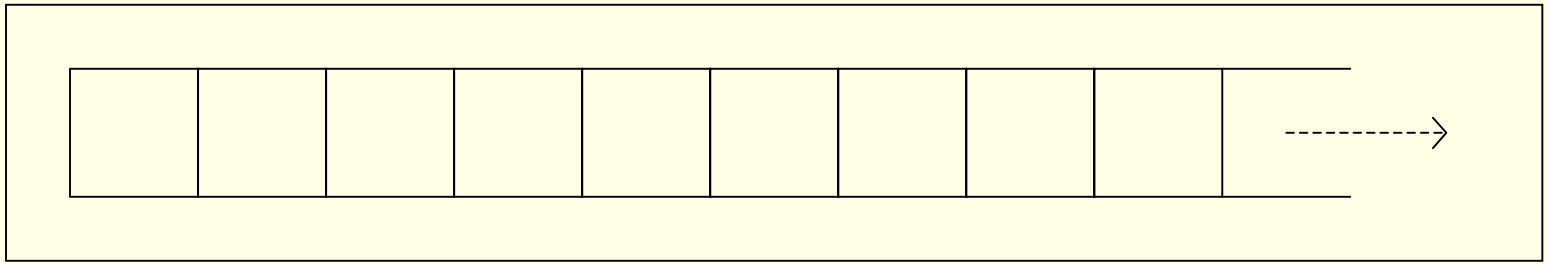
Die C++ Standardbibliothek

Der Container Vector



Johannes Kirsch
17.05.2002

Übersicht

- Motivation am Beispiel
- Container – gemainsame Eigenschaften
- Die Container-Klasse Vektor
  - Datentypen
  - Konstruktoren
  - Iteratoren
  - Elementzugriff
  - Operationen
  - Größe und Kapazität
  - Adressierung von Elementen
- Beispiel
- Quellen

Motivation am Beispiel

- Erzeugen von Mengen für verschiedenste Arten von Objekten sowie deren Manipulation
- Container – Klasse mit der Hauptaufgabe Objekte zu verwalten
- Beispiel: Verwaltung von Namen und Telefonnummer
  - typischer C-Ansatz: Feld mit Paaren (struct)
  - Feste Größe – zu groß? Zu klein? – Code zur Speicherverwaltung
- Besser: Template vector aus der Standardbibliothek

Container – gemeinsame Eigenschaften

- Von Elementen werden Kopien angelegt – müssen Copy-Konstruktor besitzen
- Alternative – Verweise auf die Elemente übergeben
- Elemente besitzen Reihenfolge und können sequentiell durchlaufen werden
- Operationen sind nicht sicher, es werden keine Exceptions geworfen
- Falls Exception auftritt ist Zustand des Containers undefiniert
- Gemeinsame Operationen:
  - empty()
  - size()
  - Vergleichsoperationen
  - swap()
  - begin(), end()
  - insert()
  - …

Die Container-Klasse Vektor

- Ein Vektor verwaltet Elemente beliebigen Typs als dynamisches Array
- Für die Verwendung benötigt man die Headerdatei `<vector>`

```
#include <vector>
```

- Template-Klasse im Namensbereich std

```
namespace std{
   template <class T, class A = allocator<T> > class vector;
}
```

- Erste Templateparameter – Elemente des Vektors
- Zweite Templateparameter – Legt das Speichermodell fest

Datentypen

- Standardisierte Typnamen:

```
public:
   typedef T value_type;
   typedef A allocator_type;
   typedef typename A::size_type size_type;
   //Datentyp zur Indizierung des Containers
   typedef implementierungsspezifisch iterator;// T*
   typedef implementierungsspezifisch const_iterator;//const T*
   typedef typename A::reference reference;
   //Referenz auf Element - X&
   ...
```

- In jedem Container vorhanden
- Datentypen ausreichend um Code für Container zu schreiben

Beispielfuktion mit Datentypen

```
template <class C> typename C::value_type summe (const C& c)
{
   typename C::value_type s = 0;
   typename C::const_iterator p = c.begin();
   while(p != c.end())
   {
        s += *p;
        ++p;
   }
   return s;
}
```

Konstruktoren

- Vector ermöglicht schnellen Zugriff auf Elemente, Ändern der Größe ist teuer – beim Erzeugen Startgröße mitgeben
- Sequenzen von Werten als Parameter
- Copy-Konstruktor und Zuweisungsoperator kopieren die Elemente – Vektoren als Referenz übergeben
- assign() Funktionen – Gegenstücke zu Konstruktoren, die für mehrere Argumente aufgerufen werden können

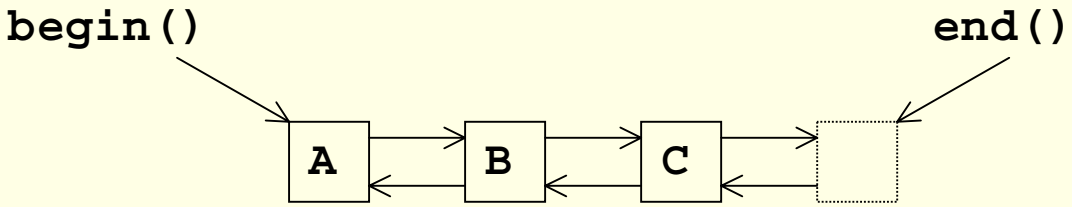
```
explizit vector(size_type n, const T& wert=T(), const A&=A());
template <class In> vector(In anf, In end, const A&=A());
vector& operator=(const vector& x);
template <class In> void assign(In anf, In end);
...
```

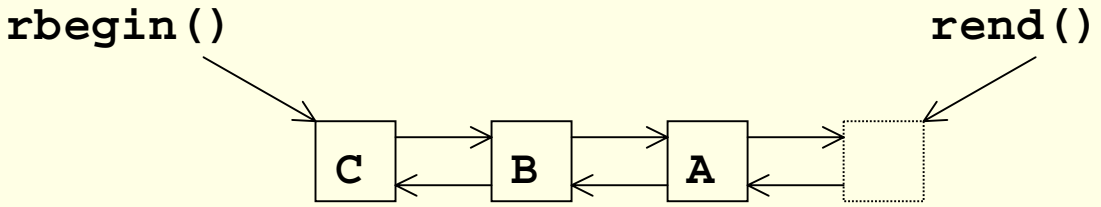
Iteratoren/Allokatoren

- Ein Iterator ist eine Abstraktion eines Zeigers auf ein Element einer Sequenz
- Alles was sich wie ein Iterator verhält ist auch einer
- Fundamentale Konzepte:
  - Dereferenzierung (Operatoren * und ->)
  - Auf das nächste Element zeigen (Operator ++)
  - Auf Gleichheit testen (Operator ==)
- Ein Iterator ist gültig, wenn er auf ein Element zeigt

Iteratoren

- Verwendung um durch Container zu navigieren, unabhängig vom Datentyp
- **`begin(), end()`** liefert **`iterator`** oder **`const_iterator`**
- Reverse-Itaretoren:
- **`rbegin(), rend()`**
- **`reverse_iterator.base()`** liefert korrespondierenden iterator





Beispiel – Reverse-Iterator

```
template <class C>
typename C::reverse_iterator
find_last(const C& c, typename C::value_type wert)
{
   typename C::reverse_iterator p = c.rbegin();
   while(p != c.rend())
   {
       if(*p == wert) return p;
       ++p; // nicht --p !
   }
   return p;
}
```

Elementzugriff

- Wichtig bei vector – leichter und effizienter Zugriff auf einzelne, beliebige Elemente

```
reference operaror[](size_type n); //ungeprüfter Zugriff
const_referance operator[](size_type n) const;
reference at(size_type n); //geprüfter Zugriff
reference front();
reference back();
```

- Falls Bereich schon geprüft – ungeprüften Operator verwenden, ansonsten geprüften Opertator. Bsp:

```
for(int i=0; i<v.size(); i++) benutze v[i];
try{
v.at(x) = v.at(y); //prüfe Bereich beim Zugriff
}catch(out_of_range){ ... }
```

Operationen (Stack)

- Übliche Verwendung von Feldern und Vektoren als Stack – daher Stackoperationen sinnvoll
- außerdem: inkrementelle Erzeugung eines Vektor
- Operationen betrachten Vektor als Stack – am Ende einfügen und entfernen

```
void push_back(const T& x);
void pop_back();
```

- Beachte: `pop_back()` liefert keinen Wert, ebtfernt nur das letzte Element
- Stack-Operationen haben sehr gutes Teiverhalten, Listen-Operationen schlechter

Operationen (Liste)

- Elemente in der Mitte eines Vektor einfügen und entfernen
- Sind allgemeiner als Stackoperationen, aber auch teurer
- Elemente dahinter müssen verschoben werden – Zuweisungsoperator für jedes verschobene Element
- Wenn diese Funktionen im Vordergrund stehen, Liste verwenden
- Vorsicht bei Iteratoren!

```
iterator insert(iterator pos, const T& x);
iterator insert(iterator pos, In anf, In end);
iterator erase(iterator anf, iterator end);
void clear();
...
```

Größe und Kapazität

- Es ist gelegentlich sinnvoll in die Speicherverwaltung einzugreifen
- Mögliche Operationen:

```
size_type size() const; // #Elemente
size_type max_size() const; // max. #Elemente
void resize(size_type sz, T wert=T());
size_type capacity() const;
void reserve(size_type n);
```

- capacity() liefert Anzahl Elemente die ein Vektor ohne neuen Speicher anzufordern aufnehmen kann
- Bei Reallokierung werden alle Verweise und Referenzen ungültig, da alle Elemente vom alten in den neuen Speicher kopiert werden (Copy-Konstruktor + Destruktor)

Vier Iterator Probleme

```
int main(){
   vector<Date> e;
   copy(istream_iterator<Date>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ));
   vector<Date>::iterator first =
        find(e.begin(), e.end(), "17/05/02" );
   vector<Date>::iterator last =
        find(e.begin(), e.end(), "12/31/02" );
   *last = "12/30/02";
   copy( first, last, ostream_iterator<Date>( cout, "\n") );
   e.insert( --e.end(), TodaysDate() );
   copy( first, last, ostream_iterator<Date>( cout, "\n") );
}
```

Vier Iterator Probleme

```
int main(){
   vector<Date> e;
   copy(istream_iterator<Date>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ));
   vector<Date>::iterator first =
        find(e.begin(), e.end(), "17/05/02" );
   vector<Date>::iterator last =
        find(e.begin(), e.end(), "12/31/02" );
   *last = "12/30/02";
   copy( first, last, ostream_iterator<Date>( cout, "\n") );
   e.insert( --e.end(), TodaysDate() );
   copy( first, last, ostream_iterator<Date>( cout, "\n") );
}
```

Vier Iterator Probleme

```
int main(){
   vector<Date> e;
   copy(istream_iterator<Date>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ));
   vector<Date>::iterator first =
        find(e.begin(), e.end(), "17/05/02" );
   vector<Date>::iterator last =
        find(e.begin(), e.end(), "12/31/02" );
   *last = "12/30/02";
   copy( first, last, ostream_iterator<Date>( cout, "\n") );
   e.insert( --e.end(), TodaysDate() );
   copy( first, last, ostream_iterator<Date>( cout, "\n") );
}
```

Vier Iterator Probleme

```
int main(){
   vector<Date> e;
   copy(istream_iterator<Date>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ));
   vector<Date>::iterator first =
        find(e.begin(), e.end(), "17/05/02" );
   vector<Date>::iterator last =
        find(e.begin(), e.end(), "12/31/02" );
   *last = "12/30/02";
   copy( first, last, ostream_iterator<Date>( cout, "\n") );
   e.insert( --e.end(), TodaysDate() );
   copy( first, last, ostream_iterator<Date>( cout, "\n") );
}
```

Vier Iterator Probleme

```
int main(){
   vector<Date> e;
   copy(istream_iterator<Date>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ));
   vector<Date>::iterator first =
        find(e.begin(), e.end(), "17/05/02" );
   vector<Date>::iterator last =
        find(e.begin(), e.end(), "12/31/02" );
   *last = "12/30/02";
   copy( first, last, ostream_iterator<Date>( cout, "\n") );
   e.insert( --e.end(), TodaysDate() );
   copy( first, last, ostream_iterator<Date>( cout, "\n") );
}
```

Vier Iterator Probleme

```
int main(){
   vector<Date> e;
   copy(istream_iterator<Date>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ));
   vector<Date>::iterator first =
        find(e.begin(), e.end(), "17/05/02" );
   vector<Date>::iterator last =
        find(e.begin(), e.end(), "12/31/02" );
   *last = "12/30/02";
   copy( first, last, ostream_iterator<Date>( cout, "\n") );
   e.insert( --e.end(), TodaysDate() );
   copy( first, last, ostream_iterator<Date>( cout, "\n") );
}
```

Quellen/Tools

- Bjarne Stroustrup – „Die C++ Programmiersprache“
  3. Auflage – 1998
- Nicolai Josuttis – „Die C++ Standardbibliothek“
- Gezeigte Beispiele: Microsoft Visual C++ 6.0