Programmieren / Algorithmen & Datenstrukturen 2

Container, Iteratoren und Algorithmen der StdLib

Prof. Dr. Skroch

Universitatea
BABEŞ-BOLYAI

Container, Iteratoren und Algorithmen der StdLib

Inhalt.

Container, Iteratoren und Algorithmen der StdLib

Grundsätzlicher Sinn: möglichst einheitlicher und damit einfacherer Umgang mit Datensequenzen.

- Programmiertechniken zielen meistens auch darauf, dass sie sich möglichst einheitlichen einsetzen lassen.
  - Der Umgang mit `int` ähnelt dem Umgang mit `double`,
  - `vector<int>` wird ähnlich wie `vector<double>` verwendet.
- Mit der StdLib kann man einheitlich
  - Daten gleichen Typs in sog. Containern sammeln, in denen sie als Sequenz von Elementen für den effizienten Zugriff organisiert sind,
  - gezielt auf bestimmte Datenelemente in den Containern zugreifen, etwa:
    - durch Index (z.B. das n-te Element),
    - durch Wert (z.B. das Elemente mit dem Wert `42`),
    - durch Eigenschaft (z.B. die Elemente mit `material == "Gold"`),
  - wichtige Operationen auf den Daten durchführen, etwa:
    - hinzufügen und löschen, sortieren und suchen,
    - grundlegende numerische Operationen (z.B. die Summe aller Elemente),

  ohne die Datentypen der Container, die Operationen für Zugriffsmechanismen oder die Algorithmen selbst implementieren zu müssen.

Beispiel für den Nutzen des Konzepts

Aufgabe: bilde die Summe einer Anzahl von Elementen.

- Ein Programm für die Summe von `double` Werten, die in einem Datenfeld gespeichert sind:

```
double sum( double array[], int n ) {
   double s{};
   for( int i{}; i<n; ++i ) s += array[i];
   return s;
}
```

- Ein Programm für die Summe von `int` Werten, die in einer einfach verketteten Liste gespeichert sind:

```
struct Node { Node* next{}; int data{}; };

int sum( Node* first ) {
   int s{};
   while( first ) {
       s += first->data;
       first = first->next;
   }
   return s;
}
```

Beispiel für den Nutzen des Konzepts

Aufgabe: bilde die Summe einer Anzahl von Elementen.

- Idee für ein allgemeines Programm (Pseudo-Quellcode):

```
sum( data_sequence ) {
  s = 0;                          // Initialisierung
  while( not_at_end( ) ) {        // Iterationsbedingung
    s += get_current_value( );
    get_next_element( );          // Iteration
  }
  return s;                       // Ergebnisrueckgabe
}
```

- Mit diesem Ansatz sind in gleicher Weise auch andere Basisaufgaben einfach lösbar, z.B. den größten Wert finden:

```
highest( data_sequence ) {
  s = 0, tmp = 0;                 // Initialisierung
  while( not_at_end( ) ) {        // Iterationsbedingung
    tmp = get_current_value( );
    if( s < tmp ) s = tmp;
    get_next_element( );          // Iteration
  }
  return s;                       // Ergebnisrueckgabe
}
```

Beispiel für den Nutzen des Konzepts

Aufgabe: bilde die Summe einer Anzahl von Elementen.

- Rudimentärer Quellcode im StdLib-Stil für ein allgemeines Programm:

```
template<class I, class T> T mysum( I first, I last, T s ) {
   while( first != last ) { s += *first; ++first; }
   return s;
}
```

- Das obige, rudimentäre Programm ist eine vereinfachte Version von `accumulate()` aus der StdLib.
- Anwendung
  - z.B. für **`double`** Werte in einem *Datenfeld*

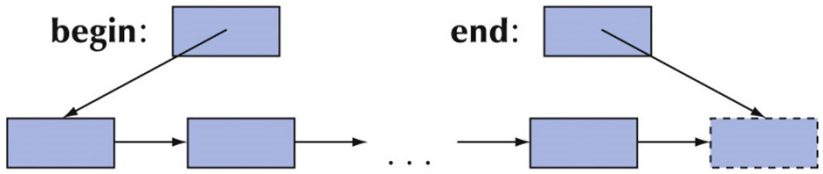
    ```
    double a[] { 2.1, 4.2, 6.3, 1.4, 3.5, 5.6 };
    double d { mysum( a, a+sizeof(a)/sizeof(*a), double{} };
    ```

    Probieren Sie das mit obiger Templatefunktion aus!
  - Z.B. für **`int`** Werte in einer *verketteten Liste*, beispielhafte Implementierung folgt...
  - Die StdLib stellt viele derartige Lösungen vordefiniert bereit...

Sequenzen und Iteratoren

Die StdLib betrachtet Datensammlungen als Sequenzen.

- Sequenzen sind das zentrale Konzept,
  - sie haben einen Anfang und ein Ende,
  - können von Anfang bis Ende vollständig und lückenlos durchlaufen werden,
  - und Werte der einzelnen Elemente aus der Sequenz können gelesen / geschrieben werden.
- Ein Iterator ist ein zeigerähnliches Objekt, das ein Element in einer Sequenz identifiziert.
  - Die beiden wichtigsten Iteratoren:
    - *begin* identifiziert den Anfang der Sequenz (falls vorhanden), ist Teil der Sequenz.
    - *end* identifiziert das Ende der Sequenz durch Verweis *direkt hinter* das letzte Element, ist also *nicht mehr* Teil der Sequenz.



    - "Mathematisch": [b; e [

Bildquelle: Stroustrup.

Sequenzen und Iteratoren

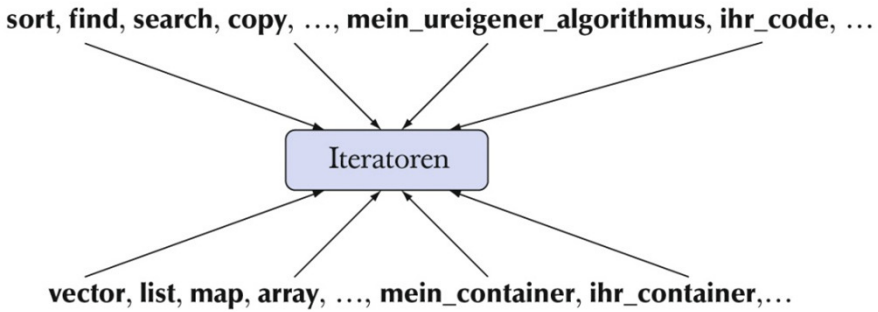
Iteratoren sind ein abstraktes Zugriffskonzept, man kann sich einen Iterator als eine Art Zeiger vorstellen.

- Ein Iterator verweist auf ein Element einer Sequenz (oder direkt hinter das letzte Element der Sequenz).
- Ein Iterator ist ein Typ, auf dessen Objekten man mindestens folgende Operationen anwenden kann:
  - `p == q` — `true` wenn beide Iteratoren `p`, `q` auf das selbe Element verweisen.
  - `p != q` — `!(p==q)`.
  - `*p` — Zugriff auf das Element, auf welches `p` verweist:
    - `*p = rv` — schreibt in das Element, auf welches `p` verweist (L-Wert),
    - `lv = *p` — liest aus dem Element, auf welches `p` verweist (R-Wert).
  - `++p` — Lässt den Iterator `p` auf das nächste Element verweisen.
- Für manche Iteratoren kann man weitere Operationen anwenden, beispielsweise `--`, `+`, `[]` .

Algorithmen, Sequenzen und Iteratoren

Iteratoren sind die Bindeglieder zwischen Containern und Algorithmen.

- *Algorithmen* manipulieren Daten in Containern, ohne sich darum zu kümmern, wie die Container diese Daten speichern.
- *Container* speichern Daten so, dass sie von *Iteratoren* als Sequenzen durchlaufen werden können, ohne sich um die Algorithmen zu kümmern, die mit diesen Daten arbeiten.
- C++11 StdLib stellt ca. 16 Container mit ihren Iteratoren, dazu ca. 90 Algorithmen bereit.



- Eigener Quellcode kann (und *soll*)
  - statt Datenspeicherung und -zugriff aufwändig und proprietär selbst zu implementieren, mit den normierten Containern und Iteratoren arbeiten,
  - und für typische Aufgaben normierte, vordefinierte Algorithmen einsetzen, statt diese aufwändig und proprietär selbst zu implementieren.

Bildquelle: Stroustrup.

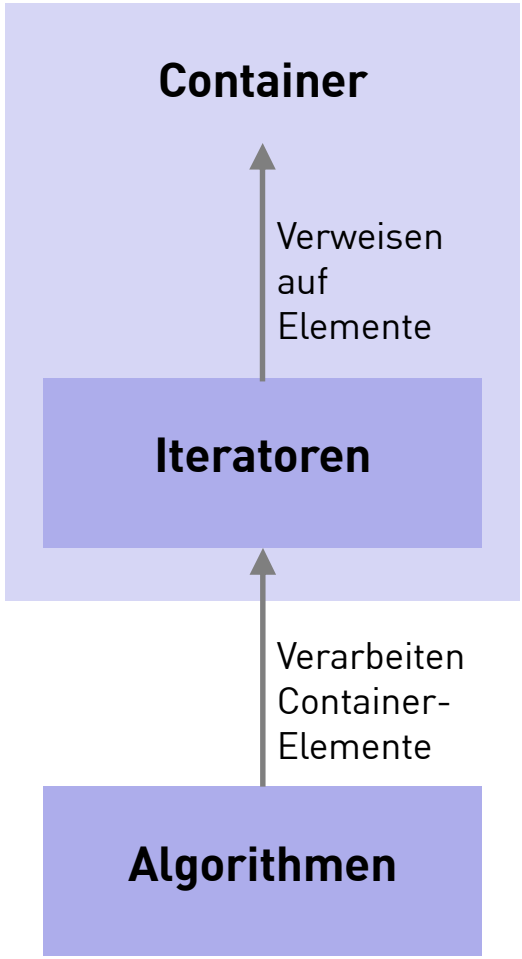
Container, Iteratoren und Algorithmen

Ein sehr nützlicher Teil der StdLib.

- *Container* nehmen Elementfolgen von beliebigem Typ auf.
  Beispiele:
  vector, list, set, map

- *Iteratoren* ermöglichen den Zugriff auf die Containerelemente.
  input, output, forward,
  bidirectional, random access

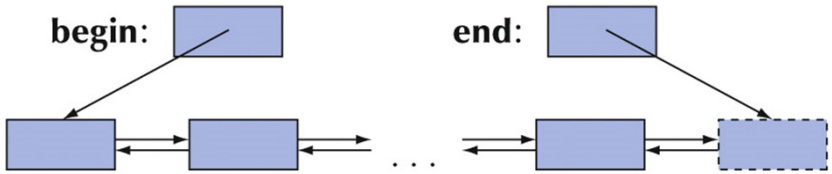
- *Algorithmen* verarbeiten die Elemente im Container.
  Beispiele:
  count(), sort(), copy_if();
  iota(), accumulate(), inner_product(),
  partial_sum(), adjacent_difference()



Verkettete Listen

Unsere doppelt verkettete Liste ist eine vereinfachte Version des `list` Containertyps aus der StdLib: zunächst die Datenstruktur für die Knoten...

- So kann man sich eine doppelt verkettete StdLib Liste bildlich vorstellen:



- So kann man die Knoten einer doppelt verketteten Liste im Quellcode repräsentieren:

```
template<class Elem> struct myNode {  // "fuer alle Typen Elem"
  Elem val;                           // der Wert vom Typ Elem
  myNode<Elem>* pre;                  // Zeiger auf vorhergehenden Knoten (predecessor)
  myNode<Elem>* suc;                  // Zeiger auf nachfolgenden Knoten (successor)
  explicit myNode( const Elem& v=Elem{},
                   myNode<Elem>* p=nullptr,
                   myNode<Elem>* s=nullptr ); // Standardkonstruktor
};
// Definition des Standardkonstruktors
template<class Elem>
myNode<Elem>::myNode( const Elem& v, myNode<Elem>* p, myNode<Elem>* s )
  : val{ v }, pre{ p }, suc{ s } {}
```

Bildquelle: Stroustrup.

Verkettete Listen

Unsere doppelt verkettete Liste ist eine vereinfachte Version des `list` Containertyps aus der StdLib: dann der Listen-Container...

- So kann man eine doppelt verkettete Liste als Datenstruktur im Quellcode repräsentieren:

```
template<class Elem> class myList {
   myNode<Elem> first; // Waechterknoten: Listenanfang
   myNode<Elem> last;  // Waechterknoten: Listenende
   unsigned int sz;    // Anzahl der Listenknoten
   // …
};
```

- Standardkonstruktor und Destruktor für den `myList` Typ:

```
template<class Elem> myList<Elem>::myList( )
: first{ myNode<Elem>{} }, last{ myNode<Elem>{} }, sz{0U}
{
  first.suc = &last; last.pre = &first;
}

template<class Elem> myList<Elem>::~myList( ) {
   myNode<Elem>* dp { first.suc };
   myNode<Elem>* tmp { nullptr };
   while( dp != &last ) {
      tmp = dp->suc;
      delete dp;
      dp = tmp;
   }
}
```

Verkettete Listen

Unsere doppelt verkettete Liste ist eine vereinfachte Version des `list` Containertyps aus der StdLib: einige Container-Operationen...

```
template<class Elem> myNode<Elem>* myList<Elem>::push_back( const Elem& v )
{
    myNode<Elem>* nn { new myNode<Elem>{v} };
    nn->suc = &last;
    nn->pre = last.pre;
    last.pre->suc = nn;
    last.pre = nn;
    ++sz;
    return nn;
}

template<class Elem> void myList<Elem>::pop_back( )
{
    myNode<Elem>* tmp = last.pre;
    if( tmp == &first ) return; // Liste ist schon leer
    last.pre = last.pre->pre;
    last.pre->suc = &last;
    delete tmp;
    --sz;
}

// usw. fuer alle weiteren Operationen
```

Iteratoren für verkettete Listen

Unsere doppelt verkettete Liste ist eine vereinfachte Version des `list` Containertyps aus der StdLib: der Iterator-Typ als Memberklasse...

- Iteratoren für den `myList` Typ:
  - Lesezugriff, Schreibzugriff (einfügen, löschen) usw. wollen wir nun über Iteratoren definieren.
  - Den Typ `myIterator` gestalten wir als Memberklasse der `myList` Klasse.
- Der `myIterator` Typ befindet sich innerhalb des `myList` Typs:

```
template<class Elem> class myList {
    // die gekapselten Datenmember
  public:
    myNode<Elem>* push_front( const Elem& ); // fuege als ersten Knoten ein
    void pop_front( );                       // loesche den ersten Knoten
    //…
    class myIterator;     // Membertyp namens myIterator, Klasse in einer Klasse
    myIterator begin( ); // Methode, die myIterator auf erstes Element liefert
    myIterator end( );   // liefert myIterator auf "eins hinter letztem Element"
    myIterator find( const Elem&, myIterator, myIterator ); // finden
    myIterator insert( const Elem&, myIterator );           // einfuegen
    myIterator remove( myIterator );                        // entfernen
    //…
};
```

Definition von `myList<Elem>::myIterator`

Unsere doppelt verkettete Liste ist eine vereinfachte Version des `list` Containertyps aus der StdLib: der Iterator-Typ als Memberklasse...

- Der `myIterator` Typ könnte so definiert werden:

```
template<class Elem> class myList<Elem>::myIterator {
    myNode<Elem>* curr; // der aktuelle Knoten
  public:
    myIterator( );
    explicit myIterator( myNode<Elem>* );
    myNode<Elem>* get_curr() const { return curr; }
    // die Iterator-Operationen:
    myIterator& operator++( );
    myIterator& operator--( ); // Container ist doppelt verkettete Liste
    Elem& operator*( );
    bool operator==( const myIterator& ) const;
    bool operator!=( const myIterator& ) const;
};

// Konstruktoren:
template<class Elem> myList<Elem>::myIterator::myIterator( )
  : curr{ nullptr } {}
template<class Elem> myList<Elem>::myIterator::myIterator( myNode<Elem>* p )
  : curr{ p } {}
```

Definition von `myList<Elem>::myIterator`

Unsere doppelt verkettete Liste ist eine vereinfachte Version des `list` Containertyps aus der StdLib: der Iterator-Typ als Memberklasse...

- Die `myIterator` Operatoren == und != können so definiert werden:

```
template<class Elem>
bool myList<Elem>::myIterator::operator==
    ( const myList<Elem>::myIterator& other ) const
{
    return curr == other.curr;
}
template<class Elem>
bool myList<Elem>::myIterator::operator!=
    ( const myList<Elem>::myIterator& other ) const
{
    return curr != other.curr;
}
```

- Aufgabe:
  - Definieren Sie den `myList<Elem>::myIterator` Typ vollständig.
  - Es fehlen v.a. noch ++, -- und *.

Mehrdeutigkeiten und die `typename` Syntax

Typen, Templates und vollqualifizierte Namen: `myIterator` ist ein Membertyp aus `myList<Elem>`.

- Für Namen, die von Template-Parametern abhängen und vollqualifiziert angegeben sind, kann der Compiler die möglichen Mehrdeutigkeiten nicht immer allgemeingültig auflösen.
- Ein Beispiel:

```
template<class T> class X {
  public:
    void method( ) { T::A* pa; /*…*/ } // zur Build-Time mehrdeutig
};
```

  - `A` könnte entweder einen benutzerdefinierten *Typ* benennen, der innerhalb des Typs `T` liegt (eine Memberklasse), d.h. die Anweisung würde `pa` als Zeiger auf ein Objekt des Typs `T::A` deklarieren.
  - Oder `A` könnte ein *Objekt* benennen, das innerhalb des Typs `T` liegt (z.B. eine `static` Membervariable), und die Anweisung wäre der binäre Operator `*` (Multiplikation?), der auf den beiden Operanden namens `T::A` und `pa` ausgeführt werden soll.
- Was genau der Fall ist wird ggf. erst zur Laufzeit des Programms klar, aber…

Mehrdeutigkeiten und die `typename` Syntax

Typen, Templates und vollqualifizierte Namen: `myIterator` ist ein Membertyp aus `myList<Elem>`.

- ...Templates werden schon beim Übersetzen des Programms aufgelöst (parametrische Polymorphie).
- Um das Problem zu lösen legt der C++ Standard fest: vollqualifizierte Namen, die von Template-Parametern abhängen, gelten nie von selbst als Typbezeichner.
  - Selbst dann nicht, wenn es syntaktisch keinen Sinn macht.
  - D.h. selbst wenn z.B. im obigen Quellcode-Fragment `T::A` als Name eines Objekts nicht existiert, wird `T::A` trotzdem vom Compiler als Name eines Objekts betrachtet (es folgt ein entsprechender Übersetzungsfehler).
- Falls ein solcher Name einen Typ bezeichnen soll, muss man es also ausdrücklich so programmieren, das Schlüsselwort lautet `typename`.
- Eine typischer Anwendungsfall für den Einsatz der `typename` Syntax ist das Beispiel mit unseren selbstgebauten Iteratoren: **`myIterator`** ist kein Objekt sondern ein Member*typ* aus **`myList<Elem>`** (vgl. nächste Seite).

Iteratoren für `myList<Elem>`

`begin()` und `end()`

- Der `myIterator`, den `begin()` liefert,
  - d.h. das erste Element in der Sequenz:

```
template<class Elem>
typename myList<Elem>::myIterator  myList<Elem>::begin( )
{
    return myIterator{ first.suc };
}
```

- Der `myIterator`, den `end()` liefert,
  - d.h. das erste Element *hinter* der Sequenz
  - (so kann man durch `begin()==end()` prüfen, ob der Container leer ist):

```
template<class Elem>
typename myList<Elem>::myIterator  myList<Elem>::end( )
{
    return myIterator{ &last };
}
```

Iteratoren für `myList<Elem>`

`find()`

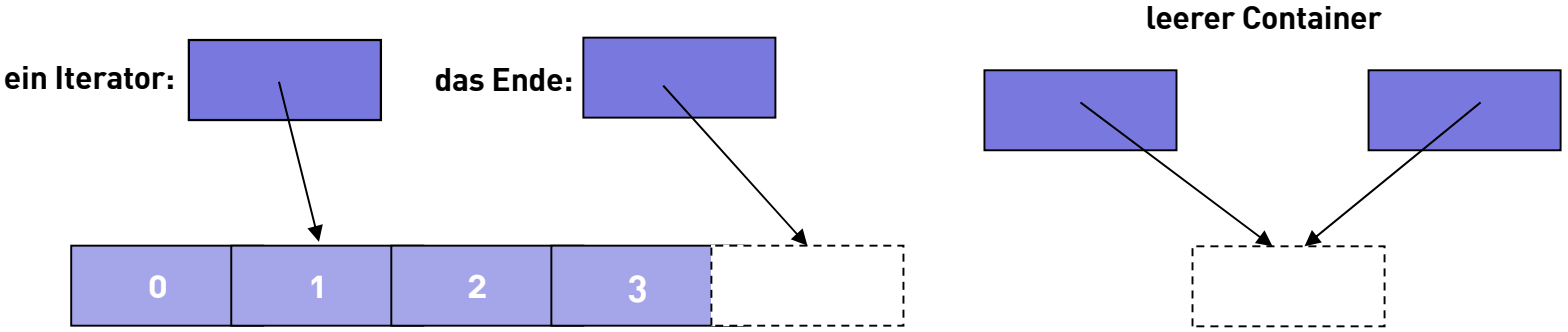
- `myIterator find( myIterator, myIterator, const Elem& )`
  - Finde das erste Element der Sequenz, das gleich einem gegebenen Wert ist.
  - Wird der Wert nicht gefunden, zeigt der Iterator auf das Element direkt hinter der Sequenz.

```
template<class Elem>
typename myList<Elem>::myIterator myList<Elem>::find
     ( myIterator p, myIterator q, const Elem& v )
{
     while( p!=q && *p!=v ) ++p;
     return p;
}
// Einsatzbeispiel:
void f( myList<long double>& mL )
{
    myList<long double>::myIterator it{};
    it = mL.find( mL.begin(), mL.end(), 1.248L );
    if( it != mL.end() ) cout << "found: " << *it;
}
```

Iteratoren und Algorithmen

`last()` ist direkt *hinter* der Sequenz.

- Ein Iterator zeigt auf (bezeichnet, referenziert) ein Element einer Sequenz.
- Das Ende der Sequenz befindet sich *direkt hinter* dem letzten Element.
  - Das Ende wird auch Sentinel (deutsch etwa "Wächter") genannt.
  - Vorsicht: "direkt-hinter-dem-letzten-Element" ist kein echtes Element.
    - Ein Iterator kann auf das Ende direkt hinter der Sequenz zeigen und mit einem anderen Iterator verglichen werden.
    - Das Ende enthält aber *keinen* Wert, der gelesen / geschrieben werden könnte.
- Wird das Element direkt nach dem letzten Element zurückgegeben, bedeutet das "nicht gefunden", "nicht erfolgreich", o.ä.



Bildquelle: Stroustrup.

Iterator–Kategorien der StdLib

Die C++ Standardbibliothek kennt fünf Kategorien von Iteratoren.

- Input:
  - Bewegt sich mit **++** vorwärts, liest mit ***** die Werte aus den Elementen.
  - Diese Art von Iterator wird z.B. von `istream` angeboten.
- Output:
  - Bewegt sich mit **++** vorwärts, schreibt mit ***** die Werte in die Elemente (wenn die Elemente nicht `const` sind).
  - Diese Art von Iterator wird z.B. von `ostream` angeboten.
- Forward:
  - Bewegt sich mit **++** vorwärts, schreibt (wenn die Elemente nicht `const` sind) und liest mit ***** die Werte der Elemente.
- Bidirectional:
  - Bewegt sich mit **++** vorwärts und mit **––** rückwärts, schreibt (wenn die Elemente nicht `const` sind) und liest mit ***** die Werte der Elemente.
  - Diese Art von Iterator wird z.B. von `list`, `map` und `set` angeboten.

Iterator–Kategorien der StdLib

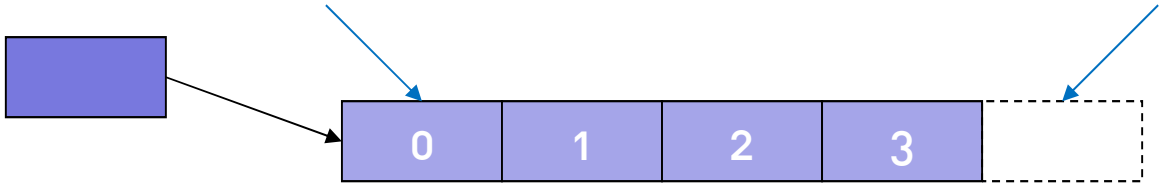
Die C++ Standardbibliothek kennt fünf Kategorien von Iteratoren.

- Random Access:
  - Bewegt sich mit **++** vorwärts und mit **––** rückwärts.
  - Schreibt (wenn die Elemente nicht `const` sind) und liest mit ***** die Werte der Elemente.
  - Schreibt (wenn die Elemente nicht `const` sind) und liest mit **[]** die Werte der Elemente.
  - Kann sich durch Addition oder Subtraktion eines Integer-Werts nach vorne oder hinten bewegen (Operatoren **+** und **–**).
  - Die Subtraktion zweier Random Access-Iteratoren, die auf Elemente in dem selben Container verweisen, ergibt den Abstand zwischen ihnen.
  - Diese Art von Iterator wird z.B. von `vector` angeboten.

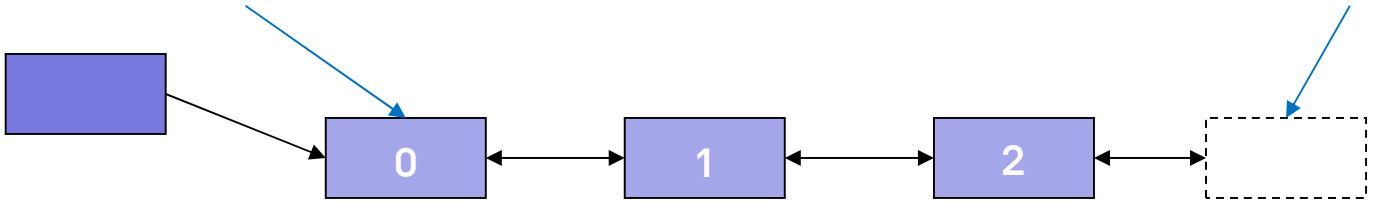
StdLib Container

Container beinhalten Elemente, die sequentiell durchlaufen werden können, und unterschiedlich implementiert sein können.

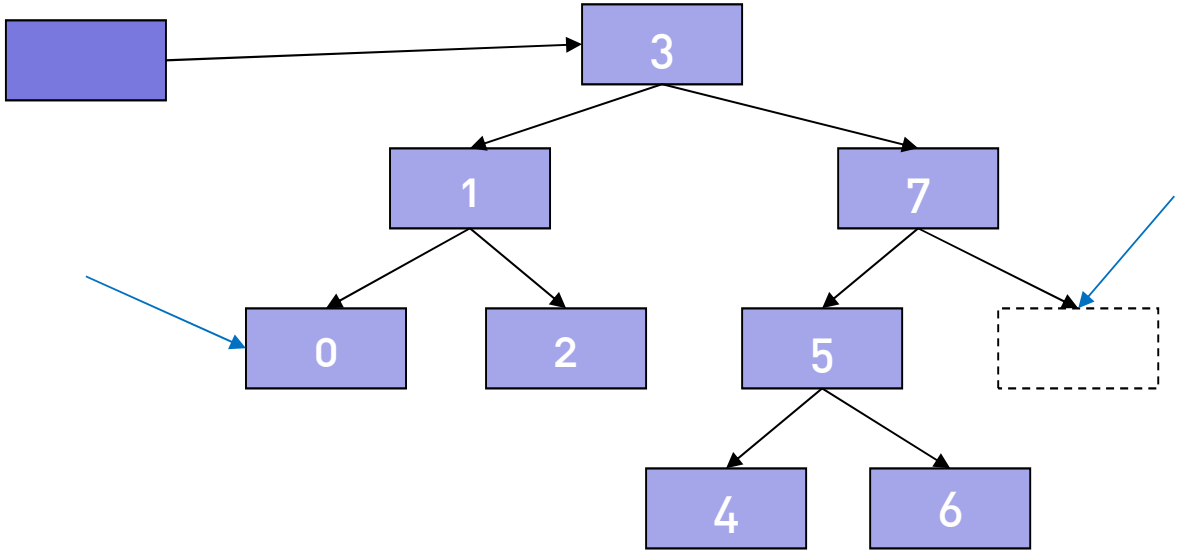
- `vector`



- `list`



- `set`



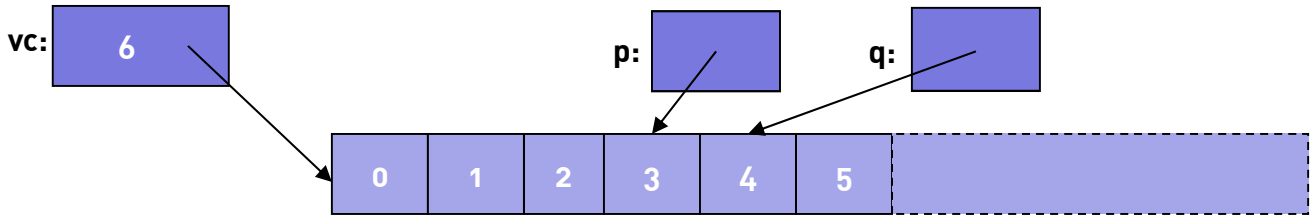
Bildquelle: Stroustrup.

StdLib: `insert()` und `vector` Container

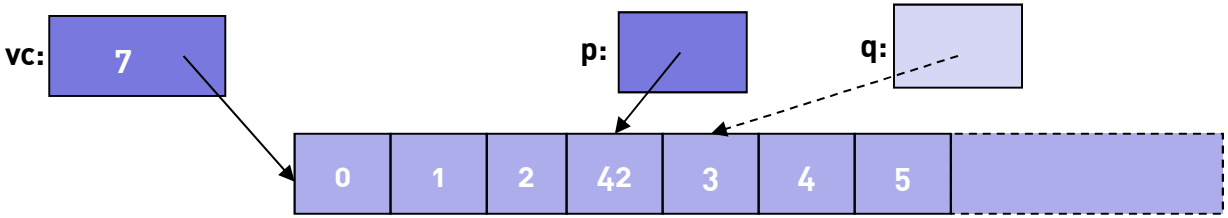
Bei Einfügeoperationen in `vector` Container müssen meist Elemente umkopiert werden.

- Beispiel:

```
vector<int>::iterator p{vc.begin()}; ++p; ++p; ++p;
vector<int>::iterator q{p}; ++q;
```




```
p = vc.insert( p, 42 );
```



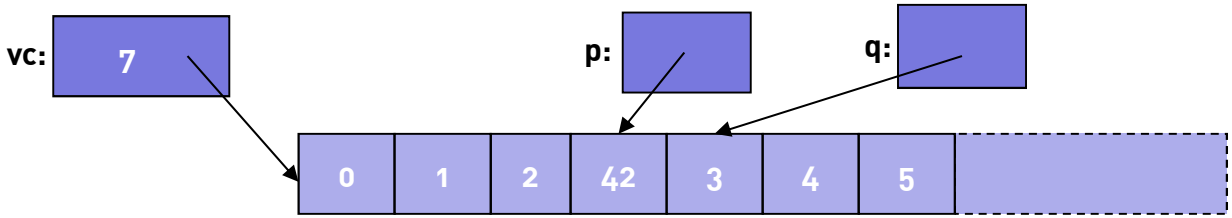
- `p` zeigt auf das eingefügte Element.
- `q` kann nicht mehr als gültig angenommen werden.
- Einige Elemente werden umkopiert (ggf. müssen sogar alle umkopiert werden).

Bildquelle: Stroustrup.

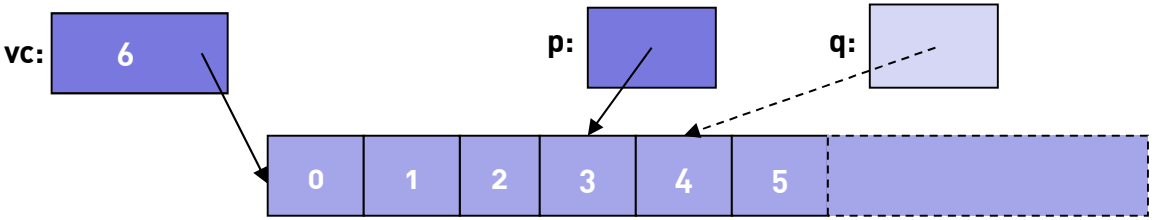
StdLib: `erase()` und `vector` Container

Auch bei Löschoperationen in `vector` Containern müssen meist Elemente umkopiert werden.

- Beispiel:




```
p = vc.erase( p );
```

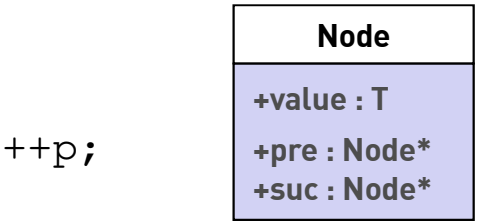


- `p` zeigt auf das Element nach dem gelöschten Element.
- `q` kann nicht mehr als gültig angenommen werden.
- Einige Elemente werden umkopiert (ggf. müssen sogar alle umkopiert werden).

Bildquelle: Stroustrup.

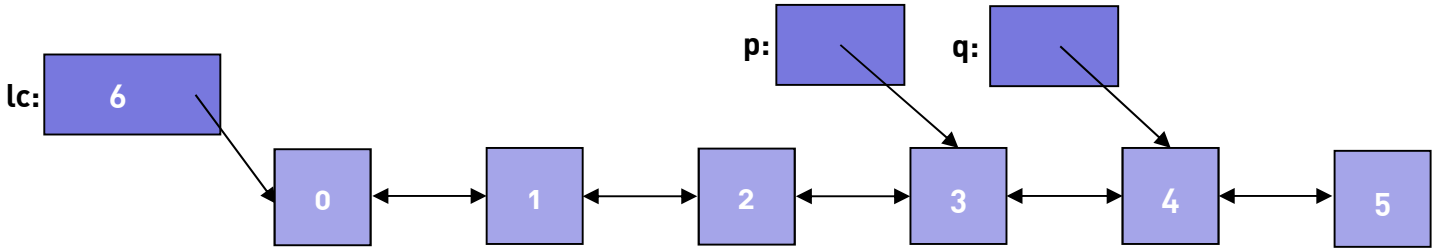
StdLib: `insert()` und `list` Container

Bei Einfügeoperationen in `list` Container werden keine Elemente umkopiert.

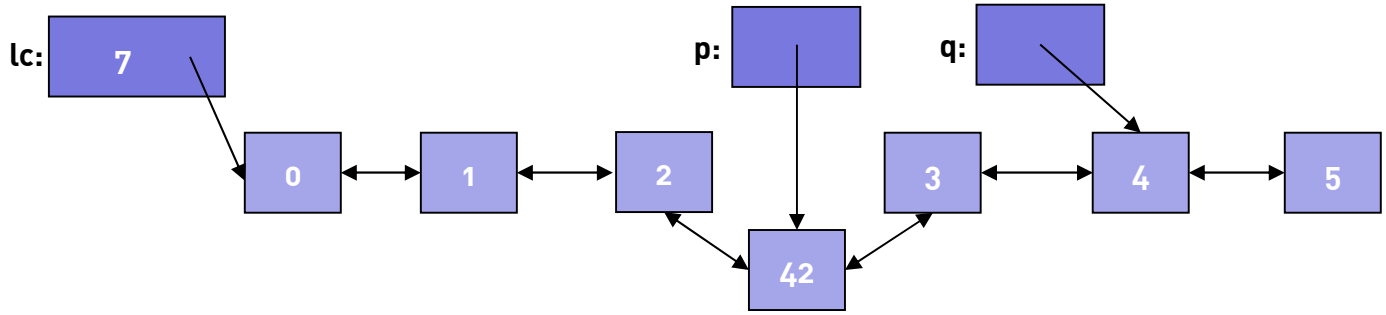


- Beispiel:

```
list<int>::iterator p{lc.begin()}; ++p; ++p; ++p;
list<int>::iterator q{p}; ++q;
```




```
p = lc.insert( p, 42 );
```



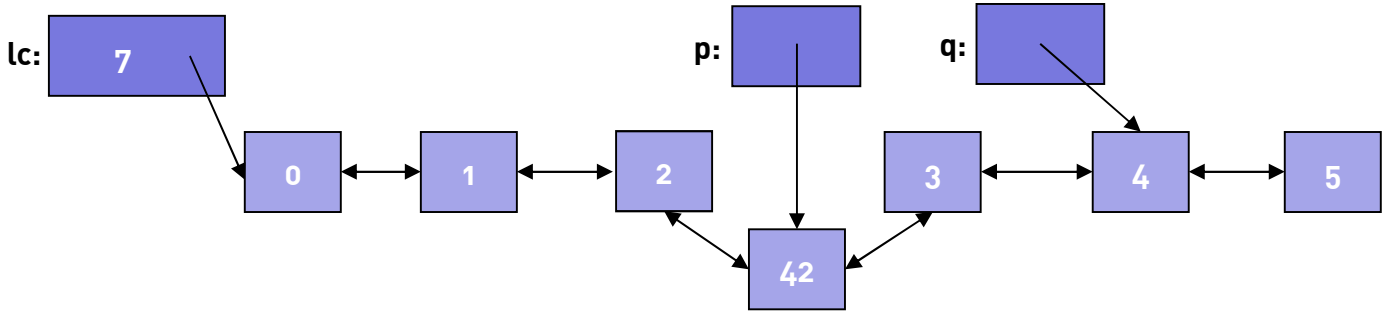
- `p` zeigt auf das eingefügte Element.
- `q` wird nicht verändert und bleibt gültig.
- Elemente werden nicht umkopiert.

Bildquelle: Stroustrup.

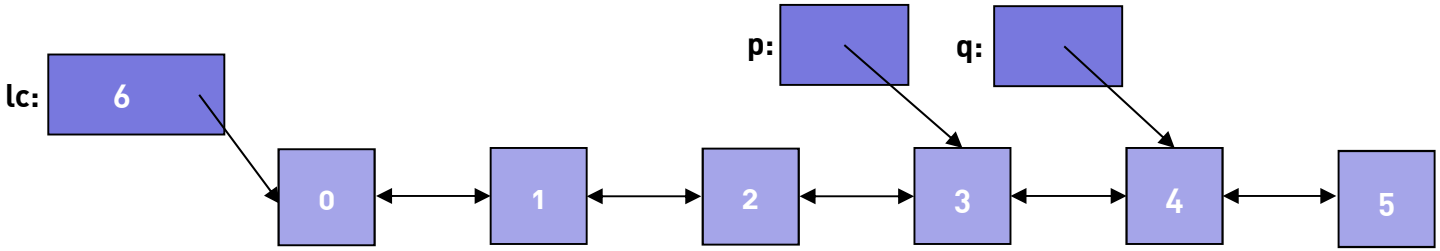
StdLib: `erase()` und `list` Container

Auch bei Löschoperationen in `list` Containern werden keine Elemente umkopiert.

- Beispiel:




```
p = lc.erase( p );
```



- `p` zeigt auf das Element nach dem gelöschten Element.
- `q` wird nicht verändert und bleibt gültig.
- Elemente werden nicht umkopiert.

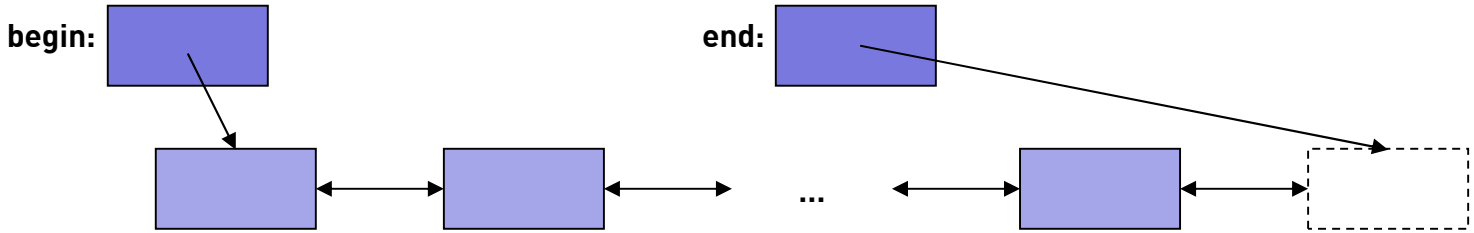
Bildquelle: Stroustrup.

Zusammenfassung

Das Modell der Iteratoren und Elementsequenzen ist das grundlegende Konzept der StdLib Container.

- Ein Iterator ist ein Typ, der mindestens die "Iterator-Operationen" für Container anbietet:
  - `++` zeige auf das nächste Element,
  - `*` liefere den Wert des Elements (Dereferenzierung),
  - `==` prüfe, ob zwei Iteratoren auf das selbe Element zeigen,
  - `!=` prüfe, ob zwei Iteratoren nicht auf das selbe Element zeigen.

- Ein Iteratoren-Paar definiert eine Elementsequenz, z.B.:
  - `begin` zeigt auf das erste Element (falls die Sequenz nicht leer ist),
  - `end` zeigt auf das Element direkt hinter dem letzten echten Element.



Bildquelle: Stroustrup.

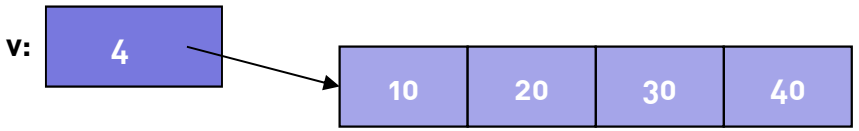
LEERE SEITE

Ein Grundalgorithmus: `myAccumulate()`

Kann die Elementwerte einer Sequenz aufaddieren.

- Idee (vgl. auch `mysum` weiter vorn)

```
template<class Iter, class T>
T myAccumulate( Iter first, Iter last, T init )
{
    while( first != last )
    {
        init = init + *first;
        ++first;
    }
    return init;
}
```




```
int s{ myAccumulate( v.begin(), v.end(), 50 ) };
// sum wird 150
// StdLib Header numeric fuer std::accumulate() notwendig
```

Ein Grundalgorithmus: `myAccumulate()`

Kann die Elementwerte einer Sequenz aufaddieren.

- Anwendungsbeispiel:

```
void f( vector<double>& vd, int* p, int n ) {

   // addiere die Elemente von vd:
   // der Typ des 3. Arguments bestimmt auch die Rechengenauigkeit
    double sum { myAccumulate( vd.begin(), vd.end(), 0.0 ) };

   // addiere die int Werte in ein int (Ueberlauf gut moeglich):
   // p+n kann man sich als &p[n] vorstellen
    int si { myAccumulate( p, p+n, 0 ) };

   // addiere die int Werte in ein long:
    long sl { myAccumulate( p, p+n, long{} ) };

   // addiere die int Werte in ein double:
    double s2 { myAccumulate( p, p+n, 0.0 ) };

   // die Variable für den Rueckgabewert kann im Initialisierer verwendet werden
    long double s3{};
    s3 = myAccumulate( vd.begin(), vd.end(), s3 );

}
```

Ein Grundalgorithmus: `myAccumulate()`

Verallgemeinerung für die Operation, die auf den Elementwerten der Sequenz ausgeführt wird.

- Anstelle der Summenbildung kann jede binäre Operation treten.
  - Jede, die den Wert von `init` aktualisiert.
- Idee:

```
template<class Iter, class T, class BinOp>
T myAccumulate( Iter first, Iter last, T init, BinOp op ) {
    while( first != last ) {
        init = op( init, *first ); // d.h. "init op *first"
        ++first;
    }
    return init;
}
```

Sehen Sie sich ggf. nochmals die *Operatoren und Operatorfunktionen* aus PAD1, Teil 7 der Grundlagen an

Ein Grundalgorithmus: `myAccumulate()`

Verallgemeinerung für die Operation, die auf den Elementwerten der Sequenz ausgeführt wird.

- Anwendungsbeispiel:

```
struct Article {
    int units;              // verkaufte Stueckzahl
    double unit_price;      // Stueckpreis
    Article( int u, double up ) : units{u}, unit_price{up} {}
    // …
};

double price( double v, const Article& r ) {
    return v + r.unit_price * r.units;
}

void f( const vector<Article>& vr ) {
  double total { myAccumulate( vr.begin(), vr.end(), 0.0, price ) }
}

// …
```

Ein Grundalgorithmus: `myAccumulate()`

Verallgemeinerung für die Operation, die auf den Elementwerten der Sequenz ausgeführt wird.

- Weiteres Anwendungsbeispiel, für Multiplikation:

```
#include <functional>
void f( list<double>& ld ) {
   double product { myAccumulate( ld.begin(),
                                  ld.end(),
                                  1.0,
                                  multiplies<double>{} ) };
   //…
}
```

Ein "*Funktor*" aus der StdLib zur Multiplikation

Funktoren

Funktoren sind Objekte, die sich wie Funktionen verhalten, aber ansonsten alle Eigenschaften von Objekten haben.

- Prinzipiell gilt: beim Aufruf einer Funktion tritt, nach Ablauf der Funktion, das von der Funktion zurückgegebene Ergebnis an die Stelle des Aufrufs.
- Die Aufgabe der Funktion als Ergebnislieferant kann auch von einem Objekt übernommen werden.
  - Dazu wird der Funktionsoperator `()` der Operatorfunktion des entsprechenden Typs überladen mit einer spezifisch definierten Operatorfunktion:
    ```
    operator()( /*Parameter*/ ){ /*Anweisungsblock*/ }
    ```
  - Objekte eines solchen Typs werden *Funktoren* genannt.
  - Der Funktionsadapter `std::bind()`, den wir schon eingesetzt hatten, gibt einen Funktor zurück.
- Wichtige Vorteile der Funktoren im Unterschied zu "traditionellen" Funktionen:
  - Funktoren können einen *Zustand* haben (d.h. Membervariablen besitzen).
  - Generalisierung / Vererbung ("ad-hoc Polymorphie") ist mit Funktoren möglich.
- In der StdLib werden häufig Funktoren verwendet.

Funktoren

Der `multiplies` Funktor aus dem Beispiel.

- Im eben besprochenen `myAccumulate()` Anwendungsbeispiel wurde die binäre Operation durch den `multiplies` Funktor realisiert.
- Der `multiplies` Funktor kann wie folgt definiert werden:

```
template <class T>
struct multiplies : binary_function <T,T,T> {
  T operator()( const T& x, const T& y ) const {
    return x*y;
  }
};
// Header: functional
```

```
template <class T1, class T2, class R>
struct binary_function {
  typedef T1 first_argument_type;
  typedef T2 second_argument_type;
  typedef R result_type;
};
```

Einstellige Prädikate

Als einstelliges Prädikat bezeichnet man in C++ eine Funktion oder einen Funktor mit genau einem Parameter und dem Ergebnistyp `bool`.

- Beispiel

```
bool odd( int i ) {  return i%2; } // Praedikat als Funktion
bool b { odd(42) }; // Aufruf der odd() Funktion:
                    // ist 42 ungerade?


class Odd { // Praedikat als Funktor
  public:
    bool operator()( int i ) const { return i%2; }
};
Odd oddNum{}; // ein Objekt namens oddNum vom Typ Odd
bool b { oddNum(42) }; // Aufruf des oddNum Funktors:
                       // ist 42 ungerade?
```

Kopieralgorithmen

Unterschiedliche StdLib Versionen des Kopierens können nützlich sein.

- **`copy( b, e, b2 )`**
  - Kopiert die Sequenz in [b; e [nach [$b2$; $b2+e–b$ [.
- **`unique_copy( b, e, b2 )`**
  - Kopiert die Sequenz in [b; e [nach [$b2$; $b2+e–b$ [.
  - Unterdrückt dabei benachbarte Duplikate.
- **`copy_if( b, e, b2, p )`**
  - Kopiert die Sequenz in [b; e [nach [$b2$; $b2+e–b$ [.
  - Übergeht dabei alle Elemente, die nicht dem Prädikat p entsprechen.

Kopieralgorithmen

Der StdLib `copy()` Algorithmus.

- Mit Hilfe eines Iterator-Paars kopiert `copy()` eine Sequenz in eine andere Sequenz, die durch einen weiteren Iterator auf ihr erstes Element identifiziert wird.

```
template<class In, class Out>
Out copy( In first, In last, Out res ) {
    while( first != last ) {
        *res = *first; ++res; ++first;
    }
    return res;
}
```

  - Einsatzbeispiel:

```
void f( vector<double>& vect_dest, const list<int>& list_source ) {
  if( vect_dest.size() < list_source.size() )
    error( "Zielcontainer zu klein" );
      // Achtung: der Zielcontainer muss genug Elemente haben,
      // um Kopien aller Elemente des Quellcontainers aufzunehmen
  copy( list_source.begin(), list_source.end(), vect_dest.begin() );
    // geht mit unterschiedlichen Container-Typen und unterschiedlichen Element-Typen
  sort( vect_dest.begin(), vect_dest.end() );
   // …
}
```

Kopieralgorithmen

Der StdLib `copy_if()` Algorithmus.

- `copy_if()` arbeitet wie `copy()`, kopiert jedoch nur die Elemente, die einem Prädikat entsprechen:

```
template<class In, class Out, class Pred>
Out copy_if( In first, In last, Out res, Pred p ) {
    while( first != last ) {
        if( p(*first) ) {
            *res = *first;
            ++res;
        }
        ++first;
    }
    return res;
}
```

Kopieralgorithmen

Ein Prädikat als Funktor mit `copy_if()`.

- Ein Prädikat namens `Smaller_than` ist als Funktor mit einem Attribut elegant programmierbar:

```
template<class T> class Smaller_than {
  public:
    Smaller_than( ) : val{} {};
    Smaller_than( const T& v ) : val{ v } {}
    bool operator()( const T& v ) const { return v < val; }
  private:
    T val;
};
```

- Einsatz des Funktors:

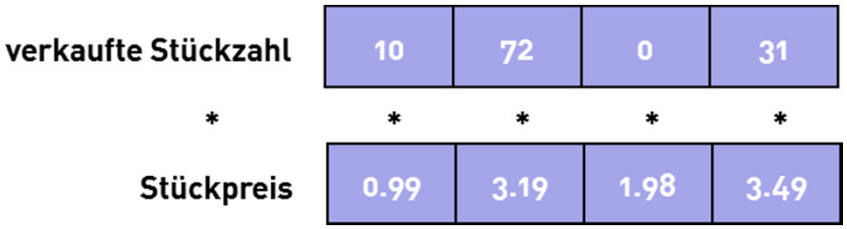
```
void f( const vector<int>& v ) {
   // kopiere alle Elemente, deren Wert kleiner als 27 ist
   vector<int> v2( v.size() );
   copy_if( v.begin(), v.end(), v2.begin(), Smaller_than<int>{27} );
   // …
}
```

Ein Grundalgorithmus: `myInnerProduct()`

Kann Elementwerte zu einem Skalar kombinieren.

- Idee:

```
template<class Iter, class Iter2, class T>
T myInnerProduct( Iter first,
                  Iter last,
                  Iter2 first2,
                  T init ) {
    // Multiplikation von zwei Sequenzen (zu einem Wert)
    while( first != last ) {
        // Multiplikation von Elementpaaren, und Summierung
        init = init + (*first) * (*first2);
        ++first;
        ++first2;
    }
    return init;
}
```



Ergebnis 347,77.
StdLib Header `numeric` für `std::inner_product()` erforderlich.

Wiederholung aus der Mathematik

Matrizenmultiplikation.

- Sei $A_{m,p}$ eine $m \times p$-Matrix und $B_{p,n}$ eine $p \times n$-Matrix.
  - $i = 1, \ldots, m$ Zeilen von A, $k = 1, \ldots, p$ Spalten von A.
  - $k = 1, \ldots, p$ Zeilen von B, $j = 1, \ldots, n$ Spalten von B.
- Dann ist die Matrizenmultiplikation von A und B definiert als:

$$AB = \begin{pmatrix} a_{11} & \cdots & a_{1p} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mp} \end{pmatrix} \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{p1} & \cdots & b_{pn} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + \cdots + a_{1p}b_{p1} & \cdots & a_{11}b_{1n} + \cdots + a_{1p}b_{pn} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{11} + a_{m2}b_{21} + \cdots + a_{mp}b_{p1} & \cdots & a_{m1}b_{1n} + \cdots + a_{mp}b_{pn} \end{pmatrix}$$

  - Die Elemente aus A (linker Operand) werden zeilenweise betrachtet und die Elemente aus B (rechter Operand) werden spaltenweise betrachtet.
  - Jeweils die erste, zweite,... Zeile von A wird mit der ersten, zweiten, ... Spalte von B kombiniert:
    - Multiplikation des ersten, zweiten, ... Elements einer A-Zeile mit dem ersten, zweiten,... Element der korrespondierenden B-Spalte,
    - Addition dieser Produkte zu dem Element an der entsprechenden Zeilen- und Spaltenposition.
- Das Ergebnis AB ist eine $m \times n$-Matrix.

Wiederholung aus der Mathematik

Skalarprodukt.

- Vektoren können als (eindimensionaler) Sonderfall der (zweidimensionalen) Matrizen angesehen werden.
  - Eine $n \times 1$-Matrix $\boldsymbol{V_{n,1} = V_n}$ heißt Spaltenvektor mit n Elementen.
  - Entsprechend heißt die $1 \times n$-Matrix $\boldsymbol{V_{1,n} = V_n^{\mathrm{T}}}$ Zeilenvektor mit n Elementen.
    - T steht hier für "transponiert": man transponiert eine Matrix, indem man die Zeilen der Reihe nach als Spalten schreibt (bzw. die Spalten der Reihe nach als Zeilen).
- Seien $V_n^{\mathrm{T}} = (v_1, v_2, \ldots, v_n)$ und $W_n^{\mathrm{T}} = (w_1, w_2, \ldots, w_n)$ Vektoren mit der gleichen Anzahl n von Elementen , dann gilt:

$$\boldsymbol{V_n^{\mathrm{T}} W_n = v_1 w_1 + v_2 w_2 + \cdots + v_n w_n}$$

  - Das Ergebnis der Berechnung ist eine Zahl, die besonders in der Matrizenalgebra auch als *Skalar* ("nulldimensionaler" Wert) bezeichnet wird,
  - manchmal auch "Skalarprodukt" oder "inneres Produkt" genannt.

Ein Grundalgorithmus: `myInnerProduct()`

Kann Elementwerte zu einem Skalar kombinieren.

- Anwendungsbeispiel:

```
// berechne den DAX
vector<double> dax_company_shareprice{};   // Aktienkurs jeder Firma
dax_company_shareprice.push_back( 36.11 );
dax_company_shareprice.push_back( 9.13 );
dax_company_shareprice.push_back( 122.10 );
// …
vector<double> dax_company_weight{};        // Gewichtung jeder Firma im DAX
dax_company_weight.push_back( 5.7727 );
dax_company_weight.push_back( 5.3815 );
dax_company_weight.push_back( 3.3586 );
// …
// Multiplikation (Aktienkurs*Gewichtung) und Addition
double dax { myInnerProduct( dax_company_shareprice.begin(),
                             dax_company_shareprice.end(),
                             dax_company_weight.begin(),
                             0.0 ) };
```

Ein Grundalgorithmus: `myInnerProduct()`

Verallgemeinerung für die Operationen, die auf den Elementwerten der Sequenz ausgeführt werden.

- Anstelle von Multiplikation und Summenbildung können auch hier beliebige binäre Operationen treten.
  - Alle, die den Wert von `init` aktualisieren.
- Idee:

```
template<class Iter, class Iter2, class T,
         class BinOp, class BinOp2>
T myInnerProduct( Iter first, Iter last, Iter2 first2, T init,
                  BinOp op, BinOp2 op2 ) {
    while( first != last ) {
        init = op( init, op2( *first, *first2 ) );
        ++first;
        ++first2;
    }
    return init;
}
```

LEERE SEITE

Assoziative Container

In der StdLib gibt es sog. assoziative Container, z.B.:

- `map` geordnete Sequenz von (*Schlüssel*, *Wert*)-Paaren
- `set` geordnete Sequenz von Schlüsseln
- `unordered_map` nicht geordnete Sequenz von (*Schlüssel*, *Wert*)-Paaren
- `unordered_set` nicht geordnete Sequenz von Schlüsseln

- `multimap` `map`, in der ein Schlüssel mehrfach vorkommen kann
- `multiset` `set`, in dem ein Schlüssel mehrfach vorkommen kann
- `unordered_multimap` nicht geordneter `multimap` Container
- `unordered_multiset` nicht geordneter `multiset` Container

- Während im `vector`-Container der Schlüssel (Index) immer vom Typ `int` ist, kann in assoziativen Containern so gut wie jeder Typ Schlüssel sein.
  - `RomeosPhoneNumbers["Julia"]` könnte die Handynummer von Julia Capulet sein.
  - Am Indexwert `"Julia"` vom Typ `std::string` findet Romeo den Wert der Telefonnummer.

Assoziativer Container `map`

`map` und `set` (eine Sonderform von `map`) sind sehr nützliche Container.

- Ein `map`-Container beinhaltet eine geordnete Sequenz von (*Schlüssel*, *Wert*)-Paaren, in der man anhand des Schlüssels nach dem Wert suchen kann.
  - Der `map`-Containertyp der StdLib ist in der Regel als *ausgeglichener binärer Suchbaum* implementiert.
- Ein `unordered_map`-Container ist ein `map`-Container, der für Zeichenketten als Schlüssel optimiert ist.
  - Die `unordered_map`-Containertyp der StdLib ist in der Regel als *Streuwert-Tabelle* implementiert.
- Ein `set`-Container (bzw. `unordered_set`-Container) ist ein `map`-Container (bzw. `unordered_map`-Container), dessen Elemente nur den Schlüssel, und keinen weiteren Wert, besitzen.

Assoziativer Container `map`

Anwendungsbeispiel.

```
void f() {
    map<string,int> words{}; // Worte und ihre Haeufigkeit
    string s{};
    while( cin>>s && s!="quit" )
        ++words[s]; // der Schluessel von words ist vom Typ string
                    // die int Werte von words werden mit 0 initialisiert
                    // wenn s schon enthalten ist, erhoeht map dessen Wert mit ++ um 1
                    // wenn s noch nicht enthalten ist, wird ein (s,0)-Paar
                    // eingefuegt und mit dem int-Operator ++ auf den Wert 1 erhoeht
                    // words[s] liefert int& als Anzahl des Worts s
   using citer = map<string,int>::const_iterator;
   for( citer p {words.cbegin()}; p != words.cend(); ++p )
           cout << p->first << ':' << p->second << '\n';
   return;
}
```

- Anmerkungen:
  - In `words[s]` ist `s` der Index vom Typ `std::string`.
  - `using` führt, zur besseren Lesbarkeit, den Alias `citer` als kürzeren Namen für `map<string,int>::const_iterator` ein.

Das `pair`-Template

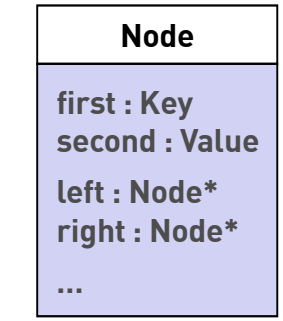
Die Elemente einer `map<string,int>` sind vom Typ `pair<string,int>`, im Header `utility` findet sich das `pair`-Template.

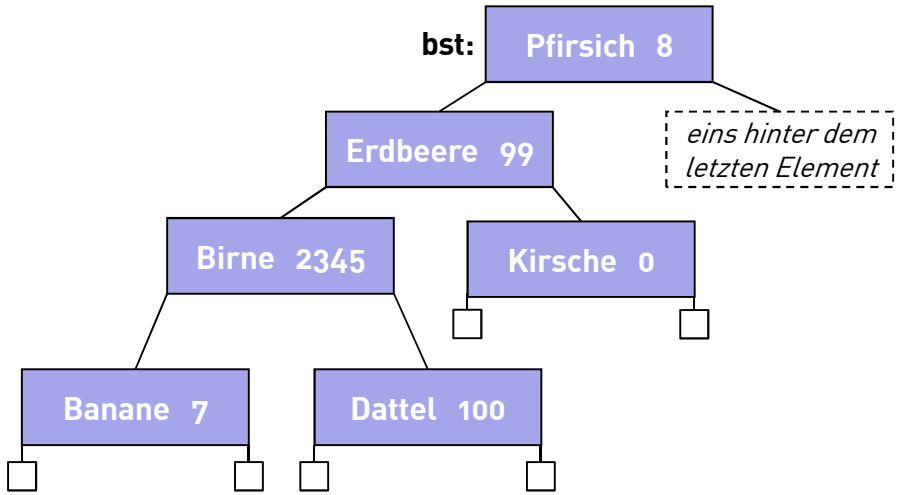
```
template<class T1, class T2> struct pair {
    T1 first;
    T2 second;
    pair()
      : first( ), second( ) {} // Standardkonstruktor
    pair( const T1& x, const T2& y )
      : first( x ), second( y ) {} // Konstruktor
    template<class U1, class U2>
    pair( const pair<U1,U2>& p )
      : first( p.first ), second( p.second ) {} // Kopierkonstruktor
    // usw.
};
// Nuetzliche Supportfunktion:
  template<class T1, class T2> pair<T1,T2> make_pair( T1 a, T2 b ) {
    return pair<T1,T2>(a,b);
  }
// Syntax, wenn man den Kopierkonstruktor nicht inline definieren moechte:
  template<class T1, class T2> template<class U1, class U2>
  pair<T1,T2>::pair( const pair<U1,U2>& p )
    : first( p.first ), second( p.second ) {}
```

Assoziativer Container `map`

`map` Implementierungen der StdLib sind in der Regel ausgeglichene binäre Suchbäume.

- Allgemein sind binäre Suchbäume Datenstrukturen, die ähnlich einer Liste aus Knoten aufgebaut sind.
  - Ein `map`-Knoten besteht aus einem Schlüssel, dem dazugehörigen Wert, und zwei Zeigern auf nachfolgende Knoten, links und rechts.
  - In jedem Knoten sind die Schlüssel des linken Teilbaums kleiner als der eigene Schlüssel.
  - In jedem Knoten ist der eigene Schlüssel kleiner oder gleich der Schlüssel des rechten Teilbaums.





- Das Beispiel zeigt, wie ein binärer Suchbaum (binary search tree) aussehen könnte, nachdem sechs Knoten eingefügt wurden, die als Schlüssel *Obstnamen* und als Wert *ganze Zahlen* enthalten.

Assoziativer Container `map`

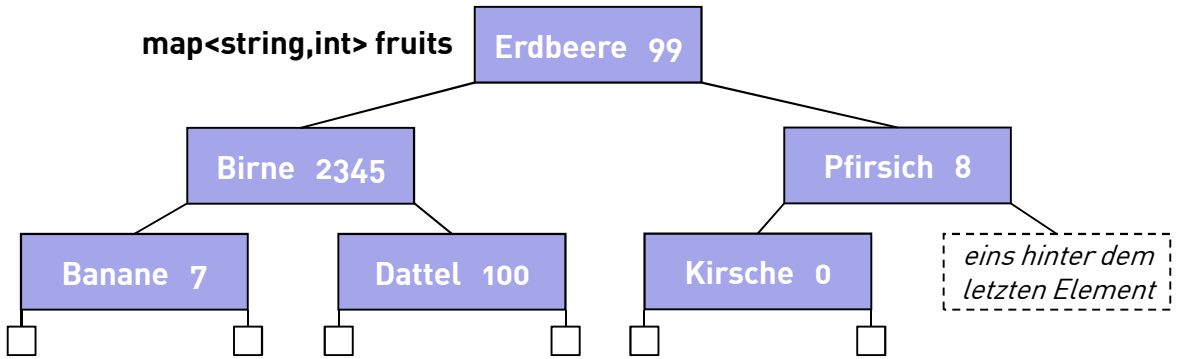
`map` Implementierungen der StdLib sind in der Regel ausgeglichene binäre Suchbäume.

- Die Methode, einen Schlüsselwert in einem `map`-Container zu suchen, ist sehr einfach, da die Knoten einer `map` nach der eben beschriebenen Vorschrift als binärer Suchbaum geordnet sind.
  - Man startet bei der Wurzel und wandert, immer die Knotenwerte mit dem Suchschlüssel vergleichend, durch den binären Suchbaum,
  - bis entweder der Wert gefunden ist (Suche erfolgreich) oder ein (leerer) Platz, an dem sich der Wert befinden müsste (Suche nicht erfolgreich).
- Es ist wünschenswert, dass *jede* Suche (besonders auch im ungünstigsten Fall) mit möglichst wenigen Vergleichen endet.
  - Das ist im vorigen Beispiel *nicht* der Fall:
    - Sie erinnern sich an die allgemeine binäre Suche: bei N Elementen reichen prinzipiell $1 + \lfloor \log_2(N) \rfloor$ binäre Vergleichsoperationen.
    - Um die Banane in den sechs Elementen zu finden sind aber 4 Vergleiche notwendig (obwohl 2^4 schon 16 ist, wir aber nur 6 Obstsorten haben).
- *Aufgabe:* wie wäre ein binärer Suchbaum mit den sechs Knoten aus dem vorigen Beispiel maximal unvorteilhaft aufgebaut?
  - Hinweis: im ungünstigsten Fall müssen dann *alle* Elemente besucht werden.

Assoziativer Container `map`

`map` Implementierungen der StdLib sind in der Regel ausgeglichene binäre Suchbäume.

- In *ausgeglichenen* binären Suchbäumen enthalten alle Knoten, die gleich weit von der Wurzel entfernt sind, ungefähr gleich viele Nachkommen.
  - Das Beispiel zeigt jetzt, wie eine `map<string,int>` Datenstruktur namens `fruits` als *ausgeglichener* binärer Suchbaum aussehen könnte, nachdem die sechs Knoten eingefügt wurden.
    - Ein `map`-Knoten kann zusätzliche Datenmember enthalten, die der `map` helfen, die Balance der Knoten im Baum sicherzustellen.



- Um einen beliebigen Knoten in einem solchen Baum mit N Knoten zu finden, müssen maximal $1 + \lfloor \log_2(N) \rfloor$ Knoten besucht werden.
- Mehr über binäre Suchbäume folgt...

Assoziativer Container `map`

Anwendungsbeispiel: Aktienkurse.

- DAX als `map`-Container:

```
// Deutscher Aktienindex, 3-Jan-2012
map<string,double> dax{}; // d.h. sinngemaess: pair(symbol,price)
dax["DAI"] = 36.11;
dax["DTE"] = 9.13;
dax["VOW"] = 122.09;
// …
map<string,double> dax_weight{}; // d.h. sinngemaess: pair(symbol,weight)
dax_weight.insert( make_pair( "DAI", 5.7727 ) );
dax_weight.insert( make_pair( "DTE", 5.3815 ) );
dax_weight.insert( make_pair( "VOW", 3.3586 ) );
// …
map<string,string> dax_name{}; // d.h. sinngemaess: pair(symbol,name)
dax_name["DAI"] = "Daimler";
dax_name["DTE"] = "Deutsche Telekom";
dax_name["VOW"] = "Volkswagen";
// …
```

Assoziativer Container `map`

Anwendungsbeispiel: Aktienkurse.

- Einsatz z.B.:

```
double daimler_price { dax["DAI"] }; // Werte aus der map lesen
double volkswagen_price { dax["VOW"] };

if( dax.find( "SOW" ) != dax.end() ) // Eintrag suchen
  cout << "Software AG ist im DAX!\n";

// ueber die map Elemente iterieren:
using dax_iterator = map<string,double>::const_iterator;

for( dax_iterator p {dax.cbegin()}; p != dax.cend(); ++p ) {
  cout << p->first /* das Symbol im "Ticker" */ << '\t'
       << p->second << '\t'
       << dax_name.at(p->first) << '\n';
}
```

Assoziativer Container `map`

Anwendungsbeispiel: Aktienkurse.

- DAX berechnen und ausgeben:

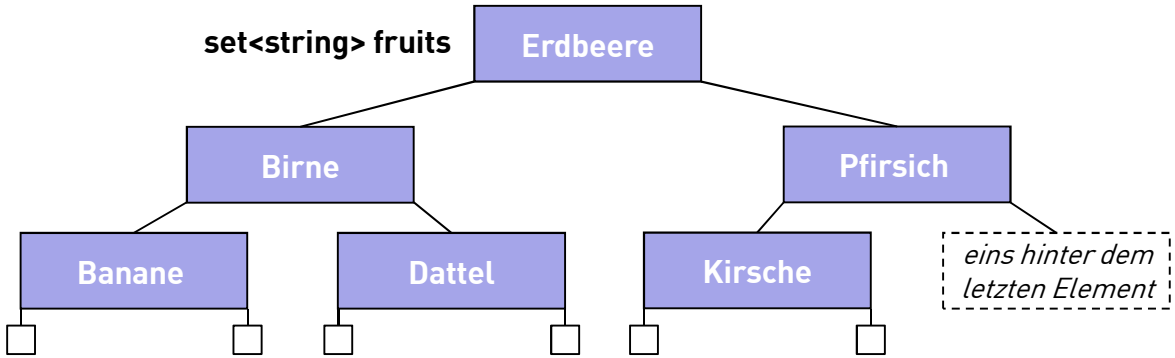
```
cout << inner_product( dax.begin(), dax.end(), // alle Firmen im DAX
                       dax_weight.begin(),     // deren Gewichtung
                       0.0,                    // Initialisierung mit 0.0
                       plus<double>{},         // addieren mit plus Funktor
                       value_product           // zuerst multiplizieren
                     );

double value_product( const pair<string,double>& a,
                      const pair<string,double>& b ) {
    return a.second * b.second;
}
```

```
template <class T> struct plus : binary_function <T,T,T> {
    T operator()( const T& x, const T& y ) const {
        return x+y;
    }
};
```

Assoziativer Container `set`

Auch `set` Implementierungen der StdLib sind in der Regel ausgeglichene binäre Suchbäume.

- `set`-Container sind `map`-Container, deren Elemente nur einen Schlüssel, und keinen weiteren Wert, besitzen.



- Auch `set`-Container der StdLib sind normalerweise als ausgeglichene binäre Suchbäume implementiert.

Assoziativer Container `unordered_map`

`unordered_map` Implementierungen der StdLib sind in der Regel Streuwert-Tabellen.

- In Streuwert-Tabellen werden Schlüsselwerte in Indizes transformiert.
  - Aus einem gegebenen Schlüsselwert (von "beliebigem" Typ `T`) wird ein ganzzahliger Wert (vom Typ `int` oder `T*`), der sog. *Streuwert* , berechnet.
    - Über den Streuwert soll dann auf eine geeignete Datenstruktur zugegriffen werden, oft ist der Streuwert ein Tabellenindex oder direkt eine Speicheradresse.
  - Die Berechnungsvorschrift, die aus Schlüsselwerten geeignete Streuwerte erzeugt, heißt *Streuwertfunktion*.
  - Die Datenstruktur, auf die mittels der Streuwerte zugegriffen werden kann, wird *Streuwert-Tabelle* genannt.
- In Streuwert-Verfahren werden Kollisionen zwischen gleichen Indizes beseitigt.
  - Das Streuwert-Verfahren muss damit umgehen können, dass die Streuwertfunktion aus unterschiedlichen Schlüsseln ggf. gleiche Streuwerte erzeugt.
    - Grund: die Anzahl der theoretisch möglichen Schlüsselwerte ist i.Allg. viel größer als die Anzahl der tatsächlich benötigten, und man möchte nicht unnötig große Tabellen vorhalten.
- Hauptvorteil von Streuwert-Tabellen:
  - Suchkomplexität von konstanter Größenordnung bzgl. der Anzahl der zu durchsuchenden Elemente.
- Streuwert-Tabellen werden oft für Zeichenketten-Schlüssel verwendet.
- Mehr über Streuwert-Tabellen folgt...

LEERE SEITE

Ein- und Ausgabe mit Dateien

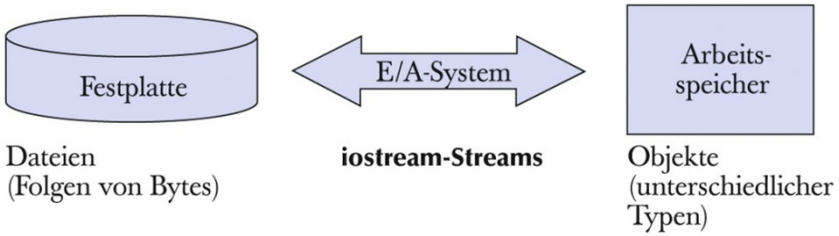
Dateien.

- Nach dem Abschalten des Computers sind die Daten im Arbeitsspeicher verloren.
  - Sog. transiente Daten.
- Deshalb speichert man alle Daten, die man weiterhin braucht, auf Festplatten und anderen permanenten Speichermedien.
  - Sog. persistente Daten.
- Betriebssysteme verwalten persistente Daten normalerweise als Dateien.
  - Eine Datei hat einen Namen.
  - Eine Datei ist üblicherweise eine Sequenz von persistent gespeicherten Bytes.
  - Eine Datei enthält ihre Daten in einem definierten Format, nach dem die gespeicherte Bytesequenz interpretiert werden kann.
- Um die Daten in einer Datei sinnvoll lesen und schreiben zu können, muss man den Dateinamen und das Dateiformat kennen.

Ein- und Ausgabe mit Dateien

Dateien.

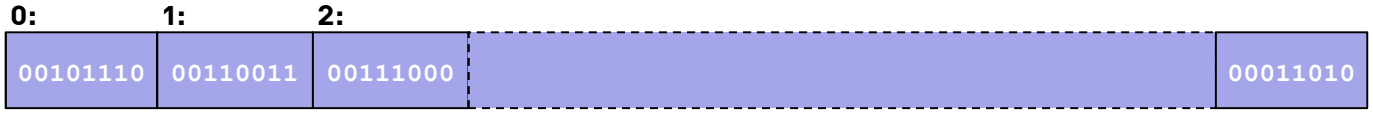
- Allgemeines Modell der Ein- und Ausgabe von und in Dateien in C++:



Sehen Sie sich nochmals das Kapitel zur *Ein- und Ausgabe* aus PAD1, L3 an

- Wir stellen uns eine Datei als benannte Sequenz von persistent gespeicherten Bytes vor, die mit Null beginnend durchnummeriert sind:

- Programme interpretieren die Bytefolgen im Sinne eines bestimmten Dateiformats:



  - Die drei ersten Bytes könnten als Gleitkomma-Literal `.38` interpretiert werden.

Bildquelle: Stroustrup.

Ein- und Ausgabe mit Dateien

Dateien lesen und schreiben.

- Ablauf beim Lesen aus einer Datei:
  - Den Name der zu lesenden, existierenden Datei kennen,
  - die Datei über ihren Namen "zum Lesen öffnen",
  - *die Daten lesen*,
  - und die Datei wieder "schließen".
- Ablauf beim Schreiben in eine Datei:
  - den Namen der zu (über-)schreibenden Datei kennen,
  - die Datei mit dem Namen "zum Schreiben öffnen",
  - *die Daten schreiben*,
  - und die Datei wieder "schließen".

Ein- und Ausgabe mit Dateien

Dateien zum Lesen öffnen.

```
// …
  cout << "Bitte den Namen der zu lesenden Datei eingeben: ";
  string name1{};
  cin >> name1; if( !cin ) error( "Kein Dateiname gelesen" );
  ifstream infile{ name1.c_str() };
  if( !infile ) error( "Kein Lesezugriff auf ", name1 );
// …
```

- `ifstream` ist ein Typ aus der StdLib für Eingabe-Ströme, die von einer Datei kommen.
  - Erzeugt man ein `ifstream` Objekt unter Angabe eines Dateinamens, dann wird die Datei dieses Namens zum Lesen geöffnet.
  - Die Datei wird geschlossen, wenn das `ifstream` Objekt seinen Scope verlässt (implizit mittels Destruktor).
- `c_str()` erzeugt eine `'\0'`-terminierte Zeichenkette ("C-Stil") aus einem Objekt vom C++ Typ `string`.
  - Die Systemschnittstelle zu vielen Betriebssystemen verlangt den Dateinamen als `'\0'`-terminierte Zeichenkette.

Ein- und Ausgabe mit Dateien

Dateien zum Schreiben öffnen.

```
// …
  cout << "Bitte den Namen der zu schreibenden Datei eingeben: ";
  string name2{};
  cin >> name2; if( !cin ) error( "Kein Dateiname gelesen" );
  ofstream outfile{ name2.c_str(), ios::app };
  if( !outfile ) error( "Kein Schreibzugriff auf ", name2 );
// …
```

- `ofstream` ist ein Typ aus der StdLib für Ausgabe-Ströme, die zu einer Datei gehen.
  - Erzeugt man ein `ofstream` Objekt unter Angabe eines Dateinamens, dann wird die Datei dieses Namens zum Schreiben geöffnet.
  - Die Datei wird geschlossen, wenn das `ofstream` Objekt seinen Scope verlässt (implizit mittels Destruktor).
- `ios::app` für den zweiten Parameter des Konstruktors bewirkt, dass die Ausgabe ans Ende der Datei angehängt wird, falls die Datei schon existiert.
  - Ohne `ios::app` kann eine ggf. existierende Datei überschrieben werden, d.h. die alten Inhalte sind verloren.

Ein- und Ausgabe mit Dateien

Dateien zum Schreiben öffnen.

- Wenn man nicht in bereits existierende Dateien schreiben möchte, kann man prüfen, ob die Datei bereits existiert.
  - Man versucht, sie zum Lesen zu öffnen.
  - Gelingt dies, gibt es die Datei schon.
  - Die Methode ist leider *nicht* absolut sicher.

```
// …
  cout << "Bitte den Namen der zu schreibenden Datei eingeben: ";
  string name{};
  cin >> name; if( !cin ) error( "Kein Dateiname gelesen" );

  if( ifstream{ name.c_str() } )
      error( "Datei existiert schon: ", name );

  ofstream of{ name.c_str() };
  if( !of ) error( "Kein Schreibzugriff auf ", name );
  // …
  // falls of dann noch weiter verwendet werden soll:
  of.close();
  of.clear();
// …
```

Ein- und Ausgabe mit Dateien

Öffnungsmodi für Dateien.

- Die Eigenschaften des Strom-Objekts bestimmen, welche Operationen man nach dem Öffnen der Datei ausführen kann und was sie bedeuten.
- Die Art und Weise des Zugriffs wird also maßgeblich festgelegt, wenn die Datei geöffnet und mit dem Strom-Objekt verbunden wird.
- Per Voreinstellung öffnet ein `ifstream`-Objekt seine Datei zum Lesen und ein `ofstream`-Objekt öffnet seine Datei zum Schreiben.
- Man kann unter folgenden Öffnungsmodi auswählen:

`app`	es wird am Ende der Datei weitergeschrieben (z.B. für Log-Dateien nützlich)
`ate`	es wird beim Öffnen direkt zum Dateiende gesprungen
`binary`	die Datei wird im sog. Binärmodus geöffnet
`in`	die Datei wird zum Lesen geöffnet
`out`	die Datei wird zum Schreiben geöffnet
`trunc`	stutzt die geöffnete Datei auf Nulllänge

```
ofstream log{ name, ios::app }; // ofstream verwendet per Voreinstellung out
fstream fs{ "myFile", ios::in | ios::out }; // sowohl in als auch out
```

- Unabhängig vom Öffnungsmodus ist das genaue Verhalten beim Umgang mit Dateien immer auch vom Betriebssystem abhängig.

Ein- und Ausgabe mit Dateien

Beispiel: Eingabedatei zeichenweise in Ausgabedatei kopieren.

```
// …
   string name1 { "ifile.txt" };
   string name2 { "ifile_dup.txt" };

   ifstream is { name1.c_str() };
   if( !is ) error( "Kein Lesezugriff auf ", name1 );

   if( ifstream { name2.c_str() } )
       error( "Datei existiert schon: ", name2 );

   ofstream os { name2.c_str() };
   if( !os ) error( "Kein Schreibzugriff auf ", name2 );

   char c {};
   while( is.get( c ) ) os.put( c );

   if( !is.eof() || !os ) { /* etwas ist schief gegangen */ }
// …


// um nicht zeichen- sondern blockweise zu kopieren:
//   os << is.rdbuf();
```

Am Dateiende wird `c` ein End-Of-File Zeichen enthalten und `get` wird die passende End-Of-File-Markierung (oft als `-1` definiert) an den `ifstream` zurückgeben.

Strom-Iteratoren

Auch für Ein- und Ausgabeströme sind in der StdLib Iteratoren definiert.

- Iteratoren für Ausgabeströme:
  - `ostream_iterator<T>` ist ein Template aus der StdLib, mit dem man Werte vom Typ `T` schreiben kann.

```
ostream_iterator<string> oo{ cout };
// an *oo zuweisen bedeutet, einen string nach cout zu schreiben
*oo = "Hallo, ";    // cout << "Hallo, "
++oo;               // zur naechsten Ausgabe-Operation
*oo = "C++!\n";     // cout << "C++!\n"
```

- Iteratoren für Eingabeströme:
  - `istream_iterator<T>` ist ein Template aus der StdLib, mit dem man Werte vom Typ `T` lesen kann.

```
istream_iterator<string> ii{ cin };
// aus *ii lesen bedeutet, einen string aus cin zu lesen
string s1 {*ii};    // cin >> s1
++ii;               // zur naechsten Eingabe-Operation
string s2 {*ii};    // cin >> s2
```

Eine Wortliste aus einer Textdatei erstellen

Unter Verwendung von Strom-Iteratoren, mit `unique_copy()` und einem `vector` Container.

```
// Fehlererkennung wird hier zur besseren Uebersicht weggelassen
// Dateinamen fuer Quell- und Zieldatei einlesen:
string from{}; string to{};
cin >> from >> to;

// Ein- und Ausgabe-Strom erstellen:
ifstream is{ from.c_str() }; ofstream os{ to.c_str() };

// Strom-Iterator zur Eingabe:
istream_iterator<string> in{ is };

// "end-of-stream" (wird standardmaessig auf EOF/end-of-file gesetzt)
// genauer: auf char_traits<Ch>::eof()
istream_iterator<string> eos{};

// Strom-Iterator zur Ausgabe, der jedesmal \n anhaengt:
ostream_iterator<string> out{ os, "\n" };

// Das vector-Objekt namens buffer wird mit dem Input initialisiert:
vector<string> buffer( in, eos );

// buffer wird in die Ausgabedatei kopiert, Duplikate werden entfernt:
sort( buffer.begin(), buffer.end() );
unique_copy( buffer.begin(), buffer.end(), out );
```

Eine Wortliste aus einer Textdatei erstellen

Unter Verwendung von Strom-Iteratoren, mit `copy()` und einem `set` Container.

```
// Fehlererkennung wird hier zur besseren Uebersicht weggelassen

// Dateinamen fuer Quell- und Zieldatei einlesen:
string from{}; string to{};
cin >> from >> to;

// Ein- und Ausgabe-Strom erstellen:
ifstream is{ from.c_str() }; ofstream os{ to.c_str() };

// Strom-Iterator zur Eingabe:
istream_iterator<string> in{ is };

// "end-of-stream" (wird standardmaessig auf EOF/end-of-file gesetzt):
// genauer: auf char_traits<Ch>::eof()
istream_iterator<string> eos{};

// Strom-Iterator zur Ausgabe, der jedesmal \n anhaengt:
ostream_iterator<string> out{ os, "\n" };

// Das set-Objekt namens buffer wird mit dem Input initialisiert:
set<string> buffer( in, eos );

// buffer wird in die Ausgabedatei kopiert (set ist bereits sortiert und ohne Duplikate)
copy( buffer.begin(), buffer.end(), out );
```

Übung

Eine Wortliste aus einer Textdatei erstellen, eine Zahlenreihe aus einer Datei einlesen.

- Erstellen Sie mit Hilfe der besprochenen Quellcodes eine Wortliste aus einem Text, der mindestens 100'000 Wörter lang ist.
- Lesen Sie die Zahlenreihe zur Geldmenge M3 ein und speichern Sie alles in einem geeigneten StdLib Container ab.

Ein- und Ausgabe mit Dateien

Binärdateien.

- Per Voreinstellung behandeln `iostream`-Ströme Daten immer als Objekte eines gewünschten Typs (wie `int` oder `string`), die typgerecht ein- bzw. ausgegeben werden sollen.
  - `istream` wandelt Zeicheneingaben in solche Objekte um.
  - `ostream` wandelt solche Objekte in Zeichenausgaben um.
- Man kann `istream`- und `ostream`-Ströme auch so einstellen, dass sie Daten einfach unverändert als rohe Bytes lesen und schreiben.
  - Das ist i.Allg. im Zusammenhang mit Dateien sinnvoll.
  - Diese sog. *binäre Ein- und Ausgabe* wird beim Öffnen einer Datei mittels des `ios::binary` Modus aktiviert.
- Merkmale der binären Speicherung von Daten (sog. *Binärdateien*):
  - Kompakt, weil die Daten nicht in menschenlesbaren Zeichen symbolisiert werden müssen.
  - Schnell, weil keine Formatierung beim Lesen oder Schreiben erfolgt.
  - (Daher sind Binärdateien auch nicht in einem Texteditor darstellbar oder druckbar.)

Ein- und Ausgabe mit Dateien

Beispiel: `vector` in Binärdatei schreiben / aus Binärdatei lesen.

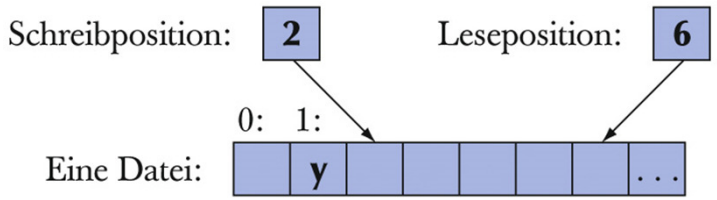
```
template<class T> char* as_bytes( T& i ) {
    void* addr = &i;
    return static_cast<char*>( addr );
}
```

```
// vector namens vd anlegen und fuellen
vector<double> vd{};
for( int i{}; i<100; ++i ) vd.push_back( i/100.0 );

// zur Illustration: Daten aus vd als lesbare Zeichen speichern
ofstream f1{ "val.txt" }; if( !f1 ) error( "Abbruch#1" );
for( int i{}; i<vd.size(); ++i )
    f1 << setprecision(2) << fixed << vd.at(i) << '\n';
f1.close(); f1.clear();

// Daten aus vd binaer speichern
f1.open{ "val.bin", ios::binary }; if( !f1 ) error( "Abbruch#2" );
for( int i{}; i<vd.size(); ++i )
    f1.write( as_bytes( vd.at(i) ), sizeof(double) );
f1.close(); f1.clear();

vector<double>{}.swap( vd ); // vd leeren, vd.clear() geht auch

// binaere Daten neu in vd einlesen
ifstream f2{ "val.bin", ios::binary };
if( !f2 ) error( "Abbruch#3" );
double d{};
while( f2.read( as_bytes( d ), sizeof(double) ) ) vd.push_back( d );
```

Ein- und Ausgabe mit Dateien

Festlegen der Schreib- und Leseposition in Dateien.

- Im Allgemeinen ist es ratsam, eine Datei nur sequentiell von Anfang bis Ende zu lesen oder zu schreiben.
- Es ist aber auch möglich, die Position in der Datei, von der gelesen bzw. an die geschrieben wird, zu bestimmen.
  - Jede Datei, die zum Lesen geöffnet wurde, besitzt eine Leseposition.
  - Jede Datei, die zum Schreiben geöffnet wurde, besitzt eine Schreibposition.



Aktuelle Leseposition abfragen:	`tellg()`
Aktuelle Schreibposition abfragen:	`tellp()`
Leseposition p aufsuchen:	`seekg( p )` oder `seekg( p, bez )`
Schreibposition p aufsuchen:	`seekp( p )` oder `seekp( p, bez )`

Mögliche Werte für den Bezugspunkt `bez` sind:
`ios::beg` (der Dateianfang)
`ios::end` (das Dateiende)
`ios::cur` (die aktuelle Position)

Bildquelle: Stroustrup.

Ein- und Ausgabe mit Dateien

Beispiel: Schreib- / Lesepositionierung in Dateien.

```
string name1 {"ifile.txt"};
ifstream sf{ name1.c_str(), ios::binary | ios::ate }; // source file
if( !sf ) error( "Kein Lesezugriff auf ", name1 );

long int nbytes{ sf.tellg() };
sf.seekg( ios::beg );

string name2 {"ifile_duplicate.txt"};
ofstream tf{ name2.c_str(), ios::binary }; // target file
if( !tf ) error( "Kein Schreibzugriff auf ", name2 );

tf << sf.rdbuf(); // blockweise kopieren

if( nbytes == tf.tellp() )
   cout << nbytes << " Byte erfolgreich kopiert";
else /* huch… */
   // Fehler beim Kopieren…
```

Übung

Binärdatei verarbeiten.

- Entschlüsseln Sie die Datei *shuffled* (die sich bei dem begleitenden Quellcode befindet).
- Hinweise:
  - Sehen Sie sich den Beispiel-Quellcode aus den Fragen im Kapitel über *Bitoperationen* aus PAD1, L3 nochmals an.
  - 883 und 4421 sind nette Primzahlen.

Einige Beispielfragen

Container, Iteratoren und Algorithmen der StdLib.

- Was verstehen Sie unter einem StdLib Container?
- Welches grundsätzliche Ziel verfolgt die StdLib mit ihren Containern?
- Erklären Sie das Konzept der Container, Iteratoren und Algorithmen mit eigenen Worten.
- Erläutern Sie, warum Sequenzen (Folgen von Elementen) in der StdLib grundlegend sind.
- Was ist ein Iterator? Welche Unterschiede und Gemeinsamkeiten sehen Sie zwischen Iteratoren und Zeigern?
- Welche Operationen kann man für alle StdLib-Iteratoren anwenden?
- Wodurch zeichnet sich der letzte Knoten in einer `myList<E>` aus?
- Nennen Sie die fünf Iterator-Kategorien der StdLib.
- Welchen entscheidenden Unterschied sehen Sie zwischen dem Einfügen und Löschen von Elementen in einen `vector` Container und in einen `list` Container?

Einige Beispielfragen

Container, Iteratoren und Algorithmen der StdLib.

- Welche Mehrdeutigkeit kann bei Templates, verschachtelten Klassen und vollqualifizierten Namen entstehen? Wie ist diese Mehrdeutigkeit im C++ Standard gelöst? Welche Syntax verwendet man, wenn man sich eindeutig auf einen Typ beziehen will?
- Erläutern Sie den Grundalgorithmus `std::myAccumulate()`.
- Erläutern Sie den Grundalgorithmus `std::myInnerProduct()`.
- Was ist ein Skalarprodukt?
- Was ist ein Funktor? Welche Vorteile hat ein Funktor im Vergleich zu einer Funktion? Welche Nachteile?
- Welchen wesentlichen Unterschied sehen Sie zwischen den Schlüsseln eines `vector` Containers und denen eines assoziativen Containers?
- Welche wesentliche Gemeinsamkeit sehen Sie zwischen den Schlüsseln eines `vector` Containers und denen eines assoziativen Containers?
- Welche Datenstruktur wird gewöhnlich zur Implementierung eines `map`-Containers verwendet? Warum?

Einige Beispielfragen

Container, Iteratoren und Algorithmen der StdLib.

- Erklären Sie das `pair`-Template aus der StdLib.
- Erklären Sie den entscheidenden Vorteil eines ausgeglichenen binären Suchbaums (im Unterschied zu einem nicht ausgeglichenen).
- Demonstrieren Sie an einem *eigenen* Beispiel den ungünstigsten Fall für einen nicht ausgeglichenen binären Suchbaum. Wie viele Knoten werden bei einer Suche im ungünstigsten Fall besucht? Wie viele werden durchschnittlich besucht?
- Welchen wesentlichen Unterschied sehen Sie zwischen `set` und `map`?
- Welche Datenstruktur wird gewöhnlich zur Implementierung eines `unordered_map` Containers verwendet? Warum?
- Erläutern Sie die Idee der Streuwert-Tabellen.
- Welchen Vorteil besitzen Streuwert-Tabellen?
- Geben Sie je ein eigenes Beispiel für den Einsatz von `copy()`, `unique_copy()` und `copy_if()`.

Einige Beispielfragen

Container, Iteratoren und Algorithmen der StdLib.

- Definieren Sie ein Prädikat namens `Is_vocal` mit der offensichtlichen Bedeutung (ein beliebiger Buchstaben, wie z.B. `E` oder `q` soll möglich sein) als Funktor.
- Definieren Sie ein Prädikat als Funktortemplate namens `Equals` mit der offensichtlichen Bedeutung.
- Informieren Sie sich über die Strom-Iteratoren der StdLib und fassen Sie für sich die wichtigsten Punkte zusammen.
- Informieren Sie sich über die Algorithmen aus dem Header `numeric` und fassen Sie für sich die wichtigsten Punkte zusammen.
- Wie kann man mit Strom-Iteratoren eine Datei lesen bzw. schreiben?
- Warum müssen Dateinamen typischerweise als `'\0'`-terminierte Zeichenkette (im "C-Stil") übergeben werden?
- Welche Öffnungsmodi für Dateien kennen Sie? Nennen Sie mindestens drei.

Nächste Einheit:
Fortgeschrittenes Suchen