# Objektorientiertes Programmieren mit C++ für Fortgeschrittene

## Kapitel 11

**11. Standard-Template-Library (STL)**

11.1. Überblick

11.2. Container

11.3. Iteratoren

11.4. Algorithmen

# Standard-Template-Library (STL) von C++ : Überblick (1)

## • Einführung

◇ Die **Standard-Template-Library (STL)** ist ein wesentlicher Bestandteil der ANSI-C++-Standard-Bibliothek.
Sie stellt ein relativ mächtiges und effizientes Werkzeug zur Verarbeitung von **Datenmengen** unterschiedlichster Typen zur Verfügung

◇ Es handelt sich um eine **generische Bibliothek** auf der Basis von **C++-Templates.**
Nicht der Typ der zu bearbeitenden Daten steht im Vordergrund, sondern die Art ihrer Verwaltung, der Zugriff zu ihnen und die auf sie anwendbaren Bearbeitungsoperationen.

◇ Die STL besteht im wesentlichen aus **drei** aufeinander abgestimmten **Teilen** :

- ▻ ***Containers Library*** : Definition von Container-Klassen.
  Container-Objekte speichern andere Objekte (→ Datenmengen)
- ▻ ***Iterator Library*** : Definition von Iterator-Klassen.
  Iterator-Objekte ermöglichen den Zugriff zu den in Container-Objekten gespeicherten Objekten (den Elementen einer Menge)
- ▻ ***Algorithm Library*** : Definition von Funktionen zur Bearbeitung von Datenmengen und Elementen von Datenmengen

## • Grundkonzept

◇ Den Kern der STL bilden **7 Container-Klassen** :
**`vector, list, deque, set, multiset, map`** und **`multimap.`**
Alle Container-Klassen sind als **Klassen-Templates** definiert. Dabei ist der Typ der in einem Container zu speichernden Objekte (Elemente) Template-Parameter (→ generische Programmierung).
Ein bestimmter Container kann also immer nur Elemente eines bestimmten Typs speichern.
Die verschiedenen Container-Klassen spiegeln die unterschiedlichen Möglichkeiten zur Realisierung und Verwaltung einer Datenmenge wieder. Dementsprechend besitzen sie jeweils spezifische Vor- und Nachteile.

◇ Die für die verschiedenen Container-Klassen **implementierten Schnittstellen** sind so gehalten, dass wo immer möglich und sinnvoll **gleichnamige Memberfunktionen** (mit **gleichartiger Funktionalität**) existieren.
So stehen u. a. für alle Container-Klassen gleichnamige Funktionen zum Ermitteln der Iteratorgrenzwerte, sowie die Vergleichsoperatoren und der Zuweisungsoperator zur Verfügung.
→ Die verschiedenen Containerklassen lassen sich teilweise **gleichartig verwenden**.

◇ Zusätzlich zu den obigen fundamentalen Container-Klassen sind drei **Container-Adapter-**Klassen definiert :
**`stack, queue`** und **`priority_queue.`**
Diese Container-Adapter passen die Container-Klassen `vector, deque` und `list` an spezielle Anforderungen an.

◇ Zu allen Container-Klassen (außer den Container-Adaptern) sind jeweils zugehörige **Iterator-Klassen** definiert.
Iteratoren erlauben einen **zeigerähnlichen Zugriff** zu den **Elementen eines Containers**. Dabei bieten sie unabhängig von der jeweiligen Container-Klasse die **gleiche Schnittstelle** für den Element-Zugriff. Zu den in einem Container gespeicherten Elementen kann damit gleichartig – unabhängig von der internen Implementierung – zugegriffen werden.
Da die Iteratoren containerklassen-spezifisch implementiert sind, berücksichtigen sie die im Container eingesetzte Datenorganisation.
Es gibt verschiedene **Iterator-Kategorien**, die – zusätzlich zu einer für alle Iteratoren gleichartigen Grundfunktionalität – unterschiedliche Operationen zur Verfügung stellen.
Nicht alle Iterator-Kategorien sind für alle Container-Klassen anwendbar.

◇ Zahlreiche **Algorithmen** zur Bearbeitung von Mengen als Ganzes bzw. Mengenelementen (z.B. Traversieren, Suchen, Finden, Sortieren, Kopieren usw.) sind durch **freie Funktions-Templates** implementiert. Sie arbeiten mit **Iteratoren** (Template-Parameter, Funktions-Parameter).
Viele dieser Funktionen können auf alle Container-Arten angewendet werden, andere nur auf solche, die eine spezielle Iterator-Kategorie anbieten.
Häufig können einer Algorithmen-Funktion **Hilfsfunktionen als Parameter** (Funktions-Pointer oder Funktions-Objekte) übergeben werden, mit denen eine Anpassung des Algorithmus an spezielle Bedürfnisse erreicht wird.

◇ Diese Art der Implementierung – **Trennung von Daten und Operationen** – **widerspricht** zwar dem **objektorientierten Gedanken**. Sie hat aber gegenüber der Implementierung der Algorithmen als Memberfunktionen der Container den **Vorteil**, dass die **Algorithmen** jeweils **nur einmal** (und nicht pro Container-Klasse) zu **implementieren** sind.

# Standard-Template-Library (STL) von C++ : Überblick (2)

## • Anmerkungen zu Container-Elementen

◇ Alle Container der STL besitzen eine **Wert-Semantik**.
Das bedeutet, dass die in einen Container aufgenommenen Elemente als **Kopie** und nicht als Referenz (Pointer) **abgelegt** werden und auch **Kopien** der enthaltenen Elemente **zurückgeliefert** werden.
**Vorteile** : Probleme, wie Verweise auf nicht mehr existierende Elemente, können nicht auftreten
**Nachteile** : Das Kopieren der Elemente dauert i.a. länger als das Kopieren ihrer Adressen
Ein Element kann nicht gleichzeitig von mehreren Containern änderbar verwaltet werden.

◇ Die prinzipiell mögliche Verwendung von **Pointern als Container-Elemente** sollte i.a. **vermieden** werden.
Hierdurch lässt sich zwar indirekt eine Referenz-Semantik implementieren, die aber Probleme beinhaltet :
- So kann es vorkommen, dass als Elemente enthaltene Pointer auf Objekte zeigen, die gar nicht mehr existieren.
- Außerdem arbeiten Vergleiche von Zeigern mit Adressen und nicht mit den durch die Zeiger referierten Objekten (Adress-Vergleich statt Werte-Vergleich).

◇ Die Container der STL können prinzipiell **Elemente beliebigen Typs** verwalten.
Allerdings müssen diese Typen die für das **Kopieren notwenigen Eigenschaften** implementieren :

▻ Es muss ein **Copy-Konstruktor** öffentlich verfügbar sein (ein selbstdefinierter, wenn der Default-Copy-Konstruktor nicht richtig arbeiten würde, andernfalls reicht dieser aus).
Er sollte ein gutes Zeitverhalten besitzen.

▻ Es muss ein **Zuweisungsoperator** öffentlich verfügbar sein (ein selbstdefinierter, wenn der Default-Zuweisungs-operator nicht richtig arbeiten würde, andernfalls reicht dieser aus).

▻ Der **Destruktor** muss öffentlich verfügbar sein, da ein Element beim Entfernen aus dem Container zerstört werden muss.

◇ Darüber hinaus müssen die Element-Typen gegebenenfalls **weitere Anforderungen** erfüllen :

▻ Für die Anwendung einiger Memberfunktionen bestimmter Container-Klassen muss der **Default-Konstruktor** definiert sein.

▻ Für Suchfunktionen muss der **Vergleichsoperator** für **Gleichheit** (`==`) definiert sein.

▻ Für Sortierfunktionen muss der **Vergleichsoperator** für "**kleiner als**" (`<`) definiert sein.

## • Fehlerbehandlung in der STL

◇ In der STL finden praktisch **keinerlei Überprüfungen** auf **Fehler** wie
- Überschreitung von Bereichsgrenzen,
- Verwendung von falschen, fehlerhaften oder ungültigen Iteratoren,
- Zugriff zu nicht vorhandenen Elementen

statt.
Beim Auftritt derartiger Fehler ist das Verhalten der entsprechenden Bibliotheks-Komponente und damit des Programms undefiniert.

◇ Lediglich **zwei Memberfunktionen** der Container-Klasse **`vector`** (**`at()`** und **`reserve()`**) sowie **eine Memberfunktion** der Container-Klasse **`deque`** (**`at()`**) können eine **Exception auslösen**.

◇ Darüber hinaus können gegebenenfalls lediglich die von anderen – durch die STL benutzten – Bibliothekskomponenten oder sonstigen aufgerufenen Funktionen erzeugten Exceptions auftreten (z.B. `bad_alloc` bei Allokationsfehlern).

◇ Die STL behandelt (fängt) diese Exceptions aber nicht. Tritt eine Exception auf, ist der Zustand aller beteiligten STL-Objekte undefiniert. Wird z. B. beim Einfügen eines Elements in einen Container eine Exception generiert, so gerät der Container in einen undefinierten Zustand, der seine weitere sinnvolle Verwendung verhindert.

◇ Der Grund für die nicht vorhandene Fehlerüberprüfung liegt in dem **Grundgedanken**, die STL mit **optimalem Zeit-verhalten** zu implementieren.
Fehlerüberprüfungen kosten aber Zeit.

# Standard-Template-Library (STL) von C++ : Container (Überblick) (1)

## • Container-Arten

◇ **Sequentielle Container**
Sie speichern die Elemente als **geordnete Mengen**, in denen jedes Element eine bestimmte **Position** besitzt, die durch den **Zeitpunkt** und **Ort** des **Einfügens** festgelegt ist.
Die enthaltenen Elemente sind damit **linear angeordnet** und können über ihre jeweilige Position angesprochen werden.
Typischerweise erfolgt die Speicherung der Elemente in dynamischen **Arrays** bzw. **Listen**.

◇ **Assoziative Container**
Sie speichern die Elemente als **sortierte Mengen**, in denen die **Position** eines Elementes durch ein **Sortierkriterium** bestimmt ist. → sehr gutes Zeitverhalten bei Suchen und Finden.
Ein neues Element wird entsprechend dem Sortierkriterium **automatisch sortiert** eingefügt.
Der Zugriff zu den Elementen erfolgt **assoziativ** über **Suchschlüssel** (*keys*).
Typischerweise erfolgt die Speicherung der Elemente in einem balancierten **Binärbaum**.

## • Sequentielle Container

◇ Die Klassen-Templates für die sequentiellen Container besitzen **2 Template-Parameter** :
**`<T, Allocator = allocator<T> >`**

▻ **`T`** ist der **Datentyp** der zu **verwaltenden Objekte** (Elemente).

▻ **`Allocator`** ist eine **Allokator-Klasse**, die das Speichermodell für die zu verwendende dynamische Speicherverwaltung definiert. Als **Default** ist die Standard-Allokator-Klasse **`allocator<T>`**, die `new` und `delete` zur Speicherallokation verwendet, festgelegt.

◇ **Vektor :**

```
template <class T, class Allocator = allocator<T> > class vector;
```

Ein Vektor (-Container) verwaltet die Elemente in einem dynamischen Array. Er ermöglicht einen direkten wahlfreien Zugriff zu den einzelnen Elementen. Hierfür existieren der Indexoperator und Direktzugriffs-Iteratoren.
Das Anhängen und Löschen von Elementen am Ende des Arrays erfolgt optimal schnell.
Ein Einfügen oder Löschen von Elementen mitten im Array ist dagegen zeitaufwändig (Verschieben von Elementen !).
Vektoren sind daher bevorzugt einzusetzen, wenn Einfüge- und Löschoperationen vor allem am Ende stattfinden.

◇ **Deque :**

```
template <class T, class Allocator = allocator<T> > class deque;
```

**Deque = *double ended queue***
Ein Deque (-Container) verwaltet die Elemente in einem nach beiden Seiten offenen (verlängerbaren) dynamischen Array. Damit ist das Einfügen und Löschen von Elementen nicht nur am Ende sondern auch am Anfang optimal schnell, aber etwas langsamer als bei einem Vektor. Auch hier ist das Einfügen oder Löschen in der Mitte zeitaufwändig.
Deques gestatten ebenfalls einen direkten wahlfreien Zugriff zu den einzelnen Elementen über einen Index bzw. mittels eines Direktzugriffs-Iterators.
Sie sollten dann gewählt werden, wenn Einfüge- u./o. Löschoperationen häufig sowohl am Ende als auch am Anfang stattfinden.

◇ **Liste :**

```
template <class T, class Allocator = allocator<T> > class list;
```

Ein Listen-Container (eine Liste) verwaltet seine Elemente in einer doppelt verketteten Liste.
Ein direkter wahlfreier Zugriff zu den einzelnen Elementen ist nicht möglich, Indexoperator und Direktzugriffs-Iteratoren sind daher nicht implementiert. Das Einfügen und Löschen von Elementen erfolgt an allen Positionen gleich schnell.
Listen sind immer dann bevorzugt einzusetzen, wenn viele Einfüge- u/o Löschoperationen insbesondere in der Mitte der gespeicherten Sequenz stattfinden.

# Standard-Template-Library (STL) von C++ : Container (Überblick) (2)

- **Container-Adapter**

  ◇ Sie implementieren jeweils eine **spezielle Container-Funktionalität**, die sie auf einen **sequentiellen Container** abbilden.

  ◇ Für Container-Adapter existieren keine Iteratoren. Auf sie können daher auch nicht die Algorithmen der STL direkt angewendet werden.

  ◇ Die Klassen-Templates für die Container-Adapter besitzen **2** bzw. **3 Template-Parameter** :
  **`<T, Container, [Compare]>`**

  ▻ **`T`** ist der **Datentyp** der zu **verwaltenden Objekte** (Elemente).

  ▻ **`Container`** ist die Container-Klasse, auf die die Abbildung erfolgt.
  Ein Objekt einer Container-Adapter-Klasse verwendet zur eigentlichen Speicherung seiner Elemente ein Objekt dieser Klasse als `protected`-Komponente

  ▻ **`Compare`** ist eine **Comparator-Klasse**, die eine Vergleichsfunktion für Objekte vom Typ `T` definiert.
  Eine Comparator-Klasse ist eine Klasse, für die der Funktionsaufruf-Operator (`operator()`) mit der Funktionalität einer Vergleichsfunktion überladen ist (→ Funktions-Objekte)
  Dieser Parameter wird nur für Priority-Queues verwendet.

  ◇ **Stack :**

```
template <class T, class Container = deque<T> > class stack;
```

  Implementiert die Funktionalität eines **Stacks** (Kellerspeicher, **LIFO**) unter Verwendung der Container-Klassen `deque` (Default), `vector` oder `list`.

  ◇ **Queue**

```
template <class T, class Container = deque<T> > class queue;
```

  Implementiert die Funktionalität eines **Pufferspeichers** (**FIFO**) unter Verwendung der Container-Klassen `deque` (Default) oder `list`.

  ◇ **Priority-Queue :**

```
template <class T, class Container = vector<T>,
          class Compare = less<T> > class priority_queue;
```

  Implementiert einen **Pufferspeicher**, bei der die Elemente nach einer durch ein **Sortierkriterium** festgelegten "Priorität" wieder ausgelesen werden können. Es wird immer das Element mit der **höchsten Priorität** zurückgeliefert.
  Das Sortierkriterium kann als Template-Parameter übergeben werden. **Default** ist die Comparator-Klasse **`less<T>`**,
  d.h. die **höchste Priorität** hat das **größte** Element.
  Priority-Queues verwenden zur Speicherung ihrer Elemente die Container-Klasse `vector` (Default) oder `deque`.

# Standard-Template-Library (STL) von C++ : Container (Überblick) (3)

## • Assoziative Container

◇ Die Klassen-Templates für assoziative Container besitzen **3** bzw. **4 Template-Parameter** :
**`<Key, [T], Compare, Allocator>`**

- **`Key`** ist der **Datentyp** des **Suchschlüssels**, nach dem die zu **verwaltenden Objekte** (Elemente) sortiert werden. Bei den Container-Klassen `set` und `multiset` sind Suchschlüssel und zu verwaltendes Objekt jeweils identisch. Dieser Typ muss sortierbar sein, d.h. es muss der **Vergleichsoperator `<`** für ihn anwendbar sein.
- **`T`** ist der **Datentyp** eines mit dem Suchschlüssel zu einem **Paar** verknüpften "**Werts**". Dieser Parameter existiert nur bei den Container-Klassen `map` und `multimap`, bei denen die mit dem Suchschlüssel verknüpften "Werte" die eigentlichen **verwalteten Objekte** sind.
- **`Compare`** ist eine **Comparator-Klasse**, die eine Vergleichsfunktion für Objekte vom Typ `Key` definiert. Eine Comparator-Klasse ist eine Klasse, für die der Funktionsaufruf-Operator (`operator()`) mit der Funktionalität einer Vergleichsfunktion überladen ist (→ Funktions-Objekte). Sie legt das **Sortierkriterium** fest. Als **Default** ist die Comparator-Klasse **`less<Key>`** vorgesehen.
- **`Allocator`** ist eine **Allokator-Klasse**, die das Speichermodell für die zu verwendende dynamische Speicherverwaltung definiert. Als **Default** ist die Standard-Allokator-Klasse **`allocator<...>`**, die `new` und `delete` zur Speicherallokation verwendet, festgelegt.

◇ **Set :**

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> > class set
```

Ein Set (-Container) verwaltet nach ihrem "Wert" sortierte Elemente, d.h. die Elemente selbst bilden auch den Suchschlüssel. Jedes Element darf nur einmal in einem Set vorkommen.
Der Zugriff zu einem gesuchten Element erfolgt sehr schnell. Auch das Einfügen u/o. Löschen von Elementen besitzt an allen Stellen ein gutes Zeitverhalten. Die Elemente von Sets können auch sequentiell durchlaufen werden.
Die Elemente können nicht direkt geändert werden. Eine Änderung könnte die Sortierung ungültig machen. Um ein Element zu ändern, muss es daher aus dem Set entfernt werden und nach der Änderung als neues Element wieder eingefügt werden.

◇ **Multiset :**

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> > class multiset
```

Ein Multiset (-Container) entspricht einem Set mit dem Unterschied, dass Elemente auch mehrfach enthalten sein können.

◇ **Map** ("Dictionary", "assoziatives Array") **:**

```
template <class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<pair<const Key,T> > > class map
```

Ein(e) Map (-Container) speichert Schlüssel-/Werte-Paare als Elemente. Die Elemente sind nach dem (Such-)Schlüssel sortiert. Der Suchschlüssel dient zum Auffinden des mit ihm assoziierten Werts (das eigentliche verwaltete Objekt). Jeder Suchschlüssel darf nur einmal in einer Map vorkommen.
Die Eigenschaften entsprechen denen eines Sets, mit dem Unterschied, dass die enthaltenen Elemente Paare sind. Der Schlüssel eines Elements darf nicht geändert werden, wohl aber sein Wert.

◇ **Multimap :**

```
template <class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<pair<const Key,T> > > class multimap
```

Ein Multimap (-Container) entspricht einer Map mit dem Unterschied, dass mehrere Elemente mit dem gleichen Schlüssel enthalten sein können.

# Standard-Template-Library (STL) von C++ : Container (Überblick) (4)

- **Für alle Container-Klassen definierte Memberfunktionen**

◇ **Anmerkungen :**

▻ In der folgenden Zusammenstellung steht **`Container`** für eine der Container-Klassen,
z.B. bei einem **Vektor** für : **`vector<T, Allocator>`**
z.B. bei einer **Map** für : **`map<Key, T, Compare, Allocator>`**

▻ **`size_type`** ist ein implementierungsabhängiger innerhalb der jeweiligen Containerklasse definierter vorzeichenloser Ganzzahl-Typ. Außerhalb der Qualifizierung muss der **vollqualifizierende Typname** verwendet werden.
Er wird über die Allokator-Klasse definiert und entspricht bei ISO-konformen Containern dem Typ **`std::size_t`**.

▻ Die mit ✲ markierten Funktionen sind für **Container-Adapter nicht** (explizit) definiert

| Funktion | Beschreibung |
|---|---|
| **Default-Konstruktor** | Konstruktor für Default-Initialisierung des Containers |
| **Konstruktor(en)** mit Parametern | Konstruktor(en) für unterschiedliche Initialisierungsmethoden |
| ✲ **Copy-Konstruktor** | Initialisierung eines Containers mit Kopie eines existierenden Containers |
| `size_type` **`size()`** `const;` | Anzahl der aktuell im Container enthaltenen Elemente (Container-Größe) |
| ✲ `size_type` **`max_size()`** `const;` | maximale Anzahl der Elemente, die ein Container aufnehmen kann |
| `bool` **`empty()`** `const;` | `true`, wenn Container leer, sonst `false` |
| ✲ `void` **`swap`**`(Container& b);` | vertauscht den Inhalt des aktuellen Containers mit dem des Containers b |
| ✲ `void` **`clear`**`();` | entfernt alle Elemente aus dem Container |
| ✲ `Container&` **`operator=`**`(const Container& b);` | Zuweisung des Inhalts von Container `b` an akt. Container |
| `iterator` **`insert`**`(iterator pos, const value_type& val);` | fügt ein neues Element mit dem Wert `val` an der Position `pos` ein.<br>`value_type` ist ein container-spezifischer Typ |
| ✲ `iterator` **`erase`**`(iterator pos);`<br>✲ `iterator` **`erase`**`(iterator fi, iterator la);`<br>**Anm.** : bei assoziativen Containern Rückgabetyp : `void` | löscht das Element an der Iterator-Pos. `pos`<br>löscht alle Elemente zwischen `fi` (einschliesslich) und `la` (ausschließlich)<br>liefert Iterator, der auf das folgende Element zeigt |
| ✲ `iterator` **`begin`**`();`<br>✲ `const_iterator` **`begin`**`() const;` | liefert Iterator , der auf das erste Element zeigt |
| ✲ `iterator` **`end`**`();`<br>✲ `const_iterator` **`end`**`() const;` | liefert Iterator, der auf die Position unmittelbar nach dem letzen Element zeigt |
| ✲ `reverse_iterator` **`rbegin`**`();`<br>✲ `const_reverse_iterator` **`rbegin`**`() const;` | liefert einen Reverse-Iterator, der auf die Position unmittelbar nach dem letzten Element zeigt |
| ✲ `reverse_iterator` **`rend`**`();`<br>✲ `const_reverse_iterator` **`rend`**`() const;` | liefert einen Reverse-Iterator, der auf das erste Element zeigt |

# Standard-Template-Library (STL) von C++ : Container (Überblick) (5)

- **Für alle Container-Klassen definierte freie Funktionen**

  ◇ Hierbei handelt es sich – bis auf eine Ausnahme – um Operatorfunktionen für die **Vergleichsoperatoren.**

  ◇ **Anmerkungen :**

  ▻ Die als **Funktions-Templates** definierten Funktionen werden in der folgenden Zusammenstellung in **vereinfachter Darstellung** angegeben.
  Dabei steht **`Container`** für eine der Container-Klassen (s. "Allen Containern gemeinsame Memberfunktionen")
  Statt z. B. die **nur** für **Vektoren** geltende Darstellung anzugeben

```
template <class T, class Allocator>
  bool operator==(const vector<T, Allocator>& x,
                  const vector<T, Allocator>& y);
```

  wird die **allgemeingültige Formulierung** verwendet:

```
  bool operator==(const Container& x, const Container& y);
```

  ▻ **Alle** hier aufgelisteten Funktionen sind **nicht** für den Container-Adapter **`priority_queue`** definiert.

  ▻ Die mit ✲ markierte Funktion ist für **Container-Adapter nicht** (explizit) definiert

| | |
|---|---|
| `bool operator==(const Container& x, const Container& y);` | `true` wenn `x==y`, sonst `false`<br><br>Zwei Container derselben Klasse sind gleich, wenn sie die gleichen Elemente in der gleichen Reihenfolge besitzen.<br>Für den Vergleich wird für alle Elemente der Reihe nach jeweils `operator==()` aufgerufen. |
| `bool operator!=(const Container& x, const Container& y);` | `true` wenn `x!=y`, sonst `false` |
| `bool operator<(const Container& x, const Container& y);` | `true` wenn `x<y`, sonst `false` |
| `bool operator<=(const Container& x, const Container& y);` | `true` wenn `x<=y`, sonst `false` |
| `bool operator> (const Container& x, const Container& y);` | `true` wenn `x>y`, sonst `false` |
| `bool operator>=(const Container& x, const Container& y);` | `true` wenn `x>=y`, sonst `false` |
| ✲ `void swap(Container& x, Container& y);` | vertauscht den Inhalt des Containers `x` mit dem Inhalt des Containers `y` |

# Standard-Template-Library (STL) von C++ : Klassen-Template `vector` (1)

- **Eigenschaften**

  ◇ Implementierung von **Vektoren**. Ein **Vektor** (-Container) verwaltet die Elemente in einem **dynamischen Array**.
  Die Elemente besitzen eine **definierte Reihenfolge** ("*ordered collection*")

  ◇ Zu den einzelnen Elementen kann **direkt wahlfrei zugegriffen** werden.
  Hierfür existieren der **Indexoperator** und **Direktzugriffs-Iteratoren**.

  ◇ Das **Anhängen** und **Löschen** von Elementen **am Ende** des Arrays erfolgt **optimal schnell**.
  Ein Einfügen oder Löschen von Elementen **mitten im Array** ist dagegen **zeitaufwändig** (Verschieben von Elementen!).
  → Vektoren sind daher **bevorzugt einzusetzen**, wenn **Einfüge- und Löschoperationen** vor allem **am Ende** stattfinden.

  ◇ Die **Definition** des Klassen-Templates `vector<>` befindet sich in der **Headerdatei** **`<vector>`**

```
template <class T, class Allocator = allocator<T> >
  class vector
  { // ...
  };
```

- **Konstruktoren**

| | |
|---|---|
| `explicit vector(const Allocator& = Allocator());` | Erzeugung eines Vektors der Länge 0 |
| `explicit vector(size_type n, const T& val = T(), const Allocator& = Allocator());` | Erzeugung eines Vektors der Länge n, Initialisierung der Elemente mit dem Wert `val` |
| `template <class InputIterator> vector(InputIterator first, InputIterator last, const Allocator& = Allocator());` | Erzeugung eines Vektors, dessen Elemente mit den zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elementen eines anderen Containers initialisiert werden |
| `vector(const vector<T, Allocator>& x);` | Copy-Konstruktor |

- **Zusätzliche spezifische Memberfunktionen (Auswahl)**

  ◇ **Methoden zum Ermitteln und Ändern der Kapazität sowie Änderung der aktuellen Größe**

  **Kapazität** = maximale Anzahl der Elemente, die der Vektor ohne Neuallokation enthalten kann.
  = Größe des aktuell allozierten Arrays

| | |
|---|---|
| `size_type capacity() const;` | Rückgabe der aktuellen Kapazität des Vektors |
| `void reserve(size_type n);` | Vergrößerung der Kapazität auf einen Wert `>=n`, wenn `n >` akt. Kapazität<br>Keine Wirkung, wenn `n<=` aktuelle Kapazität<br>Falls `n > max_size()` ist, wird Exception `length_error` geworfen<br>**Achtung** : Alle **Referenzen**, **Pointer** und **Iteratoren** auf Elemente des Vektors werden nach einer Kapazitätsvergrößerung (Neu-Allokation !) **ungültig** |
| `void resize(size_type sz, T c= T());` | Veränderung der akt. Größe des Vektors auf den Wert `sz` (Vergrößerung oder Verkleinerung)<br>bei Vergrößerung : Initialisierung der neuen Elemente mit dem Wert `c` |

# Standard-Template-Library (STL) von C++ : Klassen-Template `vector` (2)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)**

◇ **Methoden zum Elementzugriff**

| | |
|---|---|
| `T& operator[](size_type n);`<br>`const T& operator[](size_type n) const;` | Rückgabe des Elements an der Position mit dem Index `n`<br>Falls `n >= size()` ist das Verhalten undefiniert |
| `T& at(size_type n);`<br>`const T& at(size_type n) const;` | Rückgabe des Elements an der Position mit dem Index `n`<br>Falls `n >= size()` wird Exception `out_of_range` geworfen |
| `T& front();`<br>`const T& front() const;` | Rückgabe des ersten Elements (Index `0`) |
| `T& back();`<br>`const T& back() const;` | Rückgabe des letzten Elements (Index `size()-1`) |

◇ **Methoden zum Einfügen und Löschen von Elementen**

**Achtung** : Durch das Einfügen mittels **`insert()`** werden alle **Referenzen**, **Pointer** und **Iteratoren** auf Elemente ab der Einfügeposition **ungültig**. Bei erforderlicher Neu-Allokation gilt das auch für alle übrigen Positionen.

| | |
|---|---|
| `void push_back(const T& x);` | Einfügen des Objekts (Kopie) `x` als neues letztes Element |
| `void pop_back();` | Löschen des letzten Elements |
| `iterator insert(iterator pos, const T& x);` | Einfügen des Objekts `x` als neues Element an der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um eine Position weitergeschoben)<br>Rückgabe : die Einfügeposition `pos` |
| `void insert(iterator pos, size_type n, const T& x);` | Einfügen von `n` Kopien des Objekts `x` als neue Elemente ab der Position `pos`<br>(das bisherige Element an dieser Position und alle folgenden Elemente werden um `n` Positionen weitergeschoben) |
| `template <class InputIterator> void insert(iterator pos, InputIterator first, InputIterator last);` | Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers befindlichen Elemente ab der Position `pos`.<br>(das bisherige Element an dieser Position und alle folgenden Elemente werden um entsprechend viele Positionen weitergeschoben) |

# Standard-Template-Library (STL) von C++ : Klassen-Template `vector` (3)

- **Einfaches Demonstrationsprogramm `vectordem`**

```
// C++-Quelldatei vectordem_m.cpp --> Programm vectordem
// Einfaches Demo-Programm zum Klassen-Template vector<> der STL

#include <vector>
#include <string>
#include <iostream>
using namespace std;

template <class T>
  void showSizeData(const vector<T>& vec, ostream& out)
  { out << "size()     : " << vec.size() << endl;
    out << "max_size() : " << vec.max_size() << endl;
    out << "capacity() : " << vec.capacity() << endl;
  }

template <class T>
  void showContent(const vector<T>& vec, ostream& out)
  { for (int i=0; i<vec.size(); i++)
      out << vec[i] << ' ';
    out << endl;
  }

int main(void)
{
  vector<string> satz;
  cout << "\nleerer string-Vector :\n";
  showSizeData(satz, cout);
  vector<string>::size_type nsz = 5;
  satz.reserve(nsz);
  cout << "\nnach reserve(" << nsz << ") :\n";
  showSizeData(satz, cout);
  satz.push_back("Achtung,");
  satz.push_back("dies");
  satz.push_back("ist");
  satz.push_back("ein");
  satz.push_back("Satz");
  satz.push_back("als");
  satz.push_back("Beispiel");
  satz.push_back("!");
  cout << "\nstring-Vector enthaelt jetzt " << satz.size() << " Elemente :\n";
  showSizeData(satz, cout);
  cout << "\nInhalt :\n";
  showContent(satz, cout);
  swap(satz[1], satz[2]);
  string hilf=satz[satz.size()-2];
  satz.insert(satz.begin()+4, hilf);
  satz.erase(satz.end()-1);
  satz.pop_back();
  satz.back()="?";
  satz.insert(satz.begin()+5, "fuer");
  satz.insert(satz.begin()+6, "einen");
  satz.insert(satz.begin()+7, "wirklich");
  satz.insert(satz.begin()+8, "guten");
  cout << "\nstring-Vector nach Manipulation :\n";
  showSizeData(satz, cout);
  cout << "\nInhalt :\n";
  showContent(satz, cout);
  return 0;
}
```

## Standard-Template-Library (STL) von C++ : Klassen-Template `vector` (4)

- **Ausgabe des Demonstrationsprogramms `vectordem`**

```
leerer string-Vector :
size()     : 0
max_size() : 268435455
capacity() : 0

nach reserve(5) :
size()     : 0
max_size() : 268435455
capacity() : 5

string-Vector enthaelt jetzt 8 Elemente :
size()     : 8
max_size() : 268435455
capacity() : 10

Inhalt :
Achtung, dies ist ein Satz als Beispiel !

string-Vector nach Manipulation :
size()     : 11
max_size() : 268435455
capacity() : 20

Inhalt :
Achtung, ist dies ein Beispiel fuer einen wirklich guten Satz ?
```

# Standard-Template-Library (STL) von C++ : Klassen-Template `deque` (1)

- **Eigenschaften**

  ◇ Implementierung von **Deques** (Deque = ***double ended queue***)

  ◇ Ein Deque (-Container) verwaltet die Elemente in einem **nach beiden Seiten offenen** (verlängerbaren) **dynamischen Array**. Realisiert wird dieses typischerweise durch **mehrere Speicherblöcke**, wobei der erste Block logisch in die eine Richtung und der letzte in die andere Richtung wächst.
  Auch die Elemente eines Deques besitzen eine **definierte Reihenfolge** ("*ordered collection*")

  ◇ Zu den einzelnen Elementen kann ebenfalls **direkt wahlfrei zugegriffen** werden.
  Hierfür stehen der **Indexoperator** und **Direktzugriffs-Iteratoren** zur Verfügung.

  ◇ Das **Einfügen** und **Löschen** von Elementen ist nicht nur **am Ende** sondern auch **am Anfang** optimal **schnell**, aber etwas langsamer als bei einem Vektor. Auch hier ist das Einfügen oder Löschen in der Mitte zeitaufwändig.
  Deques sollten dann gewählt werden, wenn Einfüge- u./o. Löschoperationen häufig sowohl am Ende als auch am Anfang stattfinden.

  ◇ Die **Definition** des Klassen-Templates `deque<>` befindet sich in der **Headerdatei** **`<deque>`**

```
template <class T, class Allocator = allocator<T> >
  class deque
  { // ...
  };
```

- **Konstruktoren**

| | |
|---|---|
| `explicit deque(const Allocator& = Allocator());` | Erzeugung eines Deques der Länge 0 |
| `explicit deque(size_type n, const T& val = T(), const Allocator& = Allocator());` | Erzeugung eines Deques der Länge n, Initialisierung der Elemente mit dem Wert `val` |
| `template <class InputIterator> deque(InputIterator first, InputIterator last, const Allocator& = Allocator());` | Erzeugung eines Deques, dessen Elemente mit den zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elementen eines anderen Containers initialisiert werden |
| `deque(const deque<T, Allocator>& x);` | Copy-Konstruktor |

- **Zusätzliche spezifische Memberfunktionen (Auswahl)**

  ◇ **Methode zur Änderung der aktuellen Größe**

| | |
|---|---|
| `void resize(size_type sz, T c= T());` | Veränderung der akt. Größe des Deques auf den Wert `sz` (Vergrößerung oder Verkleinerung)<br>bei Vergrößerung : Initialisierung der neuen Elemente mit dem Wert `c` |

# Standard-Template-Library (STL) von C++ : Klassen-Template `deque` (2)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)**

◇ **Methoden zum Elementzugriff**

| | |
|---|---|
| `T& operator[](size_type n);`<br>`const T& operator[](size_type n) const;` | Rückgabe des Elements an der Position mit dem Index `n`<br>Falls `n >= size()` ist das Verhalten undefiniert |
| `T& at(size_type n);`<br>`const T& at(size_type n) const;` | Rückgabe des Elements an der Position mit dem Index `n`<br>Falls `n >= size()` wird Exception `out_of_range` geworfen |
| `T& front();`<br>`const T& front() const;` | Rückgabe des ersten Elements (Index `0`) |
| `T& back();`<br>`const T& back() const;` | Rückgabe des letzten Elements (Index `size()-1`) |

◇ **Methoden zum Einfügen und Löschen von Elementen**

**Achtung** : Durch das Einfügen mittels **`insert()`**

- in der **Mitte** des Deques werden alle **Referenzen**, **Pointer** und **Iteratoren** auf Elemente des Deques **ungültig**
- an einem der beiden **Enden** werden nur die **Iteratoren** auf Deque-Elemente **ungültig**

| | |
|---|---|
| `void push_back(const T& x);` | Einfügen des Objekts (Kopie) `x` als neues letztes Element |
| `void push_front(const T& x);` | Einfügen des Objekts (Kopie) `x` als neues erstes Element |
| `void pop_back();` | Löschen des letzten Elements |
| `void pop_front();` | Löschen des ersten Elements |
| `iterator insert(iterator pos, const T& x);` | Einfügen des Objekts `x` als neues Element an der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um eine Position weitergeschoben)<br>Rückgabe : die Einfügeposition `pos` |
| `void insert(iterator pos,`<br>`size_type n, const T& x);` | Einfügen von `n` Kopien des Objekts `x` als neue Elemente ab der Position `pos`<br>(das bisherige Element an dieser Position und alle folgenden Elemente werden um `n` Positionen weitergeschoben) |
| `template <class InputIterator>`<br>`void insert(iterator pos,`<br>`InputIterator first,`<br>`InputIterator last);` | Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers befindlichen Elemente ab der Position `pos`.<br>(das bisherige Element an dieser Position und alle folgenden Elemente werden um entsprechend viele Positionen weitergeschoben) |

# Standard-Template-Library (STL) von C++ : Klassen-Template deque (3)

- **Einfaches Demonstrationsprogramm dequedem**

```
// C++-Quelldatei dequedem_m.cpp --> Programm dequedem
// Einfaches Demo-Programm zum Klassen-Template deque<> der STL

#include <deque>
#include <iostream>
using namespace std;

template <class T>
  void showSizeData(const deque<T>& deq, ostream& out)
  { out << "size()     : " << deq.size() << endl;
    out << "max_size() : " << deq.max_size() << " (" << hex << deq.max_size()
        << ')' << dec << endl;
  }

template <class T>
  void showContent(const deque<T>& deq, ostream& out)
  { for (int i=0; i<deq.size(); i++)
      out << deq[i] << "  ";
    out << endl;
  }

int main(void)
{
  deque<float> fq;
  cout << "\nleerer float-Deque :\n";
  showSizeData(fq, cout);
  cout << "\nnach Einfuegen von Elementen :\n";
  fq.push_back(2.2f);
  fq.push_front(1.1f);
  fq.resize(4, 3.3f);
  showContent(fq, cout);
  cout << "\nnach Entfernung des ersten und letzten Elements :\n";
  fq.pop_front();
  fq.pop_back();
  showContent(fq, cout);
  cout << "\nnach weiterer Modifikation des Inhalts :\n";
  fq.push_front(5.5f);
  fq.push_front(6.6f);
  fq[2]=-7.7f;
  fq.pop_back();
  fq.push_back(8.8f);
  showContent(fq, cout);
  return 0;
}
```

- **Ausgabe des Demonstrationsprogramms dequedem**

```
leerer float-Deque :
size()     : 0
max_size() : 1073741823 (3fffffff)

nach Einfuegen von Elementen :
1.1  2.2  3.3  3.3

nach Entfernung des ersten und letzten Elements :
2.2  3.3

nach weiterer Modifikation des Inhalts :
6.6  5.5  -7.7  8.8
```

# Standard-Template-Library (STL) von C++ : Klassen-Template `list` (1)

- **Eigenschaften**

  ◇ Implementierung von **Listen-Containern** (Listen)
  Ein Listen-Container verwaltet seine Elemente in einer **doppelt verketteten Liste**.

  ◇ Die Elemente besitzen eine **definierte Reihenfolge**, ein direkter **wahlfreier Zugriff** zu einzelnen Elementen ist jedoch **nicht möglich**. Indexoperator und Direktzugriffs-Iteratoren sind daher nicht implementiert.

  ◇ Durch das **Einfügen** oder **Löschen** von Elementen werden **Verweise** auf andere Elemente **nicht ungültig**

  ◇ Das **Einfügen** und **Löschen** von Elementen erfolgt an **allen Positionen gleich schnell**.
  Listen sind daher immer dann bevorzugt einzusetzen, wenn viele Einfüge- u/o Löschoperationen insbesondere in der Mitte der gespeicherten Sequenz stattfinden.

  ◇ Die **Definition** des Klassen-Templates `list<>` befindet sich in der **Headerdatei** **`<list>`**

```
template <class T, class Allocator = allocator<T> >
  class list
  { // ...
  };
```

- **Konstruktoren**

| | |
|---|---|
| `explicit list(const Allocator& = Allocator());` | Erzeugung einer Liste der Länge 0 |
| `explicit list(size_type n, const T& val = T(), const Allocator& = Allocator());` | Erzeugung einer Liste der Länge n,<br>Initialisierung der Elemente mit dem Wert `val` |
| `template <class InputIterator> list(InputIterator first, InputIterator last, const Allocator& = Allocator());` | Erzeugung einer Liste, deren Elemente mit den zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elementen eines anderen Containers initialisiert werden |
| `list(const list<T, Allocator>& x);` | Copy-Konstruktor |

- **Zusätzliche spezifische Memberfunktionen (Auswahl)**

  ◇ **Methode zur Änderung der aktuellen Größe**

| | |
|---|---|
| `void resize(size_type sz, T c= T());` | Veränderung der akt. Größe der Liste auf den Wert `sz`<br>(Vergrößerung oder Verkleinerung)<br>bei Vergrößerung : Initialisierung der neuen Elemente mit dem Wert `c` |

  ◇ **Methoden zum Elementzugriff**

| | |
|---|---|
| `T& front();`<br>`const T& front() const;` | Rückgabe des ersten Elements |
| `T& back();`<br>`const T& back() const;` | Rückgabe des letzten Elements |

# Standard-Template-Library (STL) von C++ : Klassen-Template `list` (2)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)**

◇ **Methoden zum Einfügen und Löschen von Elementen**

**Achtung**: Durch das Einfügen oder Löschen werden **Referenzen**, **Pointer** und **Iteratoren** auf Elemente der Liste **nicht ungültig** (außer Verweise auf die gelöschten Elemente)

| Funktion | Beschreibung |
|---|---|
| `void push_back(const T& x);` | Einfügen des Objekts (Kopie) `x` als neues letztes Element |
| `void push_front(const T& x);` | Einfügen des Objekts (Kopie) `x` als neues erstes Element |
| `void pop_back();` | Löschen des letzten Elements |
| `void pop_front();` | Löschen des ersten Elements |
| `iterator insert(iterator pos, const T& x);` | Einfügen des Objekts `x` als neues Element an der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um eine Position weitergeschoben)<br>Rückgabe : die Einfügeposition `pos` |
| `void insert(iterator pos, size_type n, const T& x);` | Einfügen von `n` Kopien des Objekts `x` als neue Elemente ab der Position `pos`<br>(das bisherige Element an dieser Position und alle folgenden Elemente werden um `n` Positionen weitergeschoben) |
| `template <class InputIterator> void insert(iterator position, InputIterator first, InputIterator last);` | Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers befindlichen Elemente ab der Position `pos`.<br>(das bisherige Element an dieser Position und alle folgenden Elemente werden um entsprechend viele Positionen weitergeschoben) |
| `void remove(const T& val);` | Löschen aller Elemente, die den Wert `val` besitzen |
| `template <class Predicate> void remove_if(Predicate pred);` | Löschen aller Elemente, für die das als Parameter übergebene Funktionsobjekt `pred` den Wert `true` liefert |
| `void unique();` | Löschen aller Elemente jeder Gruppe aufeinanderfolgender gleicher Elemente außer dem jeweils ersten. |
| `template <class BinaryPredicate> void unique(BinaryPredicate binpred);` | Löschen aller direkten Nachfolger-Elemente eines Elements, für die das als Parameter übergebene Funktionsobjekt `binpred` den Wert `true` liefert.<br>`binpred` muss ein zweistelliges Funktionsobjekt sein, das auf jedes Element und sein jeweiliges direktes Nachfolger-Element angewendet wird. |

# Standard-Template-Library (STL) von C++ : Klassen-Template `list` (3)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, weitere Forts.)**

  ◇ **Weitere Methoden zur Modifikation von Listen**

| Memberfunktion | Beschreibung |
|---|---|
| `void splice(iterator pos, list<T, Allocator>& x);` | Einfügen aller Elemente der Liste `x` in die aktuelle Liste vor die Position `pos`.<br>Aus der Liste `x` werden alle Elemente entfernt |
| `void splice(iterator pos, list<T, Allocator>& x,`<br>`iterator i);` | Einfügen des durch den Iterator `i` referierten Elements der Liste `x` in die aktuelle Liste vor die Position `pos`.<br>Das Element wird aus der Liste `x` entfernt<br>Die Liste `x` darf mit der akt. Liste identisch sein. |
| `void splice(iterator pos, list<T, Allocator>& x,`<br>`iterator first, iterator last);` | Einfügen der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente der Liste `x` in die aktuelle Liste vor die Position `pos`.<br>Die eingefügten Elemente werden aus der Liste `x` entfernt.<br>Die Liste `x` darf mit der akt. Liste identisch sein. Falls in diesem Fall `pos` zwischen `first` und `last` liegt, ist das Verhalten undefiniert |
| `void merge(list<T, Allocator>& x);` | Einfügen aller Elemente der Liste `x` in die akt. Liste unter Beibehaltung einer Sortierreihenfolge.<br>Beide Listen müssen nach dieser Reihenfolge sortiert sein.<br>Aus der Liste `x` werden alle Elemente entfernt |
| `template <class Compare>`<br>`void merge(list<T, Allocator>& x, Compare comp);` | Einfügen aller Elemente der Liste `x` in die akt. Liste unter Beibehaltung der durch den Parameter `comp` bestimmten Sortierreihenfolge.<br>Beide Listen müssen nach dieser Reihenfolge sortiert sein.<br>Aus der Liste `x` werden alle Elemente entfernt |
| `void sort();` | Sortieren der akt. Liste unter Verwendung von `operator<()`.<br>Diese Operatorfunktion muss für den Typ `T` definiert sein |
| `template <class Compare>`<br>`void sort(Compare comp);` | Sortieren der akt. Liste unter Verwendung des Funktionsobjekts `comp` |
| `void reverse();` | Invertierung der Reihenfolge der Listenelemente |

# Standard-Template-Library (STL) von C++ : Klassen-Template `list` (4)

- **Einfaches Demonstrationsprogramm `listdem`**

```
// C++-Quelldatei listdem_m.cpp --> Programm listdem
// Einfaches Demo-Programm zum Klassen-Template list<> der STL

#include <list>
#include <iostream>
#include <cstdlib>
using namespace std;

template <class T>
  void showSizeData(const list<T>& lst, ostream& out)
  { out << "size()     : " << lst.size() << endl;
    out << "max_size() : " << lst.max_size() << endl;
  }

template <class T>
  void showContent(list<T>& lst, ostream& out)
  { typename list<T>::iterator it=lst.begin();
    if (it==lst.end()) out << "leer";
    else
      for (; it!=lst.end(); ++it)
        out << *it << "  ";
    out << endl;
  }

int main(void)
{
  list<int> il1, il2, il3;
  cout << "\nleere int-Liste :\n"; showSizeData(il1, cout);
  for (int i=0; i<10; i++)
  { il1.push_back(rand()%11+1);
    il2.push_front(i+1);
    il3.push_back(rand()%12);
  }
  cout << "\nInhalt Liste 1 :\n"; showContent(il1, cout);
  cout << "\nInhalt Liste 2 :\n"; showContent(il2, cout);
  cout << "\nInhalt Liste 3 :\n"; showContent(il3, cout);
  il1.splice(il1.begin(), il3);
  cout << "\nInhalt Liste 1 nach il1.splice(il1.begin(), il3) :\n";
  showContent(il1, cout);
  cout << "\nInhalt Liste 3 nach il1.splice(il1.begin(), il3) :\n";
  showContent(il3, cout);
  il1.sort();
  il2.reverse();
  cout << "\nInhalt Liste 1 nach sort() :\n";
  showContent(il1, cout);
  cout << "\nInhalt Liste 2 nach reverse() :\n";
  showContent(il2, cout);
  il1.merge(il2);
  cout << "\nInhalt Liste 1 nach il1.merge(il2) :\n";
  showContent(il1, cout);
  cout << "\nInhalt Liste 2 nach il1.merge(il2) :\n";
  showContent(il2, cout);
  il1.unique();
  cout << "\nInhalt Liste 1 nach unique() :\n";
  showContent(il1, cout);
  il1.remove_if(bind2nd(not_equal_to<int>(), 2));
  cout << "\nInhalt Liste 1 nach remove_if(...) :\n";
  showContent(il1, cout);
  return 0;
}
```

# Standard-Template-Library (STL) von C++ : Klassen-Template `list` (5)

- **Ausgabe des Demonstrationsprogramms `listdem`**

```
leere int-Liste :
size()      : 0
max_size() : 1073741823 (3fffffff)

Inhalt Liste 1 :
9 10 8 6 2 8 6 7 4 10

Inhalt Liste 2 :
10 9 8 7 6 5 4 3 2 1

Inhalt Liste 3 :
11 4 4 6 8 5 3 11 2 0

Inhalt Liste 1 nach il1.splice(il1.begin(), il3) :
11 4 4 6 8 5 3 11 2 0 9 10 8 6 2 8 6 7 4 10

Inhalt Liste 3 nach il1.splice(il1.begin(), il3) :
leer

Inhalt Liste 1 nach sort() :
0 2 2 3 4 4 4 5 6 6 6 7 8 8 8 9 10 10 11 11

Inhalt Liste 2 nach reverse() :
1 2 3 4 5 6 7 8 9 10

Inhalt Liste 1 nach il1.merge(il2) :
0 1 2 2 2 3 3 4 4 4 4 5 5 6 6 6 6 7 7 8 8 8 8 9 9 10 10 10 11 11

Inhalt Liste 2 nach il1.merge(il2) :
leer

Inhalt Liste 1 nach unique() :
0 1 2 3 4 5 6 7 8 9 10 11

Inhalt Liste 1 nach remove_if(...) :
2
```

# Standard-Template-Library (STL) von C++ : Container-Adapter (1)

- **Klassen-Template `stack<>`**

  ◇ Implementiert die Funktionalität eines **Stacks** (Kellerspeicher, **LIFO**) unter Verwendung der Container-Klassen `deque` (Default), `vector` oder `list`.

  ◇ Die **Definition** befindet sich in der Headerdatei **`<stack>`**

```
template <class T, class Container = deque<T> >
  class stack
  {
    public :
      typedef typename Container::value_type   value_type;
      typedef typename Container::size_type    size_type;
      typedef Container                        container_type;
    protected :
      Container c;
    public :
      explicit stack(const Container& co = Container()) : c(co) {}
      bool              empty() const             { return c.empty(); };
      size_type         size() const              { return c.size();  }
      value_type&       top()                     { return c.back();  }
      const value_type& top() const               { return c.back();  }
      void              push(const value_type& x) { c.push_back(x);   }
      void              pop()                     { c.pop_back();     }
  };
```

- **Klassen-Template `queue<>`**

  ◇ Implementiert die Funktionalität eines **Pufferspeichers** (**FIFO**) unter Verwendung der Container-Klassen `deque` (Default) oder `list`.

  ◇ Die **Definition** befindet sich in der Headerdatei **`<queue>`**

```
template <class T, class Container = deque<T> >
  class queue
  {
    public :
      typedef typename Container::value_type   value_type;
      typedef typename Container::size_type    size_type;
      typedef Container                        container_type;
    protected :
      Container c;
    public :
      explicit queue(const Container& co = Container()) : c(co) {}
      bool              empty() const             { return c.empty(); };
      size_type         size() const              { return c.size();  }
      value_type&       front()                   { return c.front(); }
      const value_type& front() const             { return c.front(); }
      value_type&       back()                    { return c.back();  }
      const value_type& back() const              { return c.back();  }
      void              push(const value_type& x) { c.push_back(x);   }
      void              pop()                     { c.pop_front();    }
  };
```

# Standard-Template-Library (STL) von C++ : Container-Adapter (2)

- **Klassen-Template `priority_queue<>`**

  ◇ Implementiert einen **Pufferspeicher**, bei der die Elemente nach einer durch ein **Sortierkriterium** festgelegten "Priorität" wieder ausgelesen werden können. Es wird immer das Element mit der **höchsten Priorität** zurückgeliefert.

  ◇ Das Sortierkriterium kann als Template-Parameter übergeben werden. **Default** ist die Funktionsobjekt-Klasse **`less<T>`**, d. h. die **höchste Priorität** hat das **größte** Element.

  ◇ Priority-Queues verwenden zur Speicherung ihrer Elemente die Container-Klasse `vector` (Default) oder `deque`.

  ◇ Für Priority-Queues sind **keine Vergleichsoperatoren** definiert.

  ◇ Die **Definition** befindet sich in der Headerdatei **`<queue>`**

```
template <class T, class Container = vector<T>,
          class Compare = less<typename Container::value_type> >
  class priority_queue
  {
    public :
      typedef typename Container::value_type   value_type;
      typedef typename Container::size_type    size_type;
      typedef Container                        container_type;
    protected :
      Container c;
      Compare comp;
    public :
      explicit priority_queue(const Compare& x = Compare(),
                              const Container& co = Container())
                             : c(co), comp(x) {}

      template <class InputIterator>
        priority_queue (InputIterator first, InputIterator last,
                        const Compare& x = Compare(),
                        const Container& co = Container())
                       : c(co), comp(x)
       { c.insert(c.end(),first,last);
         make_heap(c.begin(), c.end(), comp);
       }

      bool               empty() const             { return c.empty(); };
      size_type          size() const              { return c.size();  }
      const value_type&  top() const               { return c.front(); }

      void push(const value_type& x)
      { c.push_back(x);
        push_heap(c.begin(), c.end(), comp);
      }

      void pop()
      { pop_heap(c.begin(), c.end(), comp);
        c.pop_back();
      }
  };
```

  ◇ Die zur **Implementierung** der Priority-Queue verwendeten freien Funktionen **`make_heap()`**, **`push_heap()`** und **`pop_heap()`** sind Funktionen aus der **Algorithmen-Bibliothek** der STL.
  Sie stellen sicher, dass der Container als "**Heap**" verwaltet wird, d.h. das sein erstes Element immer das Element mit der **"höchsten" Erfüllung** des **Vergleichskriteriums** ist.

# Standard-Template-Library (STL) von C++ : Container-Adapter (3)

- **Einfaches Demonstrationsprogramm `priqueuedem` zum Container-Adapter `priority_queue`**

```
// C++-Quelldatei priqueuedem_m.cpp --> Programm priqueuedem
// Einfaches Demo-Programm zum Container-Adapter priority_queue<> der STL

#include <queue>
#include <iostream>
using namespace std;

int main(void)
{
  priority_queue<double> pqd;
  double w;
  cout << "\nleere double-Priority-Queue :\n";
  cout << "size() : " << pqd.size() << endl;
  cout << "\nAblage der folgenden Elemente :\n";
  pqd.push(w=27.33); cout << w << endl;
  pqd.push(w=63.12); cout << w << endl;
  pqd.push(w=12.84); cout << w << endl;
  pqd.push(w=21.37); cout << w << endl;
  cout << "\nEntfernen der beiden \"obersten\" Elemente :\n";
  cout << pqd.top() << endl;
  pqd.pop();
  cout << pqd.top() << endl;
  pqd.pop();
  cout << "\nAblage eines weiteren Elements :\n";
  pqd.push(w=99.88); cout << w << endl;
  cout << "\nEntfernen der beiden \"obersten\" Elemente :\n";
  cout << pqd.top() << endl;
  pqd.pop();
  cout << pqd.top() << endl;
  pqd.pop();
  cout << "\nverbleibende Anzahl von Elementen :\n";
  cout << "size() : " << pqd.size() << endl;
  return 0;
}
```

- **Ausgabe des Demonstrationsprogramm `priqueuedem`**

```
leere double-Priority-Queue :
size() : 0

Ablage der folgenden Elemente :
27.33
63.12
12.84
21.37

Entfernen der beiden "obersten" Elemente :
63.12
27.33

Ablage eines weiteren Elements :
99.88

Entfernen der beiden "obersten" Elemente :
99.88
21.37

verbleibende Anzahl von Elementen :
size() : 1
```

# Standard-Template-Library (STL) von C++ : Klassen-Template `set` (1)

## • Eigenschaften

◇ Implementierung von **Sets** (Set-Containern)
Ein Set (-Container) verwaltet nach ihrem "Wert" **sortierte Elemente** – typischerweise in einem balancierten Binärbaum. Das Sortierkriterium (Funktionsobjekt-Klasse) ist Template-Parameter (default : `less<>`)
Jedes Element darf **nur einmal** in einem Set vorkommen.

◇ Aufgrund der automatischen Sortierung in einem Binärbaum ermöglichen Sets ein **schnelles Suchen** und **Finden** von Elementen. Suchschlüssel sind die Elemente selbst.
Auch das **Einfügen** u/o. **Löschen** von Elementen besitzt an **allen Stellen** ein **gutes Zeitverhalten**.

◇ Die Elemente von Sets können auch **sequentiell durchlaufen** werden.
Ein direkter **wahlfreier Zugriff** zu einzelnen Elementen ist jedoch **nicht möglich**. Indexoperator und Direktzugriffs-Iteratoren sind daher nicht implementiert.
Beim Zugriff zu Elementen über Iteratoren werden die Elemente als Konstante betrachtet.

◇ Die Elemente können **nicht direkt geändert** werden. Eine Änderung könnte die Sortierung ungültig machen. Um ein Element zu ändern, muß es daher aus dem Set entfernt werden und nach der Änderung als neues Element wieder eingefügt werden.

◇ Die **Definition** des Klassen-Templates `set<>` befindet sich in der **Headerdatei** **`<set>`**

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> >
  class set
  { // ...
  };
```

## • Konstruktoren

| | |
|---|---|
| `explicit set(const Compare& = Compare(), const Allocator& = Allocator());` | Erzeugung eines leeren Sets |
| `template <class InputIterator> set(InputIterator first, InputIterator last, const Compare& =Compare(), const Allocator& = Allocator());` | Erzeugung eines zunächst leeren Sets, in das dann die zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente eines anderen Containers eingefügt werden |
| `set(const set<Key, Compare, Allocator>& x);` | Copy-Konstruktor |

## • Zusätzliche spezifische Memberfunktionen (Auswahl)

### ◇ Zusätzliche Methode zum Löschen eines Elements

| | |
|---|---|
| `size_type erase(const Key& x);` | Löschen des Elements `x`<br>Rückgabewert = - 1, falls gelöscht wurde (Element war enthalten)<br>- 0, falls nicht gelöscht wurde (Element war nicht enthalten) |

# Standard-Template-Library (STL) von C++ : Klassen-Template `set` (2)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)**

◇ **Methoden zum Einfügen von Elementen**

| | |
|---|---|
| `pair<iterator, bool> insert(const Key& x);` | Einfügen des Objekts `x` als neues Element, falls es noch nicht im Set vorhanden ist.<br>**Rückgabewert** : Wertepaar mit Einfügeposition (Komponente `first`) und Einfügeerfolg (Komponente `second`)<br>Einfügeerfolg = `true`, wenn eingefügt wurde |
| `iterator insert(iterator pos, const Key& x);` | Einfügen des Objekts `x` als neues Element, falls es noch nicht im Set vorhanden ist.<br>Der Parameter `pos` dient als ein Hinweis, wo im Set mit der Suche nach der Einfügeposition begonnen werden sollte.<br>**Rückgabewert** : Einfügeposition bzw Position, an der sich das Element bereits befindet. |
| `template <class InputIterator> void insert(InputIterator first, InputIterator last);` | Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers befindlichen Elemente.<br>Ein Element wird nur eingefügt, wenn es noch nicht im Set vorhanden ist. |

◇ **Methoden zum Suchen**

| | |
|---|---|
| `iterator find(const Key& x) const;` | Suchen des Elements, das gleich dem Objekt `x` ist.<br>**Rückgabewert** : - Position von `x`, falls vorhanden<br>- `end()`, falls nicht vorhanden |
| `size_type count(const Key& x) const;` | Ermittlung, ob das Objekt `x` im Set enthalten ist<br>**Rückgabewert** : - `1`, falls vorhanden<br>- `0`, falls nicht vorhanden |
| `iterator lower_bound(const Key& x) const;` | Rückgabe der Position des ersten Elements im Set, das nicht kleiner als `x` (d.h. größer gleich `x`) ist |
| `iterator upper_bound(const Key& x) const;` | Rückgabe der Position des ersten Elements im Set, das größer als `x` ist |
| `pair<iterator, iterator> equal_range(const Key& x) const;` | Rückgabe eines `pair`-Objekts aus `lower_bound(x)` und `upper_bound(x)`.<br>`lower_bound(x)` und `upper_bound(x)` sind nur dann verschieden, wenn das Objekt `x` im Set enthalten ist |

# Standard-Template-Library (STL) von C++ : Klassen-Template set (3)

- **Einfaches Demonstrationsprogramm setdem**

```
// C++-Quelldatei setdem_m.cpp --> Programm setdem
// Einfaches Demo-Programm zum Klassen-Template set<> der STL

#include <set>
#include <iostream>
#include <string>
using namespace std;

typedef set<string, greater<string> > StringDownSet;
typedef set<string>                   StringSet;

template <class SetType>
void einfuegen(SetType& sds, const string& str)
{ pair<typename SetType::iterator, bool> success = sds.insert(str);
  cout << "\"" << str << "\"";
  if (success.second) cout << " erfolgreich eingefuegt\n";
  else cout << " bereits vorhanden\n";
}

ostream& operator<<(ostream& out, StringSet& menge)
{ StringSet::iterator loc;
  for (loc=menge.begin(); loc!=menge.end(); ++loc)
    out << *loc << ' ';
  return out << endl;
}


int main(void)
{ StringDownSet wmenge1;
  wmenge1.insert("man");
  wmenge1.insert("politikern");
  wmenge1.insert("kann");
  wmenge1.insert("tauben");
  wmenge1.insert("zutrauen");
  wmenge1.insert("sowie");
  einfuegen(wmenge1, "kann");
  einfuegen(wmenge1, "wesentliches");
  cout << endl;
  StringDownSet::iterator pos;
  for (pos=wmenge1.begin(); pos!=wmenge1.end(); ++pos)
    cout << *pos << ' ';
  cout << endl;

  #ifndef _MSC_VER                // fuer ANSI-C++ kompatible Compiler
  StringSet wmenge2(wmenge1.begin(), wmenge1.end());
  #else                           // Alternative fuer Visual-C++
  StringSet wmenge2;
  for (pos=wmenge1.begin(); pos!=wmenge1.end(); ++pos)
    wmenge2.insert(*pos);
  #endif

  cout << wmenge2;
  wmenge2.erase(wmenge2.find("sowie"), wmenge2.find("wesentliches"));
  cout << wmenge2;
  wmenge2.erase(wmenge2.begin(), wmenge2.find("politikern"));
  cout << wmenge2 << endl;
  einfuegen(wmenge2, "niemals");
  cout << endl;
  cout << wmenge2;
  return 0;
}
```

# Standard-Template-Library (STL) von C++ : Klassen-Template `set` (4)

- **Ausgabe des Demonstrationsprogramms `setdem`**

```
"kann" bereits vorhanden
"wesentliches" erfolgreich eingefuegt

zutrauen wesentliches tauben sowie politikern man kann
kann man politikern sowie tauben wesentliches zutrauen
kann man politikern wesentliches zutrauen
politikern wesentliches zutrauen

"niemals" erfolgreich eingefuegt

niemals politikern wesentliches zutrauen
```

# Standard-Template-Library (STL) von C++ : Klassen-Template **multiset**

## • Eigenschaften

◇ Implementierung von **Multisets** (Multiset-Containern)
Ein Multiset (-Container) entspricht einem Set mit dem Unterschied, dass Elemente auch mehrfach enthalten sein können.

◇ Die **Definition** des Klassen-Templates `multiset<>` befindet sich ebenfalls in der **Headerdatei** **`<set>`**

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> >
  class multiset
  { // ...
  };
```

## • Konstruktoren

| | |
|---|---|
| `explicit multiset(const Compare& = Compare(), const Allocator& = Allocator());` | Erzeugung eines leeren Multisets |
| `template <class InputIterator> multiset(InputIterator first, InputIterator last, const Compare& =Compare(), const Allocator& = Allocator());` | Erzeugung eines zunächst leeren Multisets, in das dann die zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente eines anderen Containers eingefügt werden |
| `multiset(const multiset<Key, Compare, Allocator>& x);` | Copy-Konstruktor |

## • Anmerkungen zu den zusätzlichen spezifischen Memberfunktionen

◇ Das Klassen-Template `multiset` besitzt die **gleichem Memberfunktionen** wie das Klassen-Template **`set`**.
Sie **unterscheiden** sich lediglich durch die **Auswirkungen** des möglichen **Mehrfachenthaltenseins** von Elementen.

◇ Im wesentlichen bestehen die **folgenden Unterschiede** :

- **`erase`**`(x)` Löschen aller Elemente, die gleich dem Objekt `x` sind.
- **`insert`**`(x)` Der Rückgabewert ist vom Typ `iterator` (statt `pair<...>`)
  Einfügen des Objekts `x` als neues Element auch dann, wenn es bereits im Multiset enthalten ist. Es wird immer die Einfügeposition zurückgegeben.
- **`insert`**`(pos, x)` Einfügen des Objekts `x` als neues Element auch dann, wenn es bereits im Multiset enthalten ist. Der Rückgabewert ist immer die Einfügeposition
- **`insert`**`(first, last)` Einfügen immer aller Elemente aus dem Bereich `first` (einschl.) bis `last` (ausschl.), auch die, die bereits vorhanden sind.
- **`count`**`(x)` Rückgabe der Anzahl Elemente, die gleich dem Objekt `x` sind
- **`find`**`(x)` Suchen des ersten Elements, das gleich dem Objekt `x` ist.

# Standard-Template-Library (STL) von C++ : Klassen-Template `map` (1)

## • Eigenschaften

◇ Implementierung von **Maps** (Map-Containern)
Ein(e) Map (-Container) speichert **Schlüssel-/Werte-Paare** als Elemente (Klasse **`pair<const Key, T>`**).
Die Elemente sind nach dem **(Such-)Schlüssel** (Klasse `Key`) **sortiert**. Der Suchschlüssel dient zum Auffinden des mit ihm assoziierten Werts (das eigentliche verwaltete Objekt, Klasse `T`). Jeder **Suchschlüssel** darf **nur einmal** in einer Map vorkommen, d.h. mit einem Suchschlüssel kann nur ein einziger Wert assoziiert sein ("***one-to-one mapping***")

◇ Auch eine Map verwaltet ihre Elemente typischerweise in einem balancierten Binärbaum. Das Sortierkriterium (Funktionsobjekt-Klasse) ist ebenfalls Template-Parameter (default : `less<>`)

◇ Der **Schlüssel** eines Elements darf **nicht geändert** werden, wohl **aber** sein **Wert**.

◇ Die übrigen Eigenschaften (bezüglich Zeitverhalten bei Suchen/Finden und Einfügen/Löschen sowie den Zugriffsmöglichkeiten zu den Elementen) entsprechen denen eines Sets, mit dem Unterschied, dass die enthaltenen Elemente Paare sind.

◇ Die **Definition** des Klassen-Templates `map<>` befindet sich in der **Headerdatei** **`<map>`**

```
template <class Key, class T,
          class Compare = less<Key>,
          class Allocator = allocator<pair<const Key,T> > >
  class map
  { // ...
  };
```

## • Konstruktoren

| | |
|---|---|
| `explicit map(const Compare& = Compare(), const Allocator& = Allocator());` | Erzeugung einer leeren Map |
| `template <class InputIterator> map(InputIterator first, InputIterator last, const Compare& =Compare(), const Allocator& = Allocator());` | Erzeugung einer zunächst leeren Map, in die dann die zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente eines anderen Containers eingefügt werden Diese Elemente müssen vom Typ `pair<const Key, T>` sein. |
| `map(const map<Key, T, Compare, Allocator>& x);` | Copy-Konstruktor |

## • Zusätzliche spezifische Memberfunktionen (Auswahl)

### ◇ Zusätzliche Methode zum Löschen eines Elements

| | |
|---|---|
| `size_type erase(const Key& k);` | Löschen des Elements (Schlüssel-/Wert-Paar) mit dem Schlüssel `k`<br>Rückgabewert = - 1, falls gelöscht wurde (Element mit Schlüssel `k` war enthalten)<br>- 0, falls nicht gelöscht wurde (kein Element mit Schlüssel `k` enthalten.) |

# Standard-Template-Library (STL) von C++ : Klassen-Template `map` (2)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)**

◇ **Methoden zum Einfügen von Elementen**

| | |
|---|---|
| `pair<iterator, bool>`<br>`  insert(pair<const Key, T>& x);` | Einfügen des Schlüssel-/Werte-Paares `x` als neues Element, falls noch kein Element mit dem Schlüssel `x.first` in der Map vorhanden ist.<br>**Rückgabewert** : Wertepaar mit Einfügeposition (Komponente `first`) und Einfügeerfolg (Komponente `second`).<br>Einfügeerfolg = `true`, wenn eingefügt wurde |
| `iterator insert(iterator pos,`<br>`                pair<const Key, T>& x);` | Einfügen des Schlüssel-/Werte-Paares `x` als neues Element, falls noch kein Element mit dem Schlüssel `x.first` in der Map vorhanden ist.<br>Der Parameter `pos` dient als Hinweis, wo in der Map mit der Suche nach der Einfügeposition begonnen werden sollte.<br>**Rückgabewert** : Einfügeposition bzw Position, an der sich ein Element mit dem Schlüssel bereits befindet. |
| `template <class InputIterator>`<br>`  void insert(InputIterator first,`<br>`              InputIterator last);` | Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers (i. a. einer Map oder Multimap) befindlichen Elemente.<br>Ein Element wird nur eingefügt, wenn es noch kein Element mit seinem Schlüssel in der Map gibt. |

◇ **Methoden zum Suchen**

| | |
|---|---|
| `iterator find(const Key& k);`<br>`const_iterator find(const Key& k) const;` | Suchen des Elements, das den Schlüssel `k` hat.<br>**Rückgabewert** : - Position des Elements, falls vorhanden<br>- `end()`, falls nicht vorhanden |
| `size_type count(const Key& k) const;` | Ermittlung, ob ein Element mit dem Schlüssel `k` in der Map enthalten ist<br>**Rückgabewert** : - `1`, falls vorhanden<br>- `0`, falls nicht vorhanden |
| `iterator lower_bound(const Key& k);`<br>`const_iterator lower_bound(const Key& k) const;` | Rückgabe der Position des ersten Elements in der Map, dessen Schlüssel nicht kleiner als `k` (d.h. größer gleich `k`) ist |
| `iterator upper_bound(const Key& k);`<br>`const_iterator upper_bound(const Key& k) const;` | Rückgabe der Position des ersten Elements in der Map, dessen Schlüssel größer als `k` ist |
| `pair<iterator, iterator>`<br>`  equal_range(const Key& k);`<br><br>`pair<const_iterator, const_iterator>`<br>`  equal_range(const Key& k) const;` | Rückgabe eines `pair`–Objekts aus `lower_bound(k)` und `upper_bound(k)`.<br>`lower_bound(k)` und `upper_bound(k)` sind nur dann verschieden, wenn ein Element mit dem Schlüssel `k` in der Map enthalten ist |

# Standard-Template-Library (STL) von C++ : Klassen-Template map (3)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, weitere Forts.)**

◇ **Methode zum direkten Elementzugriff (Indexoperator)**

| | |
|---|---|
| `T& operator[](const Key& k);` | Ermittlung der Wert-Komponente des Elements mit dem Schlüssel `k`. Falls kein Element mit dem Schlüssel `k` existiert, wird ein neues Element angelegt (dessen Wert-Komponente mit ihrem Default-Konstruktor initialisiert wird). |

Diese Methode macht eine Map zur Implementierung eines **assoziativen Arrays**.
Die Auswahl eines "Array"-Elements erfolgt über einen Teil seines Inhalts (hier den Schlüssel `k`).
Der Auswahl-"Index" kann damit von einem beliebigen (und nicht nur einem ganzzahligen) Typ sein.

Interessant ist, dass es **keinen unerlaubten** "Wert" für den **Index** gibt.
Existiert kein Element für den Index, wird ein neues Element erzeugt.

**Beispiel** :

```
// ...
map<string, float> preisliste;   // leere Map
preisliste["Uhr"]=24.50;
// ...
```

→ Es wird ein neues Element mit dem Schlüssel `"Uhr"` angelegt.
Eine Referenz auf die Wert-Komponente diese Elements, die zunächst undefiniert ist, wird von der Index-Operatorfunktion zurückgegeben.
Anschließend wird dieser Komponente der Wert `24.50` zugewiesen.

- **Einige innere Datentypen**

◇ Innerhalb des Klassen-Templates `map` sind – wie in allen Container-Klassen – mehrere Datentypen definiert.
Es handelt sich bei ihnen um `public`–Komponenten

◇ Einige dieser Datentypen sind :

```
template <class Key, class T,
          class Compare = less<Key>,
          class Allocator = allocator<pair<const Key,T> > >
  class map
  { public :
      typedef Key                   key_type;
      typedef T                     mapped_type;
      typedef pair<const Key, T>    value_type;
      // ...
  };
```

- **Möglichkeiten zur Erzeugung von Schlüssel-/Werte-Paaren (z.B. als Parameter für `insert(...)`)**

◇ mit **`pair<>`** : z.B. `pair<const string, double>("Fahrrad", 259.00);`
Vereinfachung : Definition eines eigenen Typnamens für den `pair`–Typ.
z.B. `typedef pair<const string, double> StrDoubPair;`
`StrDoubPair("Fahrrad", 259.00);`

◇ mit **`value_type`** : z.B. `map<string, double>::value_type("Fahrrad", 259.00);`

◇ mit **`make_pair()`** : z.B. `make_pair(string("Fahrrad"), 259.00);`
Achtung : Die Typen von `pair` müssen eindeutig aus den aktuellen Parametern von `make_pair()` erkennbar sein

# Standard-Template-Library (STL) von C++ : Klassen-Template `multimap` (1)

- **Eigenschaften**

  ◇ Implementierung von **Multimaps** (Multimap-Containern)
  Ein Multimap (-Container) entspricht einer Map mit dem Unterschied, dass mehrere Elemente mit dem gleichen Schlüssel enthalten sein können ("***one-to-many mapping***")

  ◇ Die **Definition** des Klassen-Templates `multimap<>` befindet sich ebenfalls in der **Headerdatei** **`<map>`**

```
template <class Key, class T,
          class Compare = less<Key>,
          class Allocator = allocator<pair<const Key,T> > >
  class multimap
  { // ...
  };
```

- **Konstruktoren**

| | |
|---|---|
| `explicit multimap(const Compare& = Compare(), const Allocator& = Allocator());` | Erzeugung einer leeren Multimap |
| `template <class InputIterator> multimap(InputIterator first, InputIterator last, const Compare& =Compare(), const Allocator& = Allocator());` | Erzeugung einer zunächst leeren Multimap, in die dann die zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente eines anderen Containers eingefügt werden. Diese Elemente müssen vom Typ `pair<const Key, T>` sein. |
| `multimap(const multimap<Key, T, Compare, Allocator>& x);` | Copy-Konstruktor |

- **Anmerkungen zu den zusätzlichen spezifischen Memberfunktionen**

  ◇ Die Indexoperatorfunktion **`operator[]()`** ist für das Klassen-Template `multimap` **nicht definiert**.
  Da mehrere Elemente mit dem gleichen Schlüssel vorkommen können, kann dieser nicht zur Elementauswahl verwendet werden. → Multimaps eignen sich nicht als assoziative Arrays.

  ◇ Im übrigen besitzt das Klassen-Template `multimap` die **gleichen Memberfunktionen** wie **`map`**.
  Sie **unterscheiden** sich lediglich durch die **Auswirkungen** des möglichen **Mehrfachenthaltenseins** von Schlüsseln.

  ◇ Im wesentlichen bestehen die **folgenden Unterschiede** :

  ▻ **`erase`**`(k)` Löschen aller Elemente, die dem Schlüssel `k` besitzen.

  ▻ **`insert`**`(x)` Der Rückgabewert ist vom Typ `iterator` (statt `pair<...>`)
  Einfügen des Schlüssel-/Wert-Paares `x` als neues Element auch dann, wenn bereits ein Element mit dem Schlüssel `x.first` in der Multimap enthalten ist.
  Es wird immer die Einfügeposition zurückgegeben.

  ▻ **`insert`**`(pos, x)` Einfügen des Schlüssel-/Wert-Paares `x` als neues Element auch dann, wenn bereits ein Element mit dem Schlüssel `x.first` in der Multimap enthalten ist.
  Der Rückgabewert ist immer die Einfügeposition

  ▻ **`insert`**`(first, last)` Einfügen immer aller Elemente aus dem Bereich `first` (einschl.) bis `last` (ausschl.), auch die, deren Schlüssel bereits vorhanden sind.

  ▻ **`count`**`(k)` Rückgabe der Anzahl Elemente, die den Schlüssel `k` besitzen.

  ▻ **`find`**`(k)` Suchen des ersten Elements, dessen Schlüssel gleich `k` ist.

# Standard-Template-Library (STL) von C++ : Klassen-Template `multimap` (2)

- **Einfaches Demonstrationsprogramm `multimapdem`**

```
// C++-Quelldatei multimapdem_m.cpp --> Programm multimapdem
// Einfaches Demo-Programm zum Klassen-Template multimap<> der STL

#include <map>
#include <string>
#include <iostream>
#include <utility>
#include <iomanip>
using namespace std;

typedef multimap<string, string>   StrStrMMap;
typedef pair<const string, string> StringPaar;


int main(void)
{
  StrStrMMap dict;
  dict.insert(make_pair(string("clever"), string("klug")));
  dict.insert(pair<const string, string>("clever", "gewandt"));
  dict.insert(StrStrMMap::value_type("clever", "raffiniert"));
  dict.insert(StringPaar("smart", "klug"));
  dict.insert(StringPaar("smart", "gewandt"));
  dict.insert(StringPaar("smart", "elegant"));
  dict.insert(StringPaar("wise", "klug"));
  dict.insert(StringPaar("wise", "erfahren"));
  dict.insert(StringPaar("strange", "fremd"));
  dict.insert(StringPaar("strange", "seltsam"));
  dict.insert(StringPaar("odd", "seltsam"));
  dict.insert(StringPaar("odd", "ungerade"));
  dict.insert(StringPaar("quick", "schnell"));
  dict.insert(StringPaar("quick", "gewandt"));
  dict.insert(StringPaar("car", "Auto"));
  dict.insert(StringPaar("again", "nochmals"));
  dict.insert(StringPaar("ship", "Schiff"));

  // Ausgabe aller Eintraege (Schluessel-Werte-Paare)
  cout << "Enthalten sind " << dict.size() << " Eintraege\n" << left << endl;
  StrStrMMap::iterator pos;
  for (pos=dict.begin(); pos!=dict.end(); ++pos)
    cout << "english : " << setw(15) << pos->first
         << "deutsch : " << setw(15) << pos->second << endl;
  cout << right << endl;;

  // Ausgabe aller Werte zu einenm bestimmten Schluessel
  string wort("clever");
  cout << wort << " : " << dict.count(wort) << " Eintraege" << endl;
  for (pos=dict.lower_bound(wort);
       pos!=dict.upper_bound(wort) ; ++pos)
    cout << "    " << pos->second << endl;
  cout << endl;

  // Ausgabe aller Schluessel zu einem bestimmten Wert
  wort="klug";
  cout << wort << " : " <<endl;
  for (pos=dict.begin(); pos!=dict.end(); ++pos)
    if (pos->second==wort)
      cout << "    " << pos->first << endl;
  cout << endl;

  return 0;
}
```

# Standard-Template-Library (STL) von C++ : Klassen-Template `multimap` (3)

- **Ausgabe des Demonstrationsprogramms `multimapdem`**

```
Enthalten sind 17 Eintraege

english : again          deutsch : nochmals
english : car            deutsch : Auto
english : clever         deutsch : klug
english : clever         deutsch : gewandt
english : clever         deutsch : raffiniert
english : odd            deutsch : seltsam
english : odd            deutsch : ungerade
english : quick          deutsch : schnell
english : quick          deutsch : gewandt
english : ship           deutsch : Schiff
english : smart          deutsch : klug
english : smart          deutsch : gewandt
english : smart          deutsch : elegant
english : strange        deutsch : fremd
english : strange        deutsch : seltsam
english : wise           deutsch : klug
english : wise           deutsch : erfahren

clever : 3 Eintraege
   klug
   gewandt
   raffiniert

klug :
   clever
   smart
   wise
```

# Standard-Template-Library (STL) von C++ : Iteratoren (1)

- **Einführung**

◇ **Iteratoren** sind Objekte, die einen Mechanismus zum **Durchlaufen** von **Container-Objekten** zur Verfügung stellen.

◇ Die in der STL implementierten Iteratoren ermöglichen einen **zeigerähnlichen Zugriff** zu den einzelnen **Elementen** eines Containers. Analog zu einem Zeiger, der die Position eines Array-Elements angibt, **repräsentiert** ein derartiges **Iterator-Objekt** die **Position** eines **Container-Elements**.
Solche Iteratoren können somit als **Verallgemeinerung von Zeigern** aufgefasst werden.

◇ Da Iteratoren **Zustandsinformationen** des jeweiligen **Containers**, auf dem sie arbeiten, auswerten und die **Organisationsform** seiner Elemente berücksichtigen müssen, ist eine **Iterator-Klasse** immer an eine **bestimmte Container-Klasse gebunden**.
D. h. zu jeder Container-Klasse gehört jeweils eine **spezifische** passende **Iterator-Klasse**. Typischerweise ist diese als **innere Klasse** der Containerklasse definiert. Bei den Iteratoren der STL ist das immer der Fall.

◇ Alle **Iterator-Klassen** der STL stellen – unabhängig von der Container-Klasse, an die sie jeweils gebunden sind und unabhängig von ihrer eigenen Implementierung – eine **einheitliche Schnittstelle** zur Verfügung.
Diese wird durch **Operator-Funktionen** gebildet, die die für Zeiger anwendbaren Operationen implementieren (wie Dereferenzierung, Inkrementierung usw.).

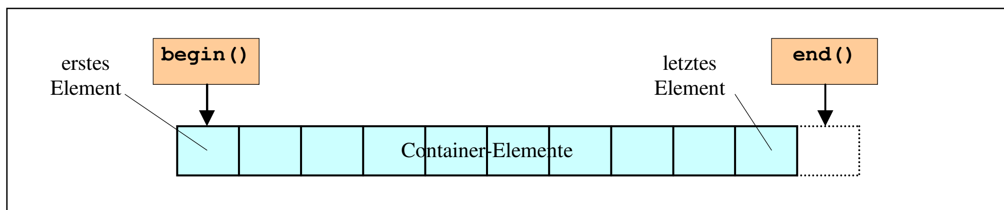
◇ Die **Container-Klassen** ihrerseits stellen eine **einheitliche Schnittstelle** zur **Bereitstellung** und **Verwendung** der **Iteratoren** bereit:

▻ In **jeder Container-Klasse** – außer den Container-Adaptern – sind die implementierungsabhängigen `public`-Typen **`iterator`** und **`const_iterator`** definiert.
Container-Elemente, die von Iterator-Objekten des Typs `const_iterator` referiert werden, können nicht geändert werden.

```
template < ... >
  class ...
  {
    public :
       // ...
       typedef implementation defined iterator;
       typedef implementation defined const_iterator;
       // ...
  };
```

▻ **Jede Container-Klasse** – außer den Container-Adaptern – stellt **Memberfunktionen** zur Ermittlung des **Iteratorbereichs** eines Container-Objekts zu Verfügung. :

- **`begin()`** liefert den Iterator, der auf das erste Element im Container zeigt
- **`end()`** liefert einen Iterator, der auf die Position unmittelbar nach dem letzten Element zeigt.




Der Bereich [ **`begin()`**, **`end()`** ) bildet also ein **halboffenes Intervall**, über das durch alle Container-Elemente iteriert werden kann.
Ist `begin()==end()`, ist das Intervall leer, d.h. der Container enthält keine Elemente.

**Beide Funktionen** existieren jeweils in **zwei überladenen Formen**, die eine liefert ein Iterator-Objekt vom Typ `iterator`, die andere ein Iterator-Objekt vom Typ `const_iterator`.

◇ Diese einheitlichen Schnittstellen ermöglichen es, dass die **Objekte** der **unterschiedlichen Container-Klassen**, die **unterschiedliche Datenstrukturen** implementieren, **gleichartig bearbeitet** werden können.

# Standard-Template-Library (STL) von C++ : Iteratoren (2)

## • Überblick über die Iterator-Bibliothek

◇ Die in der Iterator-Bibliothek enthaltenen Definitionen **iterator-spezifischer Datentypen** und **Funktionen** der STL sind in der Headerdatei **`<iterator>`** zusammengefasst.

◇ Im wesentlich handelt es sich hierbei um

▻ **grundlegende Datentypen**, die im Container-Teil der STL für die Definition der container-spezifischen Iterator-Klassen verwendet werden, sowie zur Definition eigener Iterator-Klassen eingesetzt werden können.
U.a. ist hier auch das als **Iterator-Basisklasse** dienende Klassen-Template **`iterator`** definiert

▻ **freie Iterator-Funktionen** (Funktions-Templates), die bestimmte Operationen auf Iterator-Objekte ausführen.

▻ **Iterator-Adapter**, durch die Iterator-Klassen mit spezifischen Eigenschaften vordefiniert werden.
Sie erweitern die Anwendungsmöglichkeiten der Algorithmen der STL erheblich.

Hierbei handelt es sich um

- **Reverse-Iteratoren**, mit denen die Durchlaufrichtung von Container umgekehrt wird.
- **Insert-Iteratoren**, mit denen sich die Kopier-Algorithmen der STL zum Einfügen (statt zum normalerweise stattfindenden Überschreiben) einsetzen lassen.
- **Stream-Iteratoren** und **Streambuffer-Iteratoren**, mit denen zu Eingabe- bzw. Ausgabe-Streams wie zu Containern zugegriffen werden kann.
  Das Lesen und Schreiben von Daten lässt sich damit unter Verwendung von Iteratoren durchführen.

◇ Zur Arbeit mit Iteratoren muss die Headerdatei `<iterator>` in der Regel aber nicht explizit eingebunden werden, da sie von allen Headerdateien für Container und der Headerdatei für Algorithmen bereits eingebunden wird.

## • Iterator-Kategorien

◇ In Abhängigkeit von den auf Iteratoren **anwendbaren Operationen** werden **fünf Iterator-Kategorien** unterschieden :

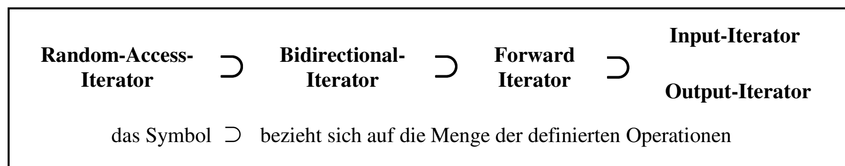
▻ Input-Iteratoren

▻ Output-Iteratoren

▻ Forward-Iteratoren

▻ Bidirectional-Iteratoren

▻ Random-Access-Iteratoren




◇ **Input-Iteratoren** (InpIter, *input iterators*)
Sie erlauben lediglich einen **lesenden Zugriff** auf das durch sie referierte Element des Containers, sowie ein Durchlaufen des Containers (der Sequenz) in Vorwärtsrichtung (Implementierung des Increment-Operators). Dabei ist mit einem Input-Iterator nur ein einmaliger Durchlauf möglich, d. h. er kann nur für *one-pass algorithms* eingesetzt werden
Außerdem lassen sich auf Input-Iteratoren der Zuweisungs-Operator, sowie der Gleichheits- und der Ungleichheits-Operator anwenden.

◇ **Output-Iteratoren** (OutpIter, *output iterators*)
Sie erlauben lediglich einen **schreibenden Zugriff** auf das durch sie referierte Container-Element, sowie ein Durchlaufen des Containers (der Sequenz) in Vorwärtsrichtung. Auch ein Output-Operator gestattet nur einen einmaligen Durchlauf.
Der Zuweisungs-Operator lässt sich auf sie anwenden, nicht jedoch der Gleichheits- und der Ungleichheits-Operator

◇ **Forward-Iteratoren** (ForwIter, *forward iterators*)
Sie kombinieren die Fähigkeiten von Input- und Output-Operatoren. Zusätzlich kann mit dem gleichen Iterator eine Elemente-Menge auch mehrfach durchlaufen werden, d.h. diese Iteratoren unterstützen *multi-pass algorithms*.

◇ **Bidirectional-Iteratoren** (BidirIter, *bidirectional iterators*)
Sie besitzen alle Fähigkeiten der Vorwärts-Iteratoren. Zusätzlich ermöglichen sie ein Durchlaufen des Containers in Rückwärtsrichtung (Implementierung des Decrement-Operators)

◇ **Random-Access-Iteratoren** (RanAccIter, *random access iterators*)
Sie besitzen alle Fähigkeiten der Bidirectional-Iteratoren. Zusätzlich erlauben sie einen direkten wahlfreien Zugriff zu jedem Element des Containers. Hierzu implementieren sie den Index-Operator sowie eine "Adress-Arithmetik".
Außerdem können auf diese Iteratoren auch die Vergleichs-Operationen `>`, `>=`, `<`, `<=` angewendet werden.
Damit stellen sie die **volle gleiche Funktionalität** wie normale typgebundene **Pointer** zu Verfügung.

# Standard-Template-Library (STL) von C++ : Iteratoren (3)

## • Überblick über die Iterator-Operationen

◇ In der folgenden Übersicht
- sind `p` und `q` Iteratoren
- bedeutet `X` : die Operation steht für die jeweilige Iterator-Kategorie zur Verfügung

| Operation | Bemerkung | InpIter | OutpIter | ForwIter | BidirIter | RanAccIter |
|---|---|---|---|---|---|---|
| **Kopieren** | | | | | | |
| p=q | Zuweisung | X | X | X | X | X |
| **Dereferenzieren** | | | | | | |
| x=*p | Lesen (Rvalue) | X | | X | X | X |
| x=p->k | x=(*p).k | X | | X | X | X |
| *p=x | Schreiben (Lvalue) | | X | X | X | X |
| **Inkrementieren** | | | | | | |
| ++p | Ergebnis= neuer Wert | X | X | X | X | X |
| p++ | Ergebnis= alter Wert | X | X | X | X | X |
| **Dekrementieren** | | | | | | |
| --p | Ergebnis= neuer Wert | | | | X | X |
| p-- | Ergebnis= alter Wert | | | | X | X |
| **Vergleichen** | | | | | | |
| p==q | gleich | X | | X | X | X |
| p!=q | ungleich | X | | X | X | X |
| p<q | kleiner | | | | | X |
| p<=q | kleiner gleich | | | | | X |
| p>q | größer | | | | | X |
| p>=q | größer gleich | | | | | X |
| **Direktzugriff** | | | | | | |
| p=p+n | n Positionen nach p | | | | | X |
| p+=n | n Positionen nach p | | | | | X |
| p=p-n | n Positionen vor p | | | | | X |
| p-=n | n Positionen vor p | | | | | X |
| x=p[n] | x=*(p+n) | | | | | X |
| p[n]=x | *(p+n)=x | | | | | X |
| **Iteratordistanz** | | | | | | |
| n=p-q | Entfernung in Anz. Positionen | | | | | X |

◇ **Anmerkung zum Inkrementieren/Dekrementieren** :
Da der **Pre-Increment**- und der **Pre-Decrement-Operator** den aktuellen – neuen – Wert (und nicht den alten Wert) zurückliefern, lassen sie sich wesentlich **effizienter** als die entsprechenden Post-Operatoren **implementieren**.
Man sollte sie daher gegenüber diesen bevorzugt einsetzen : **`++p`** ist **effizienter** als `p++`.

## • Iterator-Kategorien der STL-Container-Klassen

◇ Die STL-Containerklassen **`vector`** und **`deque`** unterstützen **Random-Acces-Iteratoren**

◇ Die STL-Containerklassen **`list, set, multiset, map`** und **`multimap`** unterstützen **Bidirectional-Iteratoren**

# Standard-Template-Library (STL) von C++ : Iteratoren (4)

- **Einfaches Demonstrationsprogramm zu Random-Access-Iteratoren `randiterdem`**

```
// C++-Quelldatei randiterdem_m.cpp --> Programm randiterdem
// Einfaches Demo-Programm zu Random-Access-Iteratoren der STL

#include <vector>
#include <iostream>
using namespace std;


int main(void)
{
  vector<int> zahlen;
  int i;
  vector<int>::iterator pos;

  for (i=1; i<=10; i++)
    zahlen.push_back(i);

  cout << "\nAnzahl : " << zahlen.end()- zahlen.begin() << endl;

  cout << "Inhalt :\n";
  cout << "Verwendung *-Operator\n";
  for (pos=zahlen.begin(); pos<zahlen.end(); ++pos)
    cout <<*pos << "  ";
  cout << endl;

  cout << "Zugriff ueber Index-Operator\n";
  for (i=0; i<zahlen.size(); i++)
    cout << zahlen.begin()[i] << "  ";
  cout << endl;

  cout << "jedes 2. Element\n";
  for (pos=zahlen.begin(); pos<zahlen.end(); pos+=2)
    cout <<*pos << "  ";
  cout << endl;


  return 0;
}
```

- **Ausgabe des Demonstrationsprogramms `randiterdem`**

```
Anzahl : 10
Inhalt :
Verwendung *-Operator
1  2  3  4  5  6  7  8  9  10
Zugriff ueber Index-Operator
1  2  3  4  5  6  7  8  9  10
jedes 2. Element
1  3  5  7  9
```

# Standard-Template-Library (STL) von C++ : Iteratoren (5)

- **Freie Iterator-Funktionen**

◇ Die STL stellt **zwei freie Funktionen** zur Verfügung, mit denen einige nur für Random-Access-Iteratoren definierte Operationen auch für Input-Iteratoren (und damit auch für Bidirectional- und Forward-Iteratoren) ermöglicht werden.

◇ Die beiden als **Funktions-Templates** in der Headerdatei **`<iterator>`** definierten Funktionen werden nachfolgend in **vereinfachter Darstellung** deklariert.

---

◇ **`void advance(InputIterator& pos, Dist n);`**

▻ **Weitersetzen** des (Input-)**Iterators** `pos` um `n` Elemente

▻ Für Bidirectional- und Random-Access-Iteratoren darf `n` auch negativ sein.

▻ Die Typen `InputIterator` und `Dist` sind tatsächlich Template-Parameter.
Da alle Iteratoren, außer den Output-Iteratoren, die Funktionalität von Input-Iteratoren implementieren, kann die aktuelle Iteratorklasse eine Input-, Forward-, Bidirectional- oder Random-Access-Iterator-Klasse sein.
Der den formalen Typ-Parameter `Dist` ersetzende aktuelle Typ muss ein für die jeweilige aktuelle Iteratorklasse anwendbarer Ganzzahltyp zur Darstellung von "Iteratorabständen" sein.

---

◇ **`Dist distance(InputIterator first, InputIterator last);`**

▻ **Ermittlung des Abstands** zwischen den (Input-)Iteratoren `first` und `last`.
= Anzahl der Increments bzw. Decrements, um von `first` nach `last` zu gelangen.

▻ Beide Iteratoren müssen zum gleichen Container gehören.

▻ Der Iterator `last` muss vom Iterator `first` aus erreichbar sein.
Das bedeutet, dass bei allen Iteratoren, die keine Random-Access-Iteratoren sind, `last` hinter `first` liegen muss (d. h. über Increments erreichbar sein muss).

▻ Der Typ `InputIterator` ist tatsächlich Template-Parameter.
Aktuell kann es sich um eine Iteratorklasse der Kategorien Input-Iterator, Forward-Iterator, Bidirectional-Iterator oder Random-Access-Iterator handeln, da alle Iteratoren dieser Klassen die Funktionalität von Input-Iteratoren implementieren.

▻ Der Rückgabetyp `Dist` ist der für die jeweilige aktuelle Iteratorklasse definierte Ganzzahltyp zur Darstellung von "Iteratorabständen" ("Abstands-Typ" der Iteratorklasse).

---

◇ Eine **weitere freie Iterator-Funktion** ist in der Algorithmen-Bibliothek (Headerdatei **`<algorithm>`**) definiert.
Es handelt sich um eine Funktion, mit der die von zwei Iteratoren referierten **Elemente vertauscht** werden können.
Auch diese Funktion ist als Funktions-Template definiert.
Ihre Deklaration wird nachfolgend ebenfalls in vereinfachter Darstellung angegeben.

---

◇ **`void iter_swap(ForwardIterator1 a, ForwardIterator2 b);`**

▻ **Vertauschen** der durch die Iteratoren `a` und `b` **referierten Elemente**.

▻ Die Typen `ForwardIterator1` und `ForwardIterator2` sind tatsächlich Template-Parameter.
Aktuell kann für sie jede Iteratorklasse der Kategorien Forward-Iterator, Bididrectional-Iterator oder Random-Access-Iterator verwendet werden.

▻ Die beiden Iteratorklassen müssen nicht gleich sein. Die referierten Elemente müssen sich lediglich gegenseitig zuweisen lassen.

---

# Standard-Template-Library (STL) von C++ : Reverse-Iteratoren (1)

## • Allgemeines

◇ In der STL ist – in der Headerdatei **`<iterator>`** – der **Iterator-Adapter `reverse_iterator`** als Klassen-Template definiert. Template-Parameter ist eine Iterator-Klasse.
Dieser Iterator-Adapter erzeugt **Reverse-Iteratoren**

◇ Reverse-Iteratoren sind Iteratoren, bei denen die **Durchlaufrichtung** durch einen Container **umgekehrt** ist, d. h. ein **Inkrementieren** bewegt den Iterator zum **vorhergehenden Element**.
Mit ihrer Hilfe kann bei allen Algorithmen die Verarbeitungsreihenfolge der Elemente umgekehrt werden, ohne dass entsprechende neue Algorithmen implementiert werden müssen.

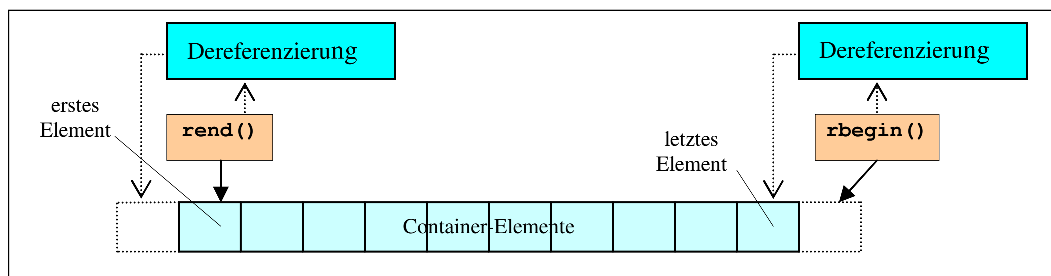
◇ Reverse-Iteratoren können nur zu Bidirectional- (und damit auch zu Random-Access-) Iteratoren gebildet werden.
Da **alle in der STL definierten Container-Klassen** Iteratoren dieser Kategorien zu unterstützen, lassen sich für sie auch Reverse-Iteratoren erzeugen.

▻ In **jeder Container-Klasse** – außer den Container-Adaptern – sind deshalb auch die implementierungsabhängigen `public`-Typen **`reverse_iterator`** und **`const_reverse_iterator`** definiert.
Container-Elemente, die von Reverse-Iterator-Objekten des Typs `const_reverse_iterator` referiert werden, können nicht geändert werden.

```
template < ... >
  class ...
  {
    public :
       // ...
       typedef std::reverse_iterator<iterator> reverse_iterator;
       typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
       // ...
  };
```

▻ Außerdem stellt **jede Container-Klasse** – außer den Container-Adaptern – auch **Memberfunktionen** zur Ermittlung des **Reverse-Iteratorbereichs** eines Container-Objekts zu Verfügung. :

- **`rbegin()`** liefert den Reverse-Iterator, der auf die Position unmittelbar nach dem letzten Element zeigt
  Dies ist der in einen Reverse-Iterator umgewandelte Funktionswert von `end()`.
- **`rend()`** liefert einen Reverse-Iterator, der auf das erste Element im Container zeigt.
  Dies ist der in einen Reverse-Iterator umgewandelte Funktionswert von `begin()`.




Auch diese **beiden Funktionen** existieren jeweils in **zwei überladenen Formen**, die eine liefert ein Reverse-Iterator-Objekt vom Typ `reverse_iterator`, die andere ein Reverse-Iterator-Objekt vom Typ `const_reverse_iterator`.

◇ Bei der **Erzeugung eines Reverse-Iterators** aus einem – normalen – Iterator **bleibt** die **Iterator-Position gleich**.
Um beim Durchlaufen von Containers-Bereichen mit Reverse-Iteratoren die **gleiche Semantik** wie mit – normalen – Iteratoren verwenden zu können (halboffenes Intervall, letzter Iterator zeigt auf nicht mehr vorhandenes Element), wird bei der **Dereferenzierung eines Reverse-Iterators** das **Element** zurückgeliefert, das sich an der **Position unmittelbar vor der eigentlich referierten Position** befindet.

# Standard-Template-Library (STL) von C++ : Reverse-Iteratoren (2)

- **Definition des Klassen-Templates `reverse_iterator` (unvollständiger Auszug)**

```
template <class Iterator>
  class reverse_iterator : public iterator< ... >
{
  protected :
    Iterator current;
  public :
    // Typ-Definitionen

    reverse_iterator();
    explicit reverse_iterator(Iterator x);
    template <class U> reverse_iterator(const reverse_iterator<U>& u);

    Iterator base() const;

    // Operatorfunktionen zum Element-Zugriff und zur Iterator-Aenderung

};

// freie Vergleichs-Operatorfunktionen
```

- **Konstruktoren**
  - ▻ Konstruktor zur Erzeugung eines **nichtinitialisierten** Reverse-Iterator-Objekts.
  - ▻ Konstruktor zur Erzeugung eines Reverse-Iterator-Objekts, das **mit einem** – normalen – **Iterator-Objekt initialisiert** wird
  - ▻ Copy-Konstruktor

- **Operationen mit Reverse-Iteratoren**
  - ◇ Für Reverse-Iteratoren sind die **gleichen Operationen** wie für die entsprechenden – normalen – Iteratoren definiert.
  - ◇ Allerdings sind **einige Wirkungs-Unterschiede** zu beachten :
    - ▻ Ein **Inkrementieren** des Reverse-Iterators wird in ein **Dekrementieren umgesetzt** (statt `++p` → `--p`) und umgekehrt (`--p` → `++p`).
    - ▻ Eine Veränderung eines Reverse-Iterators um einen **positiven Abstand** wird in eine Veränderung um einen **negativen Abstand umgesetzt** und umgekehrt.
    - ▻ Die **Dereferenzierung** eines Inverse-Iterators führt zur Rückgabe des **unmittelbar davor befindlichen Elements**
    - ▻ Bei **Vergleichsoperationen** werden die **Reihenfolgen der Operanden vertauscht** :
      **`x>y`** bzw **`x>=y`** führt zur Auswertung von **`y.current>x.current`** bzw **`y.current>=x.current`**
      **`x<y`** bzw **`x<=y`** führt zur Auswertung von **`y.current<x.current`** bzw **`y.current<=x.current`**
  - ◇ Zur **Rückwandlung eines Reverse-Iterators in einen – normalen – Iterator** existiert die Memberfunktion

    **`Iterator base() const;`**

    Sie liefert den – normalen – Iterator, der auf die gleiche Position wie der Reverse-Operator verweist.

    Wenn **`rpos`** ein Reverse-Iterator ist, kann somit mittels **`*rpos.base()`** auf das **tatsächlich referierte** (und nicht auf das davor befindliche) **Element zugegriffen** werden.

# Standard-Template-Library (STL) von C++ : Stream-Iteratoren (1)

## • Allgemeines

◇ **Iteratoren** setzen voraus, dass die **Elemente** der von ihnen durchlaufbaren Datenmenge in einer **Reihenfolge** (Sequenz) **angeordnet** sind

◇ **Stream-Iteratoren** sind **Iterator-Adapter**, die es ermöglichen, dass zu Streams, die ja auch aus einer Folge von Daten bestehen, mit der **gleichen Schnittstelle** wie zu Containern zugegriffen werden kann.
Dadurch lassen sich zahlreiche **Algorithmen der STL** (z.B. zum Kopieren) auch sinnvoll **direkt auf Streams** anwenden, d.h. Algorithmen können Eingabedaten statt aus Containern aus Streams beziehen und Ausgabedaten statt in Container in Streams ausgeben.

◇ **Ostream-Iteratoren** dienen zum **Ausgeben** von Elementen in Streams.
**Istream-Iteratoren** dienen zum **Einlesen** von Elementen aus Streams.

◇ Neben Stream-Iteratoren gibt es auch **Streambuffer-Iteratoren**. Diese ermöglichen einen **zeichenweisen** (und nicht wertweisen) Zugriff zu den von Streams verwendeten Streambuffern.

## • Ostream-Iteratoren

◇ Ostream-Iteratoren gehören zur Kategorie der **Output-Iteratoren**.
Statt an ein Container-Objekt sind sie an ein **Ausgabestream-Objekt gebunden**.
Dieses muss bei der Erzeugung eines Ostream-Iterator-Objekts bereitgestellt werden.
Ein Ostream-Iterator referiert immer die nächste Ausgabeposition im Stream.
Die **wesentliche Operation** – schreibender Zugriff zum referierten Element – führt zur **Ausgabe** des dem Element **zugewiesenen Werts** in den **Ausgabestream**.

◇ Zur Erzeugung von Ostream-Iteratorklassen definiert die STL das **Klassen-Template `ostream_iterator`**
**Template-Parameter** sind :
- der **Typ `T`** der referierten (d.h. auszugebenden) **Daten**.
- Zeichentyp **`charT`** (Default : `char`)
- Zeicheneigenschaften-Typ `traits` (Default : `char_traits<charT>`).

◇ **Konstruktoren** (Vereinfachte Darstellung für die Default-Template-Parameter):

| | |
|---|---|
| `ostream_iterator(ostream& s);` | Erzeugung eines Ostream-Iterators für das `ostream`–Objekt `s` |
| `ostream_iterator(ostream& s, const char* del);` | Erzeugung eines Ostream-Iterators für das `ostream`–Objekt `s`, der nach jeder Ausgabe eines Elements den C-String `del` ausgibt |
| `ostream_iterator(const ostream_iterator<T>& x);` | Copy-Konstruktor |

◇ **Zuweisungs-Operatorfunktion** (Vereinfachte Darstellung für die Default-Template-Parameter):

| | |
|---|---|
| `ostream_iterator<T>& operator=(const T& val);` | Ausgabe des Werts `val` in den Ausgabestream |

Statt durch eine Zuweisung an den dereferenzierten Iterator kann ein Wert auch durch direkte Zuweisung an den Iterator ausgegeben werden, d.h. **`*p=val`** und **`p=val`** sind **wirkungsgleich**. (`p` sei ein Ostream-Iterator)

◇ **Weitere Operationen**

▻ Die beiden **Increment-Operatorfunktionen** sind zwar definiert, sie bewirken aber nichts und geben lediglich eine Referenz auf das aktuelle Ostream-Iterator-Objekt zurück.

▻ Genaugenommen gibt auch die **Dereferenzierungs-Operatorfunktion** lediglich eine Referenz auf das aktuelle Objekt zurück. Die Anwendung des Zuweisungs-Operators auf dieses führt dann erst zur Ausgabe eines Werts.

# Standard-Template-Library (STL) von C++ : Stream-Iteratoren (2)

## • Istream-Iteratoren

◇ Istream-Iteratoren sind **Input-Iteratoren**.
Statt an ein Container-Objekt sind sie an ein **Eingabestream-Objekt** gebunden.
Dieses muss bei der Erzeugung eines Istream-Iterator-Objekts dem Konstruktor übergeben werden.
Ein Istream-Iterator **referiert** immer die **aktuelle Eingabeposition im Stream**. Beim **Erzeugen** eines Istream-Iterators sowie bei jedem Incrementieren des Iterators wird ein **Element** aus dem Stream **gelesen**.
Jeder **dereferenzierende Zugriff zum Iterator**, der **nur lesend** zulässig ist, liefert das **zuletzt eingelesene Element**.

◇ Die STL definiert das **Klassen-Template `istream_iterator`** zur Erzeugung von Istream-Iteratorklassen
**Template-Parameter** sind :
- der **Typ `T`** der referierten (d. h. auszugebenden) **Daten**.
- Zeichentyp **`charT`** (Default : `char`)
- Zeicheneigenschaften-Typ **`traits`** (Default : `char_traits<charT>`).
- Abstands-Typ **`Dist`** (Default : `ptrdiff_t`, ein vorzeichenloser Ganzzahltyp)

◇ **Konstruktoren** (Vereinfachte Darstellung für die Default-Template-Parameter):

| | |
|---|---|
| **`istream_iterator`**`();` | Erzeugung eines End-of-Stream-Iterators (referiert `EOF`) |
| **`istream_iterator`**`(istream& s);` | Erzeugung eines Istream-Iterators für das `istream`-Objekt `s`,<br>Einlesen des ersten Elements aus dem Stream |
| **`istream_iterator`**`(const istream_iterator<T>& x);` | Copy-Konstruktor |

◇ **Member-Operatorfunktionen** (Vereinfachte Darstellung für die Default-Template-Parameter):

| | |
|---|---|
| `const T&` **`operator*`**`() const;` | Rückgabe des zuletzt gelesenen Elements |
| `const T*` **`operator->`**`() const;` | Rückgabe eines Pointers auf das zuletzt gelesene Element,<br>über diesen kann ein Zugriff zu seinen Komponenten erfolgen |
| `istream_iterator<T>&` **`operator++`**`();` | Einlesen des nächsten Elements aus dem Stream,<br>Rückgabe des fortgeschalteten neuen Iterators |
| `istream_iterator<T>` **`operator++`**`(int);` | Einlesen des nächsten Elements aus dem Stream,<br>Rückgabe des alten Iterators |

Falls beim **Inkrementieren** des Istream-Iterators das **Stream-Objekt** in einen **Fehlerzustand** gelangt (z. B. auch bei Erreichen von `EOF`), wird der fortgeschaltete neue Iterator zu einem **End-of-Stream-Iterator**.

◇ **freie Vergleichsfunktionen**

▻ Die Operatorfunktionen für den **Vergleich auf Gleichheit** (**`operator==()`**) und **Ungleichheit** (**`operator!=()`**) sind als **freie Funktions-Templates** definiert

▻ Zwei Istream-Iteratoren sind dann **gleich**,
- wenn sie beide kein gültiges Element mehr referieren (d. h. End-of-Stream-Iteratoren sind)
- oder wenn beide keine End-of-Stream-Iteratoren sind und zum gleichen `istream`-Objekt gehören

▻ **Anmerkung zur Überprüfung auf Streamende (`EOF`)** :
Nach jedem Inkrementieren eines Istream-Iterators sollte durch Vergleich des Iterators mit einem End-of-Stream-Iterator der Fehlerzustand des Streams überprüft (und damit auch auf `EOF` geprüft) werden.

# Standard-Template-Library (STL) von C++ : Stream-Iteratoren (3)

- **Einfaches Demonstrationsprogramm zu Stream-Iteratoren `streamiterdem`**

```
// C++-Quelldatei streamiterdem_m.cpp --> Programm streamiterdem
// Einfaches Demo-Programm zu Stream-Iteratoren der STL

#include <iostream>
#include <iterator>
using namespace std;


int main(void)
{
  cout << "\nGeben Sie Integer-Werte ein : ";

  istream_iterator<int> input(cin); // Erzeugung Istream-Iterator
                                    // erster int-Wert wird aus Stream cin gelesen
  istream_iterator<int> eof;        // Erzeugung End-of-Stream-Iterator
  int i1, cnt=0, sum=0;

  while (input!=eof)
  {
    i1=*input;                   // Rückgabe letzter gelesener int-Wert
    cnt++;
    sum+=i1;
    ++input;                     // naechster int-Wert wird aus Stream cin gelesen
  }

  ostream_iterator<int> output(cout, "\n");
  cout << "\nAnz. eingegebener Werte : ";
  *output=cnt;                   // Ausgabe Anzahl Werte, alternativ : output=cnt;
  cout << "Summe aller Werte       : ";
  *output=sum;                   // Ausgabe Summe,        alternativ : output=sum;

  return 0;
}
```

- **Ein- und Ausgabe des Demonstrationsprogramms `streamiterdem` (Beispiel)**

```
Geben Sie Integer-Werte ein : 3 4 6 7 9 10
^Z
^Z

Anz. eingegebener Werte : 6
Summe aller Werte       : 39
```

# Standard-Template-Library (STL) von C++ : Insert-Iteratoren (1)

- **Allgemeines**

  ◇ Bei "**normalen**" **Iteratoren** bewirkt eine **Zuweisung** an das durch den Iterator referierte Element, dass dieses **Element überschrieben** wird (d. h. einen neuen Wert bekommt).

  **Beispiel** :
  ```
  // first, last und res seien normale Iteratoren
  while (first != last) *res++ = *first++;
  ```
  Obige Schleife bewirkt ein Kopieren der Elemente des Bereichs `[first, last)` (Quellbereich) in die Elemente eines Bereichs der mit `res` beginnt (Zielbereich). Die Elemente des Zielbereichs müssen hierzu existieren.

  ◇ Insert-Iteratoren sind **Iterator-Adapter**, die eine **Zuweisung** an den Iterator bzw an das referierte Element in ein **Einfügen** umsetzen.
  Wenn `res` in obigem Beispiel ein Insert-Iterator ist, werden Kopien der Elemente aus dem Quellereich als **neue** zusätzliche **Elemente** in den Zielbereich **eingefügt**.

- **Operationen**

  ◇ Insert-Iteratoren gehören zur Kategorie der **Output-Iteratoren**.
  Das bedeutet, dass zu einem dereferenzierten Iterator (also dem referierten Element) nur **schreibend** zugegriffen werden kann.

  ◇ Analog zu Ostream-Iteratoren kann ein derartiger schreibender Zugriff (**Zuweisung)** auch **direkt an** den **Iterator** erfolgen.
  D.h. auch für einen Insert-Iterator `p` gilt : **`*p=val`** und **`p=val`** sind **wirkungsgleich**
  Dies wird dadurch ermöglicht, dass die **Dereferenzierungs-**Operatorfunktion **`operator*()`** nicht das referierte Element sondern das Iterator-Objekt selbst `(*this)` zurückgibt, d. h. für den Insert-Iterator `p` gilt **`*p==p.`**

  ◇ Die **Zuweisungs-**Operatorfunktion **`operator=()`** ist so definiert, dass sie eine Kopie des ihr als Parameter übergebenen Elements als **neues Element** in den **Container** des Zielbereichs **einfügt**.
  Das Einfügen erfolgt **unmittelbar vor** der Position, auf die der Iterator zeigt.

  ◇ Die beiden **Increment**-Operatorfunktionen sind zwar auch definiert, aber wie bei Ostream-Iteratoren **bewirken** sie **nichts,** sondern geben lediglich das aktuelle Iterator-Objekt bzw eine Referenz hierauf zurück.
  Insert-Iteratoren können also ihre Position nicht verändern.

- **Insert-Iterator-Arten**

  ◇ In Abhängigkeit von der Position, an der sie ein Einfügen bewirken, werden **drei Arten** von Insert-Iteratoren unterschieden :

  ▻ **Front-Insert-Iteratoren (Front-Inserter)**
  Sie bewirken ein Einfügen am **Anfang** eines Containers (vor der ersten Position)
  Zum Einfügen rufen sie die Container-Funktion **`push_front(val)`** auf.
  Sie sind nur bei Containern, die diese Funktion zur Verfügung stellen, möglich : **Deques** und **Listen**

  ▻ **Back-Insert-Iteratoren (Back-Inserter)**
  Sie bewirken ein Einfügen am **Ende** eines Containers (hinter der letzten Position).
  Zum Einfügen rufen sie die Container-Funktion **`push_back(val)`** auf.
  Da nur **Vektoren**, **Deques** und **Listen** diese Funktion zur Verfügung stellen, sind sie nur bei diesen Container-Arten möglich.

  ▻ (positionierbare) **Insert-Iteratoren (Inserter)**
  Sie bewirken ein Einfügen vor einer **beliebigen gültigen** Container-**Position**.
  Die Einfügeposition ist bei der Erzeugung eines Insert-Iterators anzugeben.
  Zum Einfügen rufen sie die Container-Funktion **`insert(pos, val)`** auf.
  Da diese Funktion für alle STL-Container-Arten implementiert ist, lassen sich diese Insert-Iteratoren bei **jedem STL-Container** verwenden.
  **Anmerkung**: Bei assoziativen Containern hat die anzugebende Einfügeposition `pos` nur Hinweischarakter.

# Standard-Template-Library (STL) von C++ : Insert-Iteratoren (2)

- **Definition der Insert-Iterator-Klassen**

  ◇ Für die drei Iterator-Arten sind in der Headerdatei **`<iterator>`** **Klassen-Templates** definiert und implementiert :
  - `template <class Container> class` **`front_insert_iterator`**`;`
  - `template <class Container> class` **`back_insert_iterator`**`;`
  - `template <class Container> class` **`insert_iterator`**`;`

  ◇ **Definition des Klassen-Templates `front_insert_iterator`**

```
template <class Container>
  class front_insert_iterator : public iterator<...>
  {
    protected :
      Container* container;
    public :
      typedef Container container_type;
      explicit front_insert_iterator(Container&);
      front_insert_iterator<Container>&
        operator=(typename Container::const_reference val);
      front_insert_iterator<Container>& operator*();
      front_insert_iterator<Container>& operator++();
      front_insert_iterator<Container>  operator++(int);
  };
```

  ◇ Die **Definition** der beiden **anderen Klassen-Templates** erfolgt **analog**.
  Lediglich beim Klassen-Template **`insert_iterator`** besteht ein wesentlicher **Unterschied** :
  Dem Konstruktor ist die Einfügeposition als zusätzlicher Parameter (Typ : `Container::iterator`) zu übergeben.
  Diese wird in einer zusätzlichen `protected`–Datenkomponente gespeichert.

- **Erzeugung von Insert-Iteratoren**

  ◇ Mittels des jeweiligen **Konstruktors.**
  Diesem ist das **Container-Objekt**, für den der Insert-Iterator erzeugt werden soll, als **Parameter** zu übergeben.
  Der Konstruktor für einen **positionierbaren Insert-Iterator** (Klassen-Template **`insert_iterator`**) benötigt als **zusätzlichen Parameter** einen Iterator, der die **Einfügeposition** angibt.

  **Beispiel** :
```
vector<int> imeng;
back_insert_iterator<vector<int> > backint(imeng);
insert_iterator<vector<int> > posint(imeng, imeng.begin()+4);
```

  ◇ Als **Alternative** steht für jede Insert-Iterator-Art eine **freie Erzeugungsfunktion** in Form eines **Funktions-Templates** zur Verfügung. Diese erzeugen jeweils ein neues Insert-Iterator-Objekt und geben es als Funktionswert zurück.
  Diese Funktions-Templates sind ebenfalls in der Headerdatei **`<iterator>`** definiert.

```
template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);

template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);

template <class Container, class Iterator>
  insert_iterator<Container> inserter(Container& x, Iterator i);
```

  **Beispiel** : **`back_inserter(imeng)`**`=23;` `//` Einfügen neues Element mit Wert `23` am Ende von `imeng`

# Standard-Template-Library (STL) von C++ : Insert-Iteratoren (3)

- **Einfaches Demonstrationsprogramm zu Insert-Iteratoren `insiterdem`**

```
// C++-Quelldatei insiterdem_m.cpp --> Programm insiterdem
// Einfaches Demo-Programm zu Insert-Iteratoren der STL

#include <iostream>
#include <deque>
#include <iterator>
using namespace std;

template <class Container>
  void ausgabe(Container& cont)
  { typename Container::iterator di;
    ostream_iterator<typename Container::value_type> output(cout, "  ");
    cout << endl;
    for (di=cont.begin(); di!=cont.end(); ++di)
      *output=*di;
    cout << endl;
  }

int main(void)
{ deque<int> imeng;
  deque<int>::iterator di;
  int i;

  for (i=0; i<8; ++i)
  { imeng.push_back(i+1);
    imeng.push_front(i+11);
  }
  cout << "\nInhalt imeng :";
  ausgabe(imeng);

  back_insert_iterator<deque<int> > backint(imeng);
  for (i=0; i<4; i++)
    *backint++=i+20;                // increment ++ ist wirkungslos und uberfluessig
  cout << "\nnach back_insert :";
  ausgabe(imeng);

  di=imeng.begin()+8;
  *di=30;
  cout << "\nnach Zuweisung ueber normalen Iterator (Element Nr. 9):";
  ausgabe(imeng);

  *inserter(imeng, di) = 0;         // gleichbedeutend : inserter(imeng, di) = 0;
  cout << "\nnach Zuweisung ueber Insert-Iterator (Element Nr. 9):";
  ausgabe(imeng);
  return 0;
}
```

- **Ausgabe des Demonstrationsprogramms `insiterdem`**

```
Inhalt imeng :
18  17  16  15  14  13  12  11  1  2  3  4  5  6  7  8

nach back_insert :
18  17  16  15  14  13  12  11  1  2  3  4  5  6  7  8  20  21  22  23

nach Zuweisung ueber normalen Iterator (Element Nr. 9):
18  17  16  15  14  13  12  11  30  2  3  4  5  6  7  8  20  21  22  23

nach Zuweisung ueber Insert-Iterator (Element Nr. 9):
18  17  16  15  14  13  12  11  0  30  2  3  4  5  6  7  8  20  21  22  23
```

# Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (1)

## • Allgemeines

◇ Die STL stellt in der Algorithmen-Bibliothek ca. **70 Algorithmen** zur Bearbeitung von Containern und den in ihnen enthaltenen Elementen zur Verfügung.

◇ Diese Algorithmen sind generisch als **freie Funktions-Templates** implementiert.
Sie greifen zu den Elementen der Container nur **indirekt über Iteratoren** zu, d.h. sind unabhängig von speziellen Implementierungs-Details der verschiedenen Container-Klassen.
Diese **Trennung** der **Algorithmen** von den **Containern** widerspricht zwar objektorientierten Grundkonzepten, hat aber den Vorteil einer **größeren Flexibilität**, **effizienteren Implementierung** (Algorithmus muss nicht für jede Container-Klasse gesondert als Memberfunktion realisiert werden) und **einfacheren Erweiterbarkeit** (Anwendung auch für selbst-definierte Container- und Iterator-Klassen)

◇ **Alle Algorithmen** bearbeiten einen oder mehrere **Bereiche** (*ranges*) von **Elementen**.
Ein derartiger Bereich kann alle Elemente eines Containers umfassen oder nur aus einer Teilmenge derselben bestehen.
Der zu bearbeitende Bereich wird durch **zwei** als **Funktionsparameter** zu übergebende **Iteratoren** festgelegt.
Dabei referiert der erste Iterator das erste zu bearbeitende Element und der zweite Iterator das Element, das auf das letzte zu bearbeitende unmittelbar folgt.
Die zu bearbeitende Menge wird also immer als eine **halboffene Menge** angegeben : Sie besteht aus allen Elementen zwischen dem ersten (einschließlich) und dem letzten (ausschließlich) Element.
Dabei führen die Algorithmen **keinerlei Bereichs- und Gültigkeitsüberprüfungen** aus.
Insbesondere muss der Anwender eines Algorithmus darauf achten, dass das übergebene **Ende** eines Bereiches **vom Anfang** aus **erreichbar** sein muss. Das bedeutet, dass beide übergebenen Iteratoren zu demselben Container-Objekt gehören müssen und die Ende-Position sich logisch nicht vor der Anfangsposition befinden darf.

◇ Bei den meisten Algorithmen, die mit **mehreren Bereichen** arbeiten, wird **nur der erste** Bereich direkt durch **Anfangs- und End-Iterator** festgelegt. Für die **anderen Bereiche** wird **nur** ein **Anfangs-Iterator** angegeben. Das Ende ergibt sich dann aus der Funktion und der Arbeitsweise des Algorithmus.
Bei Algorithmen, die auf einen derartigen Bereich **schreibend** zugreifen (z.B. durch Kopieren in einen Zielbereich), muss **sichergestellt** werden, dass dieser **Bereich** eine **ausreichende Groesse** besitzt.
Als **Alternative** können einfügende Iteratoren (**Insert-Iteratoren**) für den Zielbereich verwendet werden.

◇ Bei **mehreren Bereichen** können diese in **Containern unterschiedlichen Typs** liegen.

◇ Sehr viele Algorithmen liefern einen **Iterator** als **Rückgabewert**.
Dabei wird der **End-Iterator** des übergebenen Bereichs zur **Kennzeichnung** eines **Misserfolgs** (z.B. "nicht gefunden") oder einer sonstigen Fehlfunktion verwendet.

◇ **Nicht alle Algorithmen** können auf **alle Container**-Klassen **angewendet** werden.
U.a. bestimmt die vom Algorithmus verwendete **Iterator-Katogerie** die Anwendbarkeit. So lassen sich z.B. Algorithmen, die mit Random-Access-Iteratoren arbeiten, nicht auf assoziative Container anwenden.
Weiterhin können Algorithmen, die Elemente verändern, nicht auf Container angewendet werden, deren **Elemente** als **konstant** betrachtet werden (z.B. assoziative Container).

◇ Die **Anwendung** der Algorithmen ist **nicht auf die Container-Klassen** der STL **beschränkt**. Sie lassen sich vielmehr für **alle Sequenzen** von Daten, für die **Iteratoren** zum Durchlaufen existieren, einsetzen. Insbesondere arbeiten sie auch

- ▻ mit **Ein- bzw. Ausgabe-Streams**,
- ▻ den **String-Klassen** der Standardbibliothek (es existieren String-Memberfunktionen zur Erzeugung von Anfangs- und End-Iteratoren)
- ▻ sowie **normalen C-Arrays** (als Iteratoren dienen Pointer auf die Array-Elemente)

◇ Die **Deklarationen** (und **Implementierungen**) der Funktions-Templates der STL-Algorithmen befinden sich in der **Headerdatei `<algorithm>`.**

◇ Eine besondere – sehr kleine – Gruppe von **numerischen Algorithmen** ist nicht in der STL sondern in der **numerischen Bibliothek** enthalten. Sie sind in der **Headerdatei `<numeric>`** definiert.

# Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (2)

## • Bedeutung der Template-Parameter-Namen

◇ **Template-Parameter** können sein :

- **Iteratorklassen**
  Jedes Funktions-Template ist mit wenigstens einem Iteratorklassen-Typ parametrisiert
- **Funktionsobjekt-Klassen**
  Eine Reihe von Algorithmen können durch einen derartigen Template-Parameter konfiguriert werden.
  Dadurch wird ihre Funktionalität modifiziert und erweitert.
- **Typ** der zu bearbeitenden **Daten**
  Einen derartigen Template-Parameter besitzen einige Algorithmen
- **Datentyp** zur Darstellung **ganzer Zahlen** (Größen-Angaben)
  Einen derartigen Template-Parameter besitzen einige wenige Algorithmen

◇ Die **Namen** der **formalen Template-Parameter** sind so gewählt, dass sie die **Anforderungen** ausdrücken, die an die aktuellen Typ-Parameter gestellt werden.

◇ So bedeuten z.B. die formalen Typangaben **`InputIterator, InputIterator1`** oder **`InputIterator2,`** dass der entsprechende aktuelle Typ die funktionellen Anforderungen eines **Input-Iterators** erfüllen muss.
Diese werden von allen Iterator-Kategorien außer den Output-Iteratoren bereitgestellt, d.h. aktueller Iterator-Typ kann dann auch ein Forward-Iterator, ein Bidirectional-Iterator oder ein Random-Access-Iterator sein.

◇ Analog bedeutet die formale Typangabe **`Predicate,`** dass der aktuelle Typ einstellige Funktionsobjekte beschreiben muss, die – auf einen dereferenzierten Iterator angewendet – einen auf `true` prüfbaren Wert liefern.

In den in der ANSI-Norm enthaltenen Funktions-Deklarationen werden ausführliche und damit relativ lange formale Typnamen verwendet, die den beiden o.a. Beispielen entsprechen.
Zur Erhöhung der Übersichtlichkeit und besseren Lesbarkeit werden diese hier durch die folgenden **abgekürzten Typ-Namen** ersetzt.

- **`InpIt`** für `InputIterator`
- **`OutpIt`** für `OutputIterator`
- **`ForwIt`** für `ForwardIterator`
- **`BidirIt`** für `BidirectionalIterator`
- **`RandIt`** für `RandomAccessIterator`
- **`Pred`** für `Predicate` (Klasse für einstellige Prädikate)
- **`BinPred`** für `BinaryPredicate` (Klasse für zweistellige Prädikate)
- **`Comp`** für `Compare` (Klasse für Vergleich-Funktionsobjekte)
- **`UnOp`** für `UnaryOperation` (Klasse für einstellige Funktionsobjekte)
- **`BinOp`** für `BinaryOperation` (Klasse für zweistellige Funktionsobjekte)
- **`Func`** für `Function` (allgemeine Funktionsobjekt-Klasse)

◇ **Anmerkung zu Funktionsobjekt-Parametern** :
Für Funktionsobjekt-Parameter können den Algorithmen aktuell statt Funktionsobjekten auch **Funktions-Pointer** entsprechender Funktionalität übergeben werden.

# Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (3)

## • Überblick über die Algorithmen

### ◇ Nichtmodifizierende Algorithmen

Sie führen nur Lesezugriffe zu Elementbereichen aus und ändern damit weder den Wert noch die Reihenfolge von Elementen des jeweiligen Bereichs.

- Suchen nach Elementen und Elementfolgen (diverse verschiedene Suchkriterien)
- Zählen von Elementen
- Vergleich von Elementbereichen
- Ausführen einer – nicht modifizierenden – Funktion für alle Elemente

### ◇ Modifizierende Algorithmen

Sie führen auch Schreibzugriffe zu Elementbereichen aus und ändern damit den Wert und/oder die Reihenfolge von Elementen

#### ▻ Wert-ändernde Algorithmen

Sie verändern den Wert von Elementen bzw fügen Elemente hinzu (in einem Quellbereich und/oder Zielbereich)

- Kopieren und ersetzendes Kopieren von Elementbereichen
- Vertauschen der Elemente von zwei Bereichen
- Transformation von Elementen
- Ersetzen von Elementwerten
- Füllen von Elementbereichen
- Zusammenfassung der Elemente zweier sortierter Bereiche zu einem neuen sortierten Bereich

#### ▻ Löschende Algorithmen

Sie entfernen Elemente aus einem Bereich. Hierzu zählen auch Algorithmen, die einen Quellbereich unverändert lassen und aus diesem nur einen Teil der Elemente in einen Zielbereich kopieren.

- Entfernen von Elementen
- Teil-Löschendes Kopieren von Elementbereichen

#### ▻ Mutierende Algorithmen

Sie verändern lediglich die Reihenfolge von Elementen, nicht aber ihren Wert.

- Vertauschen von zwei Elementen
- Umkehrung der Elementreihenfolge
- Rotation von Elementen
- Permutieren der Elemente eines Bereichs
- Umverteilung ("*shuffle*") der Elemente eines Bereichs
- Verschiebung von Element-Teilbereichen

#### ▻ Sortierende Algorithmen

Sie verändern die Reihenfolge der Elemente eines Bereichs so, dass sie anschließend wenigstens teilweise sortiert sind. Diese Algorithmen existieren alle in zwei Versionen : Die eine Version verwendet zum Sortieren den <–Operator (Default-Sortierkriterium), die andere verwendet ein als Funktionsobjekt übergebenes Sortierkriterium.

- Sortieren aller Elemente eines Bereichs
- Sortieren der ersten n Elemente eines Bereichs
- Sortieren hinsichtlich eines Vergleichselements

#### ▻ Heap-Operationen

Ein Heap ist ein spezieller binärer Baum, bei dem das kleinste (bzw größte) Element Wurzelelement (1. Element) ist.

- Erzeugen, Verändern (Element-Einfügen, -Entfernen) und Sortieren (=Zerstören) eines Heaps

#### ▻ Mengen-Operationen

Diese Algorithmen arbeiten mit sortierten Bereichen (Mengen)

- Bildung der Vereinigungs-, Schnitt-, Differenz- und Komplementär-Menge zweier Mengen (→ neuer Bereich)

# Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (4)

## • Anmerkung zur Darstellung der Algorithmen

◇ Zur Vereinfachung und Erhöhung der Übersichtlichkeit werden im folgenden die Funktions-Deklarationen der Algorithmen nicht in der originalen Template-Form sondern in einer vereinfachten Darstellung wiedergegeben :
Der Template-Zusatz wird weggelassen.
Er kann aber leicht rekonstruiert werden, da die Typen aller Funktionsparameter auch Template-Parameter sind.

◇ **Beispiel** :

statt :

```
template <class ForwIt, class Size, class T, class BinPred>
  ForwIt search_n(ForwIt first, ForwIt last, Size count,
                  const T& val, BinPred pred);
```

wird formuliert :

```
ForwIt search_n(ForwIt first, ForwIt last, Size count,
                const T& val, BinPred pred);
```

## • Einige nichtmodifizierende Algorithmen

◇ Die Algorithmen dieser Gruppe arbeiten mit Input- und Forward-Iteratoren und sind daher prinzipiell auf alle Container-Klassen anwendbar. Allerdings setzen einige von ihnen sortierte Bereiche voraus.

| Deklaration | Beschreibung |
|---|---|
| `InpIt` **`find`**`(InpIt first, InpIt last, const T& val);` | Suchen nach dem ersten Element mit dem Wert `val`<br>Rückgabe: Position des Elements, falls gefunden, sonst `last` |
| `InpIt` **`find_if`**`(InpIt first, InpIt last, Pred prd);` | Suchen nach dem ersten Element für das das Prädikat `prd` erfüllt ist<br>Rückgabe: Position des Elements, falls gefunden, sonst `last` |
| `ForwIt` **`search`**`(ForwIt1 first1, ForwIt1 last1, ForwIt2 first2, ForwIt2 last2);` | Suchen nach erstem Vorkommen des 2. Bereichs im 1. Bereich, Rückgabe : Pos. des 1.Elements des gefundenen Teilbereichs, bzw. `last1` |
| `ForwIt` **`lower_bound`**`(ForwIt first, ForwIt last, const T& val);` | Rückgabe: Position des ersten Elements dessen Wert `>= val` ist (Bereich muss sortiert sein) |
| `const T&` **`max`**`(const T& a, const T& b);` | Rückgabe: Referenz auf größeres Element |
| `ForwIt` **`max_element`**`(ForwIt first, FormIt last);` | Rückgabe: erste Position des größten Elements |
| `ForwIt` **`max_element`**`(ForwIt first, FormIt last, Comp cmp);` | Rückgabe: erste Position des größten Elements<br>Vergleich erfolgt mittels `cmp` |
| `InpIt::difference_type` **`count`**`(InpIt first, InpIt last, const t& val);` | Suchen nach Elementen mit dem Wert val<br>Rückgabe: Anzahl der gefundenen Elemente |
| `InpIt::difference_type` **`count_if`**`(InpIt first, InpIt last, Pred prd);` | Suchen nach Elementen, für die `prd` erfüllt ist<br>Rückgabe: Anzahl der gefundenen Elemente |
| `bool` **`equal`**`(InpIt1 first1, InpIt1 last1, InpIt2 first2);` | Überprüfung zweier Bereiche auf Gleichheit<br>Rückgabe: `true`, wenn gleich, sonst `false` |
| `UnOp` **`for_each`**`(InpIt first, InpIt last, UnOp f);` | Aufruf von `f(elem)` für alle Elemente `elem` des Bereichs `[first, last)`, Rückgabe: `f` |

# Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (5)

- **Einige wert-ändernde Algorithmen**
  - ◇ Da die Elemente von assoziativen Containern als konstant betrachtet werden, lassen sich diese Algorithmen nicht anwenden, wenn der Bereich, in dem Elemente geändert werden, in einem assoziativen Container liegt.

| | |
|---|---|
| `OutpIt copy(InpIt first, InpIt last, OutpIt res);` | Kopieren der Elemente aus dem Bereich `[first,last)` in den bei `res` beginnenden Bereich<br>`res` darf nicht im Bereich `[first,last)` liegen<br>Rückgabe: `res + (last - first)` |
| `BidirIt copy_backward(BidirIt1 first, BidirIt1 last, BidirIt2 res);` | Kopieren der Elemente aus dem Bereich `[first,last)` in umgekehrter Reihenfolge in den Bereich ab `res`<br>`res` darf nicht im Bereich `[first,last)` liegen<br>Rückgabe : `res + (last - first)` |
| `OutpIt replace_copy(InpIt first, InpIt last, OutpIt res, const T& alt, const T& neu);` | Kopieren der Elemente aus dem Bereich `[first,last)` in den bei `res` beginnenden Bereich, wobei der Wert `alt` durch den Wert `neu` ersetzt wird.<br>`res` darf nicht im Bereich `[first,last)` liegen<br>Rückgabe: `res + (last - first)` |
| `ForwIt2 swap_ranges(ForwIt1 first1, ForwIt1 last1, ForwIt2 first2);` | Vertauschen der Elemente des Bereiches `[first1, last1)` mit den Elementen des Bereichs der mit `first2` beginnt.<br>Rückgabe: `first2 + (last1 - first1)`<br>Die beiden Bereiche dürfen sich nicht überlappen |
| `OutpIt transform(InpIt first, InpIt last, OutpIt res, UnOp op);` | Jedes Element `*pos` aus dem Bereich `[first,last)` wird mittels `op(*pos)` transformiert und als neues Element in den Bereich, der bei `res` beginnt, kopiert.<br>`res` darf mit `first` übereinstimmen<br>Rückgabe: `res + (last - first)` |
| `void replace(ForwIt first, ForwIt last, const T& alt, const T& neu);` | Ersetzen des Werts `alt` der Elemente aus dem Bereich `[first,last)` durch den Wert `neu`. |
| `void fill(ForwIt first, ForwIt last, const T& val);` | Zuweisen des Werts `val` an jedes Element aus dem Bereich `[first,last)` |
| `void generate(ForwIt first, ForwIt last, Generator gen);` | Aufruf von `gen()` für jedes Element aus dem Bereich `[first,last)` und Zuweisen des dadurch erzeugten Werts an das jeweilige Element |
| `void generate_n(OutpIt first, Size n, Generator gen);` | Aufruf von `gen()` für die ersten `n` Elemente des mit `first` beginnenden Bereichs und Zuweisen des dadurch erzeugten Werts an das jeweilige Element |
| `OutpIt merge(InpIt1 first1, InpIt1 last1, InpIt2 first2, InpIt2 last2, OutpIt res);` | Zusammenfassen der Elemente der beiden sortierten Bereiche `[first1, last1)` und `[first2, last2)` in einen bei `res` beginnenden ebenfalls sortierten Bereich<br>Quell- und Zielbereiche dürfen sich nicht überlappen<br>Rückgabe: Position hinter dem letzten Element im Zielber. |

# Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (6)

## • Einige löschende Algorithmen

◇ Da die Elemente von assoziativen Containern als konstant betrachtet werden, lassen sich diese Algorithmen nicht anwenden, wenn der Bereich, in dem Elemente geändert werden, in einem assoziativen Container liegt.

| Funktion | Beschreibung |
|---|---|
| `ForwIt` **`remove`**`(ForwIt first, ForwIt last,`<br>`const T& val);` | Löschen aller Elemente aus dem Bereich `[first,last)`, die den Wert `val` haben<br>Rückgabe: Position hinter dem letzten Element im modifizierten Bereich |
| `ForwIt` **`remove_if`**`(ForwIt first,`<br>`ForwIt last, Pred prd);` | Löschen aller Elemente aus dem Bereich `[first,last)` für die das Prädikat `pred` erfüllt (`true`) ist<br>Rückgabe: Pos. hinter dem letzten Element im mod. Bereich |
| `OutpIt` **`remove_copy`** `(InpIt first,`<br>`InpIt last,`<br>`OutpIt res);`<br>`const T& val);` | Kopieren der Elemente aus dem Bereich `[first,last)`, die nicht den Wert `val` haben, in den bei `res` beginnenden Bereich<br>Rückgabe: Position hinter dem letzten Element im Zielber. |
| `OutpIt` **`remove_copy_if`** `(InpIt first,`<br>`InpIt last,`<br>`OutpIt res);`<br>`Pred prd);` | Kopieren der Elemente aus dem Bereich `[first,last)`, für die das Prädikat `prd` nicht erfüllt ist, in den bei `res` beginnenden Bereich<br>Rückgabe: Position hinter dem letzten Element im Zielber. |
| `ForwIt` **`unique`**`(ForwIt first, ForwIt last);` | Löschen aller Elemente aus dem Bereich `[first,last)`, die einem Element mit gleichem Wert unmittelbar folgen<br>Rückgabe : Pos. hinter dem letzten Element im mod. Bereich |
| `OutpIt` **`unique_copy`**`(InpIt first,`<br>`InpIt last,`<br>`OutpIt res);` | Kopieren der Elemente aus dem Bereich `[first,last)`, die nicht einem Element mit gleichem Wert unmittelbar folgen, in den bei `res` beginnenden Bereich,.<br>`res` darf nicht im Bereich `[first,last)` liegen<br>Rückgabe: Position hinter dem letzten Element im Zielber. |

## • Einige sortierende Algorithmen (nur für Container mit Random-Access-Iteratoren)

| Funktion | Beschreibung |
|---|---|
| `void` **`sort`**`(RandIt first, RandIt last);` | Sortieren der Elemente im Bereich `[first,last)`, nach aufsteigender Reihenfolge (mit `operator<`) |
| `void` **`sort`**`(RandIt first, RandIt last,`<br>`Comp cmp);` | Sortieren der Elemente im Bereich `[first,last)` mittels des durch `cmp` festgelegten Vergleichskriteriums |
| `void` **`partial_sort`**`(RandIt first,`<br>`RandIt middle,`<br>`RandIt last);` | Sortieren der Elemente im Bereich `[first,last)` so, dass anschließend die ersten `(middle-first)` sortierten Elemente im Bereich `[first, middle)` stehen.<br>Sortierung nach aufsteig. Reihenfolge (mit `operator<`)<br>Die restlichen Elemente sind unsortiert. |
| `void` **`nth_element`**`(RandIt first,`<br>`RandIt nth,`<br>`RandIt last);` | Platzierung des Elements aus dem Bereich `[first,last)` das bei vollständiger Sortierung an der Position `nth` stehen würde, an diese Position (Vergleichselement).<br>Im Teilbereich `[first, nth)` befinden sich (unsortiert) nur Elemente, die kleiner gleich dem Vergleichselement sind, im Teilbereich `[nth+1, last)` die restlichen Elemente |

# Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (7)

- **Einige mutierende Algorithmen**

  ◇ Da die Elemente von assoziativen Containern als konstant betrachtet werden, lassen sich diese Algorithmen nicht anwenden, wenn der Bereich, in dem Elemente geändert werden, in einem assoziativen Container liegt.

| | |
|---|---|
| `void swap(T& a, T& b);` | Vertauschen der Elemente `a` und `b` |
| `void reverse(BidirIt first, BidirIt last);` | Umkehren der Reihenfolge der Elemente des Bereichs `[first,last)` |
| `void rotate(ForwIt first, ForwIt middle, ForwIt last);` | Rotieren der Elemente aus dem Bereich `[first,last)` um `(middle - first)` Positionen nach links, so dass sich anschließend das ehemalige Element an der Position `middle` an der Position `first` befindet<br>`middle` muss im Bereich `[first,last)` liegen |
| `OutpIt rotate_copy(ForwIt first, ForwIt middle, ForwIt last, OutpIt res);` | Kopieren der Elemente aus dem Bereich `[first,last)` in den bei `res` beginnenden Bereich, wobei die Elemente um `(middle - first)` Positionen nach links rotiert werden, so dass sich anschließend das ehemalige Element an der Position `middle` an der Position `res` im Zielbereich befindet.<br>Quell- und Zielbereich dürfen sich nicht überlappen<br>Rückgabe: `res + (last - first)` |
| `void random_shuffle(RandIt first, RandIt last);` | Änderung der Reihenfolge der Elemente des Bereiches `[first, last)` nach einer gleichmäßigen Verteilung. |
| `void random_shuffle(RandIt first, RandIt last, RandNumGen& rand);` | Änderung der Reihenfolge der Elemente des Bereiches `[first, last)` nach einer durch den Zufallszahlen-generator `rand` festgelegten Verteilung |
| `BidirIt partition(BidirIt first, , BidirIt last, Pred prd);` | Verschieben aller Elemente aus dem Bereich `[first,last)`, für die das Prädikat `prd` erfüllt ist, vor alle Elemente, für die es nicht erfüllt ist..<br>Rückgabe: Position des ersten Elements, für die das Prädikat `prd` nicht erfüllt ist |

- **Einige Mengen-Operationen**

| | |
|---|---|
| `OutpIt set_intersection (InpIt1 first1, InpIt1 last1, InpIt2 first2, InpIt2 last2, OutpIt res);` | Bildung einer sortierten Menge aus den Elementen, die sowohl in dem sortierten Bereich `[first1,last1)` als auch in `[first2, last2)` enthalten sind und Kopieren derselben in den bei `res` beginnenden Bereich.<br>(Bildung der Schnittmenge)<br>Rückgabe: Position hinter dem letzten Element im Zielber. |
| `OutpIt set_union(/* Parameter wie oben */);` | Bildung der Vereinigungsmenge |
| `OutpIt set_difference(/* Par. wie oben */);` | Bildung der Differenzmenge (Menge1 – Menge2) |

# Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (8)

- **Einfaches Demonstrationsprogramm (1) zu Algorithmen `algodem1`**

```
// C++-Quelldatei algo1dem1_m.cpp --> Programm algo1dem1
// Einfaches Demo-Programm zu Algorithmen der STL

#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
#include <iterator>
using namespace std;

template <class T>
  class ValOut
  { public: void operator()(const T& x) { cout << x << "  "; }
  };

template <class T>
  void checkFor(list<T>& lis, const T& val)
  { typename list<T>::iterator p;
    p=find(lis.begin(), lis.end(), val);
    cout << "gesucht : " << val ;
    if (p==lis.end()) cout << " und nicht gefunden" << endl;
    else              cout << " und gefunden : " << *p << endl;
  }

bool mod3(int x) { return x%3==0; }

int main(void)
{ list<int> li1;
  list<int>::difference_type cnt;
  list<int>::iterator ilp;
  li1.push_back(2);   li1.push_back(7);
  li1.push_back(9);   li1.push_back(3);
  li1.push_back(6);   li1.push_back(5);
  li1.push_back(15);  li1.push_back(11);
  ValOut<int> ivout;                         // Funktionsobjekt
  cout << "\nInhalt der Liste :\n";
  for_each(li1.begin(), li1.end(), ivout);
  cout << endl << endl;;
  checkFor(li1, 7);
  checkFor(li1, 8);
  cnt=count_if(li1.begin(), li1.end(),mod3);
  cout << "\nAnzahl durch 3 teilbarer Zahlen : " << cnt << endl;
  ilp=find_if(li1.begin(), li1.end(),mod3);
  cout << "Erste durch 3 teilbare Zahl     : " << *ilp << endl;
  ilp=max_element(li1.begin(), li1.end());
  cout << "Groesste enthaltene Zahl        : " << *ilp << endl;
  vector<int> vi1(li1.size());
  vector<int>::iterator ivp;
  copy(li1.begin(), li1.end(), vi1.begin());
  cout << "\nInhalt des Vektors :\n";
  for_each(vi1.begin(), vi1.end(), ivout);
  cout << endl << endl;
  int i=8;
  ivp=lower_bound(vi1.begin(), vi1.end(), i);
  cout << "Erste Zahl >= " << i << " (unsortiert) : " << *ivp << endl;
  sort(vi1.begin(), vi1.end());
  cout << "\nInhalt des Vectors nach Sortierung :\n";
  copy (vi1.begin(), vi1.end(), ostream_iterator<int>(cout, "  "));
  cout << endl << endl;
  ivp=lower_bound(vi1.begin(), vi1.end(), i);
  cout << "Erste Zahl >= " << i << " (sortiert)   : " << *ivp << endl;
  return 0;
}
```

# Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (8)

- **Ausgabe des Demonstrationsprogramms `algodem1`**

```
Inhalt der Liste :
2  7  9  3  6  5  15  11

gesucht : 7 und gefunden : 7
gesucht : 8 und nicht gefunden

Anzahl durch 3 teilbarer Zahlen : 4
Erste durch 3 teilbare Zahl     : 9
Groesste enthaltene Zahl        : 15

Inhalt des Vektors :
2  7  9  3  6  5  15  11

Erste Zahl >= 8 (unsortiert) : 15

Inhalt des Vectors nach Sortierung :
2  3  5  6  7  9  11  15

Erste Zahl >= 8 (sortiert)   : 9
```

# Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (9)

- **Einfaches Demonstrationsprogramm (2) zu Algorithmen `algodem2`**

```
// C++-Quelldatei algo1dem2_m.cpp --> Programm algo1dem2
// Einfaches Demo-Programm zu Algorithmen der STL

#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
#include <set>
#include <cstdlib>
#include <functional>
using namespace std;

class GenInt
{ public :
    GenInt (unsigned s=1) { srand(s);}
    int operator()(void) { return (int)(rand()%100);}
};


int main(void)
{ vector<int> vi1;
  vi1.push_back(2);    vi1.push_back(7);    vi1.push_back(9);
  vi1.push_back(3);    vi1.push_back(24);   vi1.push_back(6);
  vi1.push_back(15);   vi1.push_back(5);    vi1.push_back(11);
  ostream_iterator<int> iaus(cout, "  ");
  cout << "\nInhalt des Vektors :\n";
  copy (vi1.begin(), vi1.end(), iaus);   cout << endl << endl;
  set<int> si1;
  copy(vi1.begin(), vi1.end(), inserter(si1, si1.begin()));
  cout << "\nInhalt des Sets :\n";
  copy (si1.begin(), si1.end(), iaus);   cout << endl << endl;
  GenInt rnum(1111);
  vi1.resize(6);
  generate(vi1.begin(), vi1.end(), rnum);
  cout << "\nInhalt des Vektors nach Zufallszahlen-Zuweisung :\n";
  copy (vi1.begin(), vi1.end(), iaus);   cout << endl << endl;
  generate_n(back_inserter(vi1), 5, rnum);
  cout << "\nInhalt des Vektors nach Verlaengerung um Zufallszahlen :\n";
  copy (vi1.begin(), vi1.end(), iaus);   cout << endl << endl;
  sort(vi1.begin(), vi1.end());
  list<int> li1(si1.size()+vi1.size());
  merge(si1.begin(), si1.end(), vi1.begin(), vi1.end(), li1.begin());
  cout << "\nInhalt der Zusammenfassung des Sets und des sortierten Vektors "
          "(--> Liste) :\n";
  copy (li1.begin(), li1.end(), iaus);   cout << endl << endl;;
  list<int>::iterator ilp = unique(li1.begin(), li1.end());
  cout << "\nInhalt der Liste nach Entfernung aufeinanderfolgender Duplikate :\n";
  copy (li1.begin(), li1.end(), iaus);   cout << endl << endl;;
  li1.erase(ilp, li1.end());
  cout << "\nInhalt der Liste nach Entfernung uberfluessiger End-Elemente :\n";
  copy (li1.begin(), li1.end(), iaus);   cout << endl << endl;;
  list<int> li2;
  remove_copy_if(li1.begin(), li1.end(), back_inserter(li2),
                 not1(bind2nd(modulus<int>(), 2)));
  cout << "\nInhalt der 2. Liste nach entfernenden Kopieren :\n";
  copy (li2.begin(), li2.end(), iaus);   cout << endl << endl;;
  return 0;
}
```

# Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (10)

- **Ausgabe des Demonstrationsprogramms `algodem2`**

```
Inhalt des Vektors :
2  7  9  3  24  6  5  15  11


Inhalt des Sets :
2  3  5  6  7  9  11  15  24


Inhalt des Vektors nach Zufallszahlen-Zuweisung :
66  47  24  58  15  50


Inhalt des Vektors nach Verlaengerung um Zufallszahlen :
66  47  24  58  15  50  58  15  47  53  69


Inhalt der Zusammenfassung des Sets und des sortierten Vektors (--> Liste) :
2  3  5  6  7  9  11  15  15  15  24  24  47  47  50  53  58  58  66  69


Inhalt der Liste nach Entfernung aufeinanderfolgender Duplikate :
2  3  5  6  7  9  11  15  24  47  50  53  58  66  69  53  58  58  66  69


Inhalt der Liste nach Entfernung ueberfluessiger End-Elemente :
2  3  5  6  7  9  11  15  24  47  50  53  58  66  69


Inhalt der 2. Liste nach entfernenden Kopieren :
3  5  7  9  11  15  47  53  69
```