eine neue Sequenz, die die Elemente von `s` in umgekehrter Reihenfolge enthält.

`is_seq` und `is_seq_v` sind analog zu `std::is_pointer` und `std::is_pointer_v` und ähnlichen Schablonen definiert und können verwendet werden, um zur Übersetzungszeit zu überprüfen, ob ein Typ ein Sequenztyp ist.

`elem` und `elem_t` sind analog zu `remove_all_extents` und `remove_all_extents_t` definiert und können verwendet werden, um den direkten oder indirekten Elementtyp eines Sequenztyps zu ermitteln (der dann kein Sequenztyp mehr ist).

Der direkte Elementtyp eines Sequenztyps `S` kann mittels `S::elem_t` ermittelt werden.

## 7 Tabellen (`tab.ch`)

Für Typen `K` und `V` stellt ein Objekt `t` des Typs `tab<K, V>` eine Tabelle mit Schlüsseln des Typs `K` und zugehörigen Werten des Typs `V` dar, vergleichbar mit `map<K, V>` und `unordered_map<K, V>`, aber mit einer Syntax und Semantik ähnlich zu offenen Typen (vgl. §8):

- Der parameterlose Konstruktor, der auch implizit aufgerufen werden kann, initialisiert das Objekt als nil-Objekt.

- Bei expliziter oder impliziter Initialisierung mit `uniq` erhält man eine leere Tabelle.
- Beim Kopieren einer Tabelle entsteht keine neue Tabelle, sondern lediglich ein zusätzlicher Zeiger auf dieselbe Tabelle.
- Ein Objekt des Typs `tab<K, V>` kann implizit als `bool`-Wert verwendet werden, der genau dann `true` ist, wenn das Objekt nicht nil ist (auch wenn die Tabelle leer ist).
- `*t` liefert die Größe der Tabelle `t`, d. h. die Anzahl ihrer Einträge. Wenn `t` nil ist, erhält man den Wert 0.
- Für einen Schlüssel `k` des Typs `K` und einen Wert `v` des Typs `V` speichert `t(k, v)` das Schlüssel-Wert-Paar `(k, v)` in der Tabelle `t`. Wenn `t` bereits einen Eintrag mit Schlüssel `k` enthält, wird dieser überschrieben. Wenn `t` nil ist, ist die Operation jedoch wirkungslos.
- Für einen Schlüssel `k` des Typs `K` liefert `t(k)` den Wert, der zum Schlüssel `k` in der Tabelle `t` gespeichert ist. Wenn `t` keinen Eintrag mit Schlüssel `k` enthält oder wenn `t` nil ist, erhält man nil.
- Für einen Schlüssel `k` des Typs `K` liefert `t[k]` genau dann `true`, wenn `t` einen Eintrag mit Schlüssel `k` enthält.
- `+t` liefert ein Duplikat der Tabelle `t`, d. h. eine neue, von `t` unabhängige Tabelle mit den gleichen Einträgen. Für `t` gleich nil erhält man wiederum nil.
- `t.begin()` und `t.end()` liefern konstante Iteratoren mit der üblichen Bedeutung, sodass die Einträge von `t` mit Anweisungen wie `for (auto [k, v] : t) ......` durchlaufen werden können.

Implementierungstechnisch ist ein Objekt des Typs `tab<K, V>` einfach ein `shared_ptr<unordered_map<K, V>>`. Dementsprechend muss es für den Typ `K` eine Spezialisierung von `std::hash` sowie einen Gleichheitsoperator geben.

# 8 Offene Typen (`type.ch`)

## 8.1 Typen

`TYPE(X)` definiert einen offenen Typ mit dem Namen `X` mit folgenden Konstruktoren und weiteren Operationen:

- `X()` liefert ein nil-Objekt, d. h. logisch kein Objekt.
  Dieser Konstruktor kann auch implizit aufgerufen werden.
- `X(uniq)` liefert ein neues leeres Objekt mit einer eindeutigen Identität. (`uniq` ist eine globale Konstante.)
  Dieser Konstruktor kann auch implizit aufgerufen werden.
- Für ein Attribut `a` von `X` (siehe unten) und prinzipiell beliebige Werte `y` ist `X(a, y ...)` äquivalent zu `X(uniq)(a, y ...)`.
  Die Bedeutung des hier verwendeten Klammeroperators wird später erklärt.
- Für ein Objekt `x` des Typs `X` liefert der Kopierkonstruktor `X(x)` ein Objekt mit derselben Identität wie `x`, d. h. logisch dasselbe Objekt.
  Dieser Konstruktor kann auch implizit aufgerufen werden.
- Für Objekte `x1` und `x2` des Typs `X` überschreibt die Zuweisung `x1=x2` die Identität von `x1` durch die Identität von `x2`, d. h. anschließend enthält die Variable `x1` logisch dasselbe Objekt wie `x2`.
- Ein Objekt `x` des Typs `X` kann implizit als `bool`-Wert verwendet werden, der genau dann `true` ist, wenn `x` ein „echtes“ Objekt ist, das direkt oder indirekt mittels `X(uniq)` initialisiert wurde.

- Für Objekte `x1` und `x2` des Typs `X` kann mittels `x1==x2` und `x1!=x2` überprüft werden, ob sie dieselbe Identität besitzen oder nicht.
  Durch eine eigene Definition von `operator==` für den Typ `X` (als normale Funktion oder Funktionsschablone) kann ein beliebiges anderes Vergleichskriterium definiert werden. Der vordefinierte `operator!=` ruft lediglich den entsprechenden `operator==` auf, sodass er nicht separat umdefiniert werden muss.
- Für ein „echtes“ Objekt `x` eines offenen Typs `X` liefert `+x` ein Duplikat des Objekts, d. h. ein neues Objekt des Typs `X` mit denselben Attributwerten wie `x`. (Das heißt, sämtliche Attributwerte von `x` werden mit ihren jeweiligen Kopierkonstruktoren in das neue Objekt kopiert.) Für ein nil-Objekt `x` liefert `+x` ebenfalls ein nil-Objekt.

## 8.2 Attribute

Für einen Namen `a`, einen offenen Typ `X` und einen (nahezu) beliebigen Typ `Y` definiert `ATTR1(a, X, Y)` bzw. `ATTRN(a, X, Y)` ein einwertiges bzw. mehrwertiges Attribut `a` des Typs `X` mit Zieltyp `Y`.

Unterschiedliche Typen können Attribute mit demselben Namen und gleichen oder unterschiedlichen Zieltypen besitzen.

## 8.3 Einwertige Attribute

Für einen offenen Typ `X`, ein Objekt `x` dieses Typs und ein einwertiges Attribut `a` dieses Typs mit Zieltyp `Y` liefert `x(a)` den aktuellen Wert des Attributs `a` des Objekts `x` als R-Wert. Falls das Attribut für dieses Objekt noch keinen Wert besitzt, erhält man den Ersatzwert `nil`.

Für einen Wert `y` mit Typ `Y` setzt `x(a, y)` den Wert des Attributs `a` des Objekts `x` auf `y` und liefert das Objekt `x` zurück. Damit können beliebig viele derartige Operationen bequem nacheinander ausgeführt werden:

```
x(a1, y1)(a2, y2).....
```

Durch Kombination mit dem Konstruktor des Typs `X` können außerdem bequem Objekte mit beliebig vielen initialen Attributwerten erzeugt werden:

```
X(a1, y1)(a2, y2).....
```

Wenn `x` ein nil-Objekt ist, ist `x(a, y)` wirkungslos.

## 8.4 Mehrwertige Attribute

Ein mehrwertiges Attribut `a` mit Zieltyp `Y` entspricht zunächst einem einwertigen Attribut mit Zieltyp `seq<Y>`, d. h. `x(a)` liefert die aktuellen Werte des Attributs `a` des Objekts `x` als (ggf. leere) Sequenz mit Typ `seq<Y>` und `x(a, ys)` ersetzt diese Werte durch die Elemente der Sequenz `ys`.

Neben dieser grundlegenden „Schreibfunktion“ zum Ersetzen aller Werte auf einmal, gibt es jedoch weitere Schreibfunktionen `x(a, y ...)` mit folgender Bedeutung:

- Wenn der erste Parameter `y` (nach dem Attribut `a`) ein einzelner Wert mit Typ `Y` oder eine Sequenz mit Typ `seq<Y>` ist, werden die aktuellen Werte des Attributs `a` des Objekts `x` durch die Werte aller Parameter `y` ersetzt, sofern deren Verkettung zu einer einzigen Sequenz typkorrekt ist.
  Dies schließt die zuvor beschriebene grundlegende Schreibfunktion als Spezialfall mit ein.
- Andernfalls (d. h. wenn der erste Parameter `y` weder in `Y` noch in `seq<Y>` umgewandelt werden kann) werden alle aktuellen Werte des Attributs durch die Werte der Sequenz `x(a)(y ...)` ersetzt,

d. h. alle Parameter `y` werden an den Klammeroperator der aktuellen Sequenz `x(a)` übergeben, der je nach Art der Parameter z. B. eine Teilsequenz davon liefert oder eine Kopie der Sequenz, in der bestimmte Elemente durch andere ersetzt sind, usw. (Vgl. die Beschreibung von Sequenzen.)

Wenn `x` ein nil-Objekt ist, ist `x(a, y ...)` wiederum wirkungslos.

## 8.5 Tabellenwertige Attribute

Für einen Namen `a`, einen offenen Typ `X` und (nahezu) beliebige Typen `K` und `V` definiert `ATTRT(a, X, K, V)` ein tabellenwertiges Attribut `a` des Typs `X` mit Schlüsseltyp `K` und Werttyp `V`.

Ein solches Attribut entspricht zunächst einem einwertigen Attribut mit Zieltyp `tab<K, V>`, d. h. `x(a)` liefert den aktuellen Wert des Attributs `a` des Objekts `x` als Objekt mit Typ `tab<K, V>` und `x(a, t)` ersetzt diesen Wert durch die Tabelle `t`.

Neben diesen grundlegenden Funktionen zum Abfragen und Ersetzen der gesamten Tabelle, gibt es folgende weitere Funktionen zum direkten Zugriff auf einzelne in der Tabelle gespeicherte Schlüssel-Wert-Paare:

- Für einen Wert `k` des Typs `K` ist `x(a, k)` gleichbedeutend mit `x(a)(k)`, d. h. es liefert den Wert, der in der Tabelle `x(a)` für den Schlüssel `k` gespeichert ist.
  Wenn `x` ein nil-Objekt ist oder noch keinen Wert für das Attribut `a` besitzt oder dieser Attributwert nil ist oder die Tabelle `x(a)` keinen Wert für den Schlüssel `k` enthält, erhält man jeweils nil.
- Für einen Wert `v` des Typs `V` speichert `x(a, k, v)` das Schlüssel-Wert-Paar `(k, v)` in der Tabelle `x(a)` und liefert das Objekt `x` zurück. Wenn `x` noch keinen Wert für das Attribut `a` besitzt, wird ihm zuvor eine neue, leere Tabelle zugewiesen, sofern `x` kein nil-Objekt ist. Das heißt, `x(a, k, v)` führt ggf. `x(a, uniq)` und anschließend `x(a)(k, v)` aus.
  Wenn `x` ein nil-Objekt ist oder `x(a)` explizit auf nil gesetzt wurde, ist die Operation wirkungslos.

## 8.6 Attributreferenzen

Für ein Objekt `x` eines offenen Typs und ein Attribut `a` dieses Typs liefert `x[a]` eine logische Referenz mit Typ `aref<Y>` (wenn `a` ein einwertiges Attribut mit Zieltyp `Y` ist) bzw. `aref<seq<Y>>` (wenn `a` ein mehrwertiges Attribut mit Zieltyp `Y` ist) bzw. `aref<tab<K, V>>` (wenn `a` ein tabellenwertiges Attribut mit Schlüsseltyp `K` und Werttyp `V` ist) auf den Wert des Attributs `a` des Objekts `x`.

Eine solche Referenz `x[a]` kann implizit als `bool`-Wert verwendet werden, der genau dann `true` ist, wenn das Objekt `x` einen Wert für das Attribut `a` besitzt, der jedoch auch nil sein kann. Der `bool`-Wert bezieht sich auf den Zeitpunkt der Verwendung der Referenz, nicht auf den Zeitpunkt, als die Referenz gebildet wurde.

Die Anwendung des Operators `~` auf eine solche Referenz `x[a]` entfernt den Wert des Attributs `a` des Objekts `x`, sodass `x[a]` anschließend `false` ist. Demgegenüber setzt `x(a, nil)` den Wert des Attributs auf nil, sodass `x[a]` anschließend `true` ist, obwohl `x(a)` in beiden Fällen nil liefert.

Außerdem kann eine solche Referenz wie folgt zum Lesen und Verändern des Attributwerts verwendet werden:

- `x[a]()` ist äquivalent zu `x(a)`.
- `x[a](y)` ist äquivalent zu `x(a, y)`.
- Wenn die möglichen Werte des Attributs `a` Sequenzen sind, d. h. wenn das Attribut entweder mehrwertig oder aber einwertig mit einem Sequenztyp als Zieltyp ist, ist `x[a](y ...)` äquivalent zu

`x(a, y ...)` bei mehrwertigen Attributen. Wenn das Attribut einwertig mit Zieltyp `seq<Y>` ist, wird es hierfür wie ein mehrwertiges Attribut mit Zieltyp `Y` behandelt.

- Wenn die möglichen Werte des Attributs `a` Tabellen sind, d. h. wenn das Attribut entweder tabellenwertig oder aber einwertig mit einem Tabellentyp als Zieltyp ist, ist `x[a](y ...)` äquivalent zu `x(a, y ...)` bei tabellenwertigen Attributen. Wenn das Attribut einwertig mit Zieltyp `tab<K, V>` ist, wird es hierfür wie ein tabellenwertiges Attribut mit Schlüsseltyp `K` und Werttyp `V` behandelt.

## 8.7 Typ- und Attributschablonen

`TYPETEMP(X, tpar ...)` definiert eine Schablone (template) von offenen Typen mit dem Namen `X` und einem oder mehreren Schablonenparametern `tpar`, zum Beispiel (die Namen der Schablonenparameter können auch weggelassen werden):

```
TYPETEMP(List, typename T)
TYPETEMP(Pair, typename A, typename B)
```

`ATTR1TEMP(a, X, Y, tpar ...)` bzw. `ATTRNTEMP(a, X, Y, tpar ...)` bzw. `ATTRTTEMP(a, X, K, V, tpar ...)` definiert eine Schablone von ein- bzw. mehr- bzw. tabellenwertigen Attributen mit dem Namen `a` und Schablonenparametern `tpar` für die Typen `X` mit Zieltyp(en) `Y` bzw. mit Schlüsseltyp(en) `K` und Werttyp(en) `V`. `X` muss ein Schablonenbezeichner (template id) sein, aus dem alle Schablonenparameter deduziert werden können, z. B. `List<T>` oder `Pair<A, B>`. `Y`, `K` und `V` können normale Typbezeichner oder ebenfalls Schablonenbezeichner sein, die beliebig von einem oder mehreren Schablonenparametern abhängen können (d. h. die Parameter müssen aus `Y` bzw. `K` und `V` nicht deduziert werden können).

Wenn `X`, `Y`, `K` oder `V` Kommas enthalten, müssen sie mit `LIT` (definiert in `util.ch`) geklammert werden, damit sie jeweils als ein einziges Makroargument übergeben werden, zum Beispiel:

```
ATTR1TEMP(head, List<T>, T, typename T)
ATTR1TEMP(first, LIT(Pair<A, B>), A, typename A, typename B)
```

## 8.8 Definitionen in Namensräumen und anderen Dateien

Wenn Typen und Attribute in mehreren Quelldateien benötigt werden, können sie – analog zu Strukturen bzw. Klassen sowie inline-Funktionen und -Variablen – in Definitionsdateien stehen, die dann in mehrere Quelldateien eingebunden werden können.

Typen und Attribute können prinzipiell unabhängig voneinander in unterschiedlichen Namensräumen definiert werden. Insbesondere kann in einem Namensraum ein Attribut für einen Typ definiert werden, der in einem anderen Namensraum definiert wurde. Dieses Attribut kann sogar den gleichen Namen (und den gleichen oder einen anderen Zieltyp) besitzen wie ein anderes Attribut des Typs, das in einem anderen Namensraum definiert wurde.

Die Namen der Typen und Attribute müssen hierfür bei Bedarf nach den normalen Regeln von C++ qualifiziert oder mittels `using` sichtbar gemacht werden.

## 8.9 Virtuelle Attribute

Virtuelle Attribute können wie gewöhnliche Attribute verwendet werden, obwohl ihre Werte nicht direkt in Objekten offener Typen gespeichert werden, sondern von benutzerdefinierten Funktionen nach Belieben gelesen und gespeichert werden können.

Ein virtuelles Attribut mit dem Namen `a` muss zunächst (unabhängig von einem konkreten Typ) mittels `ATTR(a)` definiert werden, sofern nicht bereits ein Attribut mit dem gleichen Namen mittels `ATTR1`, `ATTRN`, `ATTRT`, `ATTR1TEMP`, `ATTRNTEMP` oder `ATTRTTEMP` (im selben Namensraum in derselben Datei) definiert wurde. (Wenn bereits ein gleichnamiges Attribut definiert wurde, ist die Definition `ATTR(a)` nicht nötig, aber auch nicht störend.)

Anschließend können beliebig viele zugehörige Funktionen definiert werden, die alle den „virtuellen“ Namen `ATTRV` besitzen (in Wirklichkeit ist `ATTRV` bzw. `CH_ATTRV` ein Makro) und deren Parameterliste mit dem Attributnamen `a` beginnt (obwohl `a` in Wirklichkeit kein Typname ist).

Der zweite Parameter muss ein Objekt `x` des offenen Typs `X` sein, zu dem das Attribut gehören soll. Eventuelle weitere Parameter `y` ergeben sich aus den gewünschten Aufrufmöglichkeiten, weil jeder Aufruf der Art `x(a, y ...)` letztlich auf einen Aufruf `ATTRV(aa, x, y ...)` einer so definierten Funktion abgebildet wird. Dabei steht `aa` für irgendeinen geeigneten Wert des ersten Parameters.

Zum Beispiel:

```
// Offener Typ Person mit mehrwertigem Attribut firstnames.
TYPE(Person)
ATTRN(firstnames, Person, str)

// Virtuelles Attribut firstname für prinzipiell beliebige offene Typen
// mit prinzipiell beliebigen Zieltypen.
ATTR(firstname)

// Einwertiges virtuelles Attribut firstname für Person mit Zieltyp
// str durch eine Lese- und eine Schreibfunktion definieren.
// Die Lesefunktion liefert den ersten Vornamen der Person p, falls
// vorhanden, andernfalls nil.
// Die Schreibfunktion ersetzt den ersten Vornamen der Person p durch s,
// falls es bereits einen gibt, andernfalls wird s als erster Vorname
// hinzugefügt.
// Die Schreibfunktion sollte das Objekt p zurückliefern, damit
// verkettete Verwendungen des Klammeroperators wie gewohnt
// funktionieren.
str ATTRV (firstname, Person p) {
    return p(firstnames)[A];
}
Person ATTRV (firstname, Person p, str s) {
    return p(firstnames, A|1, s);
}

// Verwendung von firstname wie ein anderes einwertiges Attribut,
// zum Beispiel:
Person p1(firstname, "Hans");
str s1 = p1(firstname);
```

## 8.10 Streuwertfunktionen für offene Typen

Für jeden offenen Typ `X` ist der Typ `hash<X>` analog zu `std::hash` definiert, d. h. er besitzt einen konstanten Klammeroperator mit Parametertyp `X` und Resultattyp `size_t`, der für ein Objekt des Typs `X` seinen Streuwert anhand seiner Identität berechnet und zurückliefert.

Damit kann `std::hash<X>` einfach wie folgt definiert werden, was auch durch den Makroaufruf `HASH(X)` abgekürzt werden kann:

```
template <>
struct std::hash<X> : CH::hash<X> {};
```

Weil diese Definition aber nur im globalen Namensraum erlaubt ist, kann sie im allgemeinen nicht durch `TYPE(X)` bereitgestellt werden. Außerdem besteht so die Möglichkeit, `std::hash<X>` für einen Typ bei Bedarf auch anders zu definieren.

Für einen offenen Typ `X` erstellt `HASH(X)` genau die zuvor genannte Definition von `std::hash<X>`. Für einen Schablonenbezeichner `X` erstellt `HASHTEMP(X, tpar ...)` eine entsprechende Schablonendefinition mit den Schablonenparametern `tpar`, zum Beispiel `HASHTEMP(List<T>, typename T)`.

## 8.11 Automatische Speicherbereinigung

Der Speicherplatz, den Objekte offener Typen und ihre Attributwerte belegen, wird automatisch freigegeben, wenn die Objekte nicht mehr erreichbar sind.

Ein Objekt eines offenen Typs ist direkt erreichbar, wenn es sich in einem globalen, lokalen oder dynamischen Objekt befindet (das natürlich auch Teil eines größeren Objekts, d. h. eines Struktur- oder Feldobjekts sein kann).

Ein Objekt ist indirekt erreichbar, wenn es ausgehend von einem Attributwert eines anderen erreichbaren Objekts von der Funktion `follow` wie folgt gefunden wird. Diese Funktion erhält als Parameter einen solchen Attributwert per Referenz sowie eine Funktion und muss diese Funktion für jedes Objekt eines offenen Typs (wiederum per Referenz) aufrufen, das sich logisch innerhalb dieses Attributwerts befindet.

`follow` ist hierfür wie folgt durch Funktionsschablonen vordefiniert:

- Wenn der Typ des übergebenen Attributwerts ein offener Typ ist, wird die übergebene Funktion direkt mit diesem Attributwert aufgerufen.
- Wenn der Typ des übergebenen Attributwerts ein ein- oder mehrfacher Sequenztyp ist (was insbesondere für mehrwertige Attribute der Fall ist), dessen direkter oder indirekter Elementtyp ein offener Typ ist, wird `follow` mit derselben Funktion rekursiv für jedes Element der Sequenz aufgerufen.

Für alle anderen Typen ist die Funktion als leere Funktion vordefiniert, kann aber bei Bedarf geeignet spezialisiert werden, zum Beispiel:

```
template <typename T, typename F>
void follow (const std::vector<T>& v, const F& f) {
    for (T& x : v) follow(x, f);
}
```

Die automatische Speicherbereinigung wird immer dann ausgeführt, wenn die Gesamtzahl der existierenden verschiedenen Objekte offener Typen einen bestimmten Wert erreicht, der zunächst relativ willkürlich vordefiniert ist. Anschließend wird dieser Wert, der sozusagen die Größe des „Heaps“ für Objekte offener Typen darstellt, auf die Anzahl der „überlebenden“ Objekte geteilt durch den Belegungsfaktor `CH_GC_OCC` gesetzt, bei dem es sich um einen `double`-Wert zwischen 0 und 1 (jeweils ausschließlich) handeln muss. Das bedeutet, dass die Belegung des Heaps zu diesem Zeitpunkt gerade diesem Faktor entspricht und dass die Größe des Heaps nach jeder Speicherbereinigung entsprechend angepasst wird.

CH_GC_OCC kann vom Benutzer als Makro definiert werden. Wenn es nicht definiert ist, wird es mit dem Wert 0.4 definiert.

Durch Definition des Makros CH_GC_OFF kann die automatische Speicherbereinigung deaktiviert werden. Das kann zum Beispiel sinnvoll sein,

- um die Laufzeit eines Programms mit und ohne automatische Speicherbereinigung zu vergleichen;
- um ein Programm zu beschleunigen, das die automatische Speicherbereinigung nicht benötigt, weil es nicht sehr viele Objekte erzeugt;
- um bei einem fehlerhaften Programm Fehler in der automatischen Speicherbereinigung (hoffentlich) auszuschließen.

## 8.12 Sonstiges

Für einen Attributnamen a liefert a() ein generisches Funktionsobjekt, sodass jeder Aufruf der Art a()(x, ...) äquivalent zum Aufruf x(a, ...) ist.

Derartige Funktionsobjekte können u. a. zum Transformieren von Sequenzelementen verwendet werden (vgl. §6.9). Wenn ps beispielsweise eine Sequenz von Personen p ist und der Typ Person einwertige (gewöhnliche oder auch virtuelle) Attribute name und spouse besitzt, liefert ps(spouse()) eine Sequenz mit den Ehepartnern p(spouse) und ps(spouse())(name()) eine Sequenz mit den Namen p(spouse)(name) der Ehepartner der Personen p. (Falls eine Person keinen Ehepartner besitzt, ist der entsprechende Name nil, also eine leere Zeichenkette.)

is_open und is_open_v sind analog zu std::is_pointer und std::is_pointer_v und ähnlichen Schablonen definiert und können verwendet werden, um zur Übersetzungszeit zu überprüfen, ob ein Typ ein offener Typ ist.

## 8.13 Anmerkungen

Offene Typen haben gegenüber gewöhnlichen Strukturen oder Klassen einige Vorteile:

Zum einen können Attribute auch noch „später“ oder an einer anderen Stelle des Programms zu einem Typ hinzugefügt werden. So kann jedes „Modul“ bei Bedarf weitere Attribute definieren, von denen die übrigen Module u. U. gar nichts wissen müssen.

Zum anderen muss ein bestimmtes Objekt eines offenen Typs nicht immer Werte für sämtliche Attribute des Typs besitzen. Nicht vorhandene Attributwerte belegen dann auch keinen Speicherplatz. Beim Zugriff auf den Wert eines nicht vorhandenen Attributs erhält man einen wohldefinierten nil-Wert.

Damit decken offene Typen auch bequem variante Strukturen ab, die man ansonsten relativ aufwendig entweder mit union oder mit std::variant oder mit „künstlichen“ Klassenhierarchien (z. B. eine abstrakte Basisklasse Expr für arithmetische Ausdrücke mit konkreten Unterklassen wie Const für konstante Ausdrücke und Add, Sub etc. für Addition, Subtraktion etc.) modellieren müsste. Anders als bei Lösungen mit union, braucht man normalerweise auch kein explizites „tag“, um die konkret vorliegende Variante zu erkennen, sondern kann einfach die charakteristischen Attribute der einzelnen Varianten abfragen und damit indirekt die vorliegende Variante erkennen.

# 9 Garantierte Initialisierungsreihenfolge

Die im folgenden beschriebenen Makros können verwendet werden, um beim Starten bzw. Beenden eines C++-Programms Initialisierungs- bzw. Terminierungscode jeweils in einer wohldefinierten und sinnvollen Reihenfolge auszuführen.

`BEGIN(name)` bzw. `END(name)` auf globaler Ebene definiert den Anfang bzw. das Ende einer Initialisierungseinheit mit dem Namen `name`.

Dazwischen können mit `INIT { ..... }` bzw. `EXIT { ..... }` beliebig viele Initialisierungs- bzw. Terminierungsblöcke definiert werden, die bei der Initialisierung bzw. Terminierung dieser Einheit in textueller bzw. umgekehrter textueller Reihenfolge ausgeführt werden.

Mit `EMBED(other)` kann ausgedrückt werden, dass die Initialisierung bzw. Terminierung der Einheit `other` bei der Initialisierung bzw. Terminierung der umschließenden Einheit `name` an dieser Stelle ausgeführt werden soll. Wenn eine Einheit auf diese Weise mehrmals initialisiert und terminiert werden soll, wird die tatsächliche Initialisierung nur beim ersten Mal und die tatsächliche Terminierung nur beim letzten Mal ausgeführt.

Wenn es eine Initialisierungseinheit mit dem Namen `main` anstelle der normalen Funktion `main` gibt, besteht das Hauptprogramm aus der Initialisierung und der anschließenden Terminierung dieser Einheit.

Währenddessen enthält die globale Variable `prog` mit Typ `str` den Namen des Programms, der sonst in der Funktion `main` als `argv[0]` zur Verfügung steht, und die Variable `args` mit Typ `seq<str>` die Kommandozeilenargumente, die sonst als `argv[1]` bis `argv[argc-1]` verfügbar sind (sofern die Parameter von `main` die üblichen Namen `argc` und `argv` besitzen).

# 10 Offene Funktionen

Eine offene Funktion kann aus beliebig vielen globalen und lokalen Teilfunktionen bestehen.

Eine globale Teilfunktion wird aktiviert bzw. deaktiviert, wenn ein Initialisierungs- bzw. Terminierungsblock (vgl. §9) ausgeführt wird, der sich an der Definitionsstelle der Teilfunktion befindet. Dementsprechend müssen globale Teilfunktionen immer innerhalb einer Initialisierungseinheit definiert werden.

Wenn globale Teilfunktionen derselben offenen Funktion in mehreren Übersetzungseinheiten definiert werden, wird ihre Aktivierungs- und Deaktivierungsreihenfolge dementsprechend durch die Initialisierungseinheiten definiert, in denen sie sich befinden.

Eine lokale Teilfunktion wird aktiviert bzw. deaktiviert, wenn der Konstruktor bzw. Destruktor einer lokalen Variablen ausgeführt wird, die an der Definitionsstelle der Teilfunktion definiert ist.

Beim Aufruf einer offenen Funktion wird immer die zuletzt aktivierte (und noch nicht wieder deaktivierte) Teilfunktion ausgeführt. Jede Teilfunktion kann bei Bedarf beliebig oft die vorige Teilfunktion aufrufen, d. h. die Teilfunktion, die unmittelbar vor ihr aktiviert wurde. Wenn es keine vorige Teilfunktion mehr gibt, ist ein solcher Aufruf wirkungslos und liefert ggf. (wenn der Resultattyp nicht `void` ist) nil.

`R FUNC(f, pars) { impl }` definiert eine globale Teilfunktion der offenen Funktion `f` mit Parameterliste `pars`, Resultattyp `R` und Implementierung `impl`.

`R LFUNC(f, pars) { impl };` definiert analog eine lokale Teilfunktion der offenen Funktion `f` mit Parameterliste `pars`, Resultattyp `R` und Implementierung `impl`. Nach der schließenden geschweiften Klammer muss ein Semikolon stehen, ähnlich wie bei der Initialisierung einer lokalen Variablen mit einem Lambda-Ausdruck. (Tatsächlich wird eine lokale Teilfunktion auf einen Lambda-Ausdruck abgebildet, in dem alle lokalen Variablen aus der Umgebung der Definition per Referenz zugreifbar sind, d. h. der Lambda-Ausdruck beginnt mit `[&]`.)

`R DECLFUNC(f, pars)` deklariert die offene Funktion `f` mit Parameterliste `pars` und Resultattyp `R`, ohne eine Teilfunktion zu definieren, damit sie anschließend verwendet werden kann. Das ist nur erforderlich, wenn zuvor noch keine globale Teilfunktion der Funktion definiert wurde.

Innerhalb einer Teilfunktion kann die vorige Teilfunktion mittels `PREV()` aufgerufen werden. Dabei werden dieselben Argumente übergeben, mit denen die aktuelle Teilfunktion aufgerufen wurde, selbst wenn die Parameter der aktuellen Teilfunktion durch Zuweisungen verändert wurden (es sei denn, es handelt sich um Parameter mit Referenztypen).

`ONLYIF(cond)` im Code einer Teilfunktion (typischerweise ganz am Anfang) ist gleichbedeutend mit

```
if (!(cond)) return PREV();
```

d. h. wenn die Bedingung `cond` nicht erfüllt ist, wird der Aufruf einfach an die vorige Teilfunktion weitergegeben.

`WITHIN(f, pars) { code }` kann in beliebigen Ausdrücken verwendet werden, um zu überprüfen, ob momentan ein Aufruf der offenen Funktion `f` mit Parameterliste `pars` ausgeführt wird. Wenn ja, wird der Code in den geschweiften Klammern wie bei einem Lambda-Ausdruck ausgeführt, in dem die Argumente dieses Funktionsaufrufs in den Parametern der Parameterliste `pars` verfügbar sind, und das gesamte Konstrukt liefert den Wert `true`. Wenn momentan mehrere rekursive Aufrufe der Funktion ausgeführt werden, erhält man die Argumente des innersten Aufrufs. Wenn momentan kein Aufruf der Funktion ausgeführt wird, wird der Code in den geschweiften Klammern nicht ausgeführt, und das gesamte Konstrukt liefert den Wert `false`. Auch hier sind innerhalb der geschweiften Klammern alle lokalen Variablen aus der Umgebung per Referenz zugreifbar.

# Änderungsprotokoll

2022-10-03

- Version zu Beginn des Wintersemesters 2022/2023.

2022-10-11

- Ergänzung von Tabellen und offenen Typen.

2022-10-26

- Ergänzung von Initialisierungseinheiten und offenen Funktionen.