C++ Standardbibliothek

- Die C++ Standardbibliothek (standard library) stellt ein erweiterbares Rahmenwerk mit folgenden Komponenten zur Verfügung:
  - Diagnose
  - Strings
  - Container
  - Algorithmen
  - komplexe Zahlen und numerische Algorithmen
  - Anpassung an nationale Zeichensätze
  - Ein-/Ausgabe und vieles mehr.
- Sie basiert zum Teil auf der „Standard Template Library“ (STL), die von Hewlett-Packard (HP) entwickelt wurde.

Container

- Eine Behälterklasse (container) ist eine Datenstruktur zur Speicherung einer Anzahl von Objekten.
- In C gibt es zwei Arten von eingebauten Containern:
  - C-Felder enthalten gleichartige Objekte:

```
int  a[500];
```

  - Strukturen fassen logisch zusammengehörige Daten zusammen

```
struct Student {
                string                  name;
                Geschlecht              geschlecht;
                unsigned short          semesterzahl;
                Studienfach             studienfach;
                unsigned long           matrikelnummer;
                unsigned short          uebungsnummer;
                string                  name_des_bremsers;
                vector<int>             resultate_der_uebungen;
                float                   note;
        };
```

Container

- In der C++ Standardbibliothek gibt es komplexere, aber komfortablere Behälterklassen:

Header-Name:

<vector>
<list>
<queue>
<stack>
<deque>
<map>
<set>
<bitset>

C++ Standardbibliothek

- Zunächst studieren wir die Klasse <vector>.
- Wir diskutieren diese Klasse, indem wir eine eigene Klasse <Vektor> mit ähnlicher Funktionalität vorstellen (implementieren).
- Anschließend untersuchen wir die Laufzeiten der wichtigsten Operationen unserer Klasse <Vektor>.

Eine eigene Vektor-Klasse

Auf den folgenden Seiten stellen wir die Deklaration und einen Teil der Definition (Implementierung) einer Beispiel-klasse Vector vor:

```
#ifndef vektor_t
#define vektor_t
#include<cstddef>                          // wegen NULL

template <class T>
class  Vektor {
   public:
            /*  Der öffentliche Teil der Klassendefinition
               folgt auf den nächsten Seiten !!!          */
   private:
            T       *start;          //  Zeiger auf Anfang des Vektors
            int     number;          //  Größe=Zahl der Vektorelemente
};
```

Eine eigene Vektor-Klasse


```
// Öffentliche Teil der Klassendefinition von „Vektor".
// 1.  Konstruktoren, Destruktor, Zuweisungsoper.

Vektor(int x = 0)                          //  Allg. Konstruktor
: number(x), start(new T[x]) {
}

Vektor(const Vektor<T>& v)                 //  Kopierkonstruktor
: number(v.number), start(new T[v.number]) {
      for (int i = 0; i < number; i++)  start[i] = v.start[i];
}

~Vektor( ) { delete [ ] start; }           //  Destruktor

Vektor<T>&  operator=(const Vektor<T>&);
```

Eine eigene Vektor-Klasse


```
// Indexoperatoren (inline)

    T& operator[](int index) {
            return start[index];
    }

    const  T& operator[](int index) const {
            return start[index];
    }
```

Eine eigene Vektor-Klasse


```
// Wie groß ist der Vektor? Ist er leer?
bool empty( ) const { return number == 0;}
int  size()       const { return number;}

/* Größe des Vektors ändern: Falls new_size < size, dann wird ein
   kleinerer Vektor allokiert und die entsprechenden Elemente
   des alten Vektors werden kopiert. Falls new_size > size, dann
   wird ein größerer Vektor allokiert, der alte Vektor kopiert und
   die neuen Elemente werden mit Kopien von t gefüllt. */
void resize(int new_size, T t = T( ));

// Zeiger auf Anfang und Position nach dem Ende
T* begin() { return start; }
const T* begin() const { return start;}
T* end() { return start+number; }
const T* end() const { return start+number;}
```

Eine eigene Vektor-Klasse


```
/* Es folgen die Implementierungen der Funktionen, deren
   Implementierungen noch nicht angegeben wurden. */

// Zuweisungsoperator
template<class T>
inline Vektor<T> & Vektor<T>::operator=(const Vektor<T>& v) {
   if (this != &v) {   // Zuweisung identischer Objekte vermeiden
     T  *temp = new T[v.number];
     for (int i = 0; i < v.number; ++i)  temp[i] = v.start[i];
     delete [ ] start;
     number = v.number;
     start = temp;
   }
   return *this;
}
```

Eine eigene Vektor-Klasse


```
/* Größe des Vektors ändern: Falls new_size < size, dann werden
   die entsprechenden Elemente des alten Vektors kopiert. Falls
   new_size > size, dann wird der alte Vektor kopiert und die
   neuen Elemente werden mit Kopien von t gefüllt. */

   template<class T>
   void Vektor<T>::resize(int new_size, T  t = T( )) {
     T *temp = new T[new_size];

     int smaller_number = (new_size < number) ? new_size : number;
     for (int i = 0; i < smaller_number; ++i)  temp[i] = start[i];
     delete [ ] start;
     start = temp;

     if (smaller_number == number)
        for (int i = number; i < new_size; ++i) start[i] = t;
     number = new_size;
   }
```

Wörterbücher

- Jeder Container enthält eine Menge K von n Objekten.
- Jedes Objekt besitzt einen eindeutigen Schlüssel k.
- Mittels dieses Schlüssels kann das Objekt identifiziert werden.
- Als Beispiel für einen Container betrachten wir im Folgenden einen Vektor<T> mit Elementen vom Datentyp T.
- Wir nehmen o.B.d.A. der Einfachheit wegen an, dass der Datentyp T =„int“ ist und dass die gespeicherten ganzen Zahlen gleichzeitig Objekte und Schlüssel sind.

Wörterbücher

Typische Operationen auf Container-Objekten sind:

Search(K,k)	Suche ein Element x in K mit Schlüssel k.
Insert(K,x)	Füge das Element x in K ein.
Delete(K,x)	Entferne das Element x aus K.
Minimum(K)	Suche das (ein) Element von K mit minimalem Schlüssel.
Maximum(K)	Suche das (ein) Element von K mit maximalem Schlüssel.
Successor(K,x)	Suche das Element, dessen Schlüssel in der Ordnung von K auf den Schlüssel von x folgt.
Predessor(K,x)	Suche das Element, dessen Schlüssel in der Ordnung von K dem Schlüssel von x vorangeht.

Eine dynamische Menge, die diese Operationen unterstützt, bezeichnet man als „Wörterbuch“ (dictionary).

Laufzeiten der Operationen für Vektoren

	nicht-sortierter Vektor	sortierter Vektor
Search(K,k)	O(n)	???
Insert(K,x)		
Delete(K,x)		
Minimum(K)		
Maximum(K)		
Successor(K,x)		
Predessor(K,x)		

Eine eigene Vektor-Klasse


```
/*  Rekursive binäre Suche für einen sortierten Vektor. */

template<class T>
int  binarySearch( const Vektor<T>& v, const T& value, int left, int right)
{
       if( left >= right)  return -1;

       int middle = left + (right-left)/2;

        if (middle == left) {
                  if (v[left] == value) return left;
                  else return -1;
        }
        else  {
                  if  ( v[middle] > value)
                            return binarySearch(v, value, left, middle);
                  else       return binarySearch(v, value, middle, right);
        }
}
```

Eine eigene Vektor-Klasse


```
/*  Binäre Suche für einen sortierten Vektor. */

   template<class T>
   int  binarySearch( const Vektor<T>& v, const T& value)
    {
          if (v.size() == 0) return -1;
          int left = 0, right = v.size();
          int middle = left + (right-left)/2;

          for ( ; middle != left; middle = left + (right-left)/2)
               if  ( v[middle] > value)     right = middle;
              else                          left = middle;

          if (v[left] == value) return left;
          else return -1;
    }
```

C_1

C_2*X

C_3

Eine eigene Vektor-Klasse

Laufzeit von „binarySearch“ für v.size() = n:

$$T(n) = c_1 + c_3 + c_2 * x;$$

Wie groß ist x (o.B.d.A. $n = 2^{\log(n)}$ mit $\log(n) \in \mathbb{Z}$) ?

$$\frac{n}{2^x} = 1 \Leftrightarrow x = \log(n)$$

Hieraus folgt:

$$T(n) \leq c*\log(n) \Rightarrow T(n) \in O(\log(n))$$

Eine eigene Vektor-Klasse

Wir betrachten im Folgenden die rekursive Version von „binarySearch“. Die Laufzeit dieser Version kann durch die folgende Rekurrenz abgeschätzt werden:

$$T(n) \leq c + T\left(\left\lceil \frac{n}{2} \right\rceil\right)$$

o.B.d.A. $n = 2^{\log(n)}$ mit $\log(n) \in Z$:

$$T(n) \leq c + T\left(\frac{n}{2}\right) \leq c + c + T\left(\frac{n}{4}\right)$$

$$\leq \underbrace{c + c + \cdots + c}_{\log(n)+1} = c\log(n) + c$$

$$\Rightarrow T(n) \in O(\log(n))$$

Laufzeiten der Operationen für Vektoren

	nicht-sortierter Vektor	sortierter Vektor
Search(K,k)	O(n)	O(log(n))
Insert(K,x)	O(1)	
Delete(K,x)	O(1)	

nicht-sortierter Vektor:

Insert(K,x): Der Schlüssel k von x wird in dem Feldelement i mit $0 \leq i < n$ gespeichert: v[i] = k;

Delete(K,x): Der Schlüssel k von x, der in dem Feldelement i gespeichert ist, wird mit einem anderen Schlüssel k_new überschrieben: v[i] = k_new;

Laufzeiten der Operationen für Vektoren

	nicht-sortierter Vektor	sortierter Vektor
Search(K,k)	O(n)	O(log(n))
Insert(K,x)	O(1)	O(n)
Delete(K,x)	O(1)	O(n)

Worst-Case bei sortiertem Vektor:

Insert(K,x): Ein neuer kleinster Schlüssel k wird eingefügt. Alle Elemente in dem Vektor werden verschoben!

Delete(K,x): Das Element mit dem kleinsten Schlüssel wird gelöscht. Alle verbleibenden Elemente werden verschoben.

Laufzeiten der Operationen für Vektoren

	nicht-sortierter Vektor	sortierter Vektor
Search(K,k)	O(n)	O(log(n))
Insert(K,x)	O(1)	O(n)
Delete(K,x)	O(1)	O(n)
Minimum(K)	O(n)	O(1)
Maximum(K)	O(n)	O(1)
Predecessor(K,x)	O(n)	O(1)
Successor(K,x)	O(n)	O(1)

Eine eigene Vector-Klasse

- Bei gewissen Operationen, z.B. Einfügen und Löschen von Elementen, muss die Feldgröße geändert werden und große Teile des Feldes müssen dann kopiert werden.
- Falls diese Operationen häufig durchgeführt werden, so benötigt man eine effizientere „dynamische“ Vektor-Klasse:
  - Wir reservieren doppelt soviel Speicherplatz wie benötigt:

    capacity = 2*number
  - Solange noch freier Speicherplatz vorhanden ist,

    capacity > number

    können wir neue Elemente effizient einfügen (push_back).

Eine eigene Vector-Klasse

  - Falls durch viele Einfügungen

    capacity == number

    dann verdoppeln wir den zur Verfügung stehenden Speicherplatz

    capacity *=2

    und kopieren die vorhandenen Elemente in das neu allokierte Feld.
  - Falls durch viele Delete-Operationen

    capacity == 4*number

    dann halbieren wir den zur Verfügung stehenden Speicherplatz

    capacity /=2

    und kopieren die vorhandenen Elemente in das neu allokierte Feld.

Eine eigene Vector-Klasse


```
// Version 2 der Klasse Vektor
#ifndef vektor_t
#define vektor_t
#include<cstddef>                        // wegen NULL

template <class T>
class  Vektor {
   public:
            /*  Der öffentliche Teil der Klassendefinition
                folgt auf den nächsten Seiten !!!          */
   private:
            T   *start;        //  Zeiger auf Anfang des Vektors
            int  number;       //  Zahl der gespeicherten Elemente
            int  capacity;     //  Größe des reservierten Speicherplatzes
};
```

Eine eigene Vektor-Klasse


```
// Öffentliche Teil der Klassendefinition von „Vektor".
// 1.  Konstruktoren, Destruktor, Zuweisungsoper.

Vektor(int x = 0)                                  //  Allg. Konstruktor
: number(x), capacity(2*x), start(new T[2*x]) {
}

Vektor(const Vektor<T>& v)                         //  Kopierkonstruktor
: number(v.number), capacity(v.capacity), start(new T[v.capacity]) {
      for (int i = 0; i < number; i++)  start[i] = v.start[i];
}

~Vektor( ) { delete [ ] start; }                   //  Destruktor

Vektor<T>&  operator=(const Vektor<T>&);    //  Zuweisungsoperator
```

Eine eigene Vektor-Klasse


```
// Indexoperatoren (inline)

    T& operator[](int index) {
            return start[index];
    }

    const  T& operator[](int index) const {
            return start[index];
    }
```

Eine eigene Vektor-Klasse


```
// Wie groß ist der Vektor? Ist er leer?
bool empty( ) const { return number == 0;}
int  size()       const { return number;}

/* Größe des Vektors ändern: Falls new_size < size, dann wird ein
   kleinerer Vektor allokiert und die entsprechenden Elemente
   des alten Vektors werden kopiert. Falls new_size > size, dann
   wird ein größerer Vektor allokiert, der alte Vektor kopiert und
   die neuen Elemente werden mit Kopien von t gefüllt. */
void resize(int new_size, T t = T( ));

// Zeiger auf Anfang und Position nach dem Ende
T* begin() { return start; }
const T* begin() const { return start;}
T* end() { return start+number; }
const T* end() const { return start+number;}
```

Eine eigene Vektor-Klasse


```
/* Es folgen die Implementierungen der Funktionen, deren
   Implementierungen noch nicht angegeben wurden. */

// Zuweisungsoperator
template<class T>
inline Vektor<T> & Vektor<T>::operator=(const Vektor<T>& v) {
   if (this != &v) {   // Zuweisung identischer Objekte vermeiden
     T  *temp = new T[v.capacity];
     for (int i = 0; i < v.number; ++i)  temp[i] = v.start[i];
     delete [ ] start;
     number = v.number;
     capacity = v.capacity;
     start = temp;
   }
   return *this;
}
```

Eine eigene Vektor-Klasse


```
/* Größe des Vektors ändern: Falls new_size < size, dann werden
   die entsprechenden Elemente des alten Vektors kopiert. Falls
   new_size > size, dann wird der alte Vektor kopiert und die
   neuen Elemente werden mit Kopien von t gefüllt. */

   template<class T>
   void Vektor<T>::resize(int new_size, T  t = T( )) {
     capacity = 2*new_size;
     T *temp = new T[capacity];

     int smaller_number = (new_size < number) ? new_size : number;
     for (int i = 0; i < smaller_number; ++i)  temp[i] = start[i];
     delete [ ] start;
     start = temp;

     if (smaller_number == number)
        for (int i = number; i < new_size; ++i)  start[i] = t;
     number = new_size;
   }
```

Eine eigene Vektor-Klasse

Die Klasse <vector> der STL stellt unter anderem folgende Operationen zum Einfügen eines neuen letzten Elements und zum Löschen des letzten Elements zur Verfügung:

```
void  push_back(const T& t);

void  pop_back();
```

Wir diskutieren auf den beiden folgenden Seiten ähnliche Operationen für unsere Vektor-Klasse.

Eine eigene Vektor-Klasse


```
/* Unsere Implementierung der Funktion „push_back“ */

template<class T>
void Vektor<T>::push_back(const T& t) {
   if (number < capacity) start[number++] = t;
   else {
            capacity = (capacity > 0) ? capacity * 2: 2;
            T*  temp = new T[capacity];
            for (int i = 0; i < number; ++i)  temp[i] = start[i];
            temp[number++] = t;
            delete [ ] start;
            start = temp;
        }
}
```

Eine eigene Vektor-Klasse


```
/* Unsere Implementierung der Funktion „pop_back“ */

template<class T>
void Vektor<T>::pop_back( ) {
   if (number > (capacity/4)) --number;
   else if (number != 0) {
            --number;
            capacity /= 2;
            T*  temp = new T[capacity];
            for (int i = 0; i < number; ++i)  temp[i] = start[i];
            delete [ ] start;
            start = temp;
         }
}
```

Eine eigene Vektor-Klasse

– Push_back- und Pop_back-Operationen sind meist sehr effizient (O(1)), aber manchmal auch sehr teuer (O(n)).

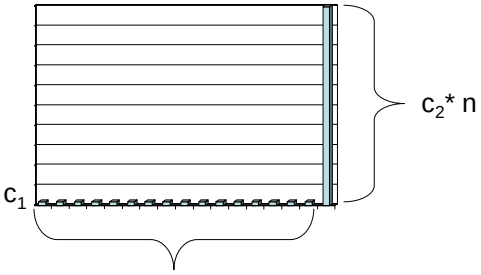
– Wie sieht die durchschnittliche Laufzeit aus?

- Man mittelt die Kosten von n aufeinander folgenden Operationen.
- Hierbei bestimmt man die Kosten T(n) der ungünstigsten Folge von n Operationen (worst case).

Die amortisierten Kosten T(n)/n beschreiben die durchschnittliche Performanz/Kosten von Daten-strukturen/Operationen im Worst-Case.

Eine eigene Vektor-Klasse

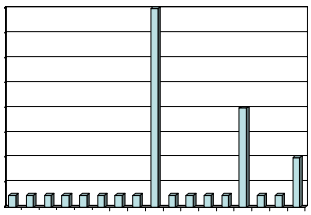
– Wir betrachten zunächst eine Folge von n „push_back“ Operationen, wobei n = „number“ die Zahl der Elemente vor der ersten „push_back“ Operation ist:



$$\frac{T(n)}{n} = \frac{(c_1 + c_2)n - c_1}{n} \leq c_1 + c_2 \Rightarrow \frac{T(n)}{n} \in O(1)$$

Eine eigene Vektor-Klasse

– Wir betrachten nun eine Folge von n „pop_back“ Operationen, wobei n = number die Zahl der Elemente vor der ersten „pop_back“ Operation ist:



$$T(n) \leq \sum_{i=1}^{\log(n)} c\frac{n}{2^i}$$

$$\frac{T(n)}{n} \leq c\sum_{i=1}^{\log(n)} \frac{1}{2^i} \leq c\left(\frac{(1/2)^{\log(n)+1} - 1}{(1/2) - 1}\right) - c \leq 2c - c \leq c$$

$$\Rightarrow \frac{T(n)}{n} \in O(1)$$

Eine eigene Vektor-Klasse

– Auf den beiden vorhergehenden Folien haben wir gezeigt, dass, ausgehend von einem Vektor mit n Elementen und Kapazität 2n,

  • eine Folge von n „push_back“ Operationen amortisierte Kosten O(1) verursacht und dass

  • eine Folge von n „pop_back“ Operationen amortisierte Kosten O(1) verursacht.

– Man kann auch zeigen, dass eine beliebige gemischte Folge von n „push_back“ und „pop_back“ Operationen nur amortisierte Kosten von O(1) besitzt.

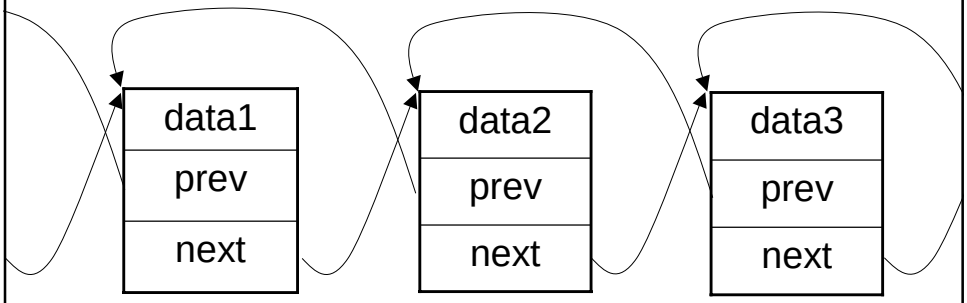
Eine Beispielklasse für Listen


```
template<class T>
class List {
   public:
            /*  Einige öffentliche Methoden werden auf den
                      folgenden Folien diskutiert.                */
   private:
            class ListElement  {       // „nested classes“
                     friend class List<T>;
                     T data;
                     ListElement *next, *prev;
                     ListElement(const T& dat)
                     :  data(dat), next(NULL), prev(NULL) { }
            };       // Ende der Deklaration von ListElement

   ListElement  *start, *end;          // Anfang und Ende der Liste
   int  number;                        // Anzahl der Listenelemente
};
```

Eine Beispielklasse für Listen

- Die Struktur eines Listenelements wird durch die geschachtelte Klasse „ListElement“ beschrieben.



Eine Beispielklasse für Listen

- Damit die Funktionen der Klasse „List“ auf die Daten der Klasse „ListElement“ zugreifen können, wird „List“ innerhalb von „ListElement“ als „friend“-Klasse deklariert.

- Deklariert man in einer Klasse1 eine andere Klasse2 als „Freund“-Klasse, so haben die Funktionen der Klasse2 Zugriff auf alle Daten und Funktionen von Klasse1, auch auf die privaten.

Eine Beispielklasse für Listen


```
// Defaultkonstruktor
List()
: end(NULL), start(NULL), number(0) {  }

// Kopierkonstruktor
List(const List&);

// Destruktor
~List( );

// Zuweisungsoperator
List& operator=(const List&);
```

Eine Beispielklasse für Listen


```
//  Ist die Liste leer?  Größe der Liste ausgeben.
bool empty( ) const { return number == 0;}
int  size()       const { return number;}

// am Anfang bzw. am Ende einfügen
void push_front( const T&);
void push_back( const T&);

// am Anfang und Ende löschen
void pop_front();
void pop_back();

// am Anfang bzw. Ende lesen
T& front();
const T& front() const;
T& back();
const T& back() const;

// Anwenden von Funktionen f ( ) auf alle Elemente
void apply( void (*f)(T&));
```

Eine Beispielklasse für Listen


```
// Datei list.cpp :
// Implementierung von einigen Funktionen der Klasse
template<class T>                          // Kopierkonstruktor
List<T>::List(const List<T>& L)
 : end(NULL), start(NULL), number(0) {
        ListElement *temp = L.end;
        while (temp) {
                push_front(temp->data);
                temp = temp->prev;
        }
}

template<class T> List<T>::~List()   { /* Übungsaufgabe*/ }
```

Eine Beispielklasse für Listen


```
        /* push_front führt ein neues Element
                am Anfang der Liste ein.        */

template<class T>
List<T>::push_front(const T& dat) {
    ListElement *temp = new ListElement(dat);
    // der Kopierkonstruktor setzt temp->prev = NULL
    temp->next = start;
    if (!start)  end = temp;
    else start->prev = temp;
    start = temp;
    number++;
}
```

Eine Beispielklasse für Listen


```
/* Übung: Wie löscht man alle Listenelemente mit einem
                    bestimmten Datenelement?                */

template<class T>
void List<T>::remove(const T& dat) {
  ListElement  *it = start, *old;
   while (it != NULL; )
            if (it->data == dat) {
                    it->prev->next = it->next;
                    it->next->prev = it->prev;
                    old = it;
                    it = it->next;
                    delete *old;
            }
}
```

Eine Beispielklasse für Listen


```
//  Funktion f auf alle Elemente der Liste anwenden
template<class T>
List<T>::apply(void(*f) (T&))  {
  ListElement *temp = start;
  while (temp) {
            // wende f auf das aktuelle Listenelement an
            f(temp->data);
            temp = temp->next;
  }
}
```

Laufzeiten für Wörterbuch-Operationen

	sortiertes Feld	nicht sortiertes Feld	sortierte doppelt-verkettete Liste	nicht sortierte doppelt verkettete Liste
Search	O(log n)	O(n)	O(n)	O(n)
Insert	O(n)	O(1)	O(n)	O(1)
Delete	O(n)	O(1)	O(1)	O(1)
Successor	O(1)	O(n)	O(1)	O(n)
Predecessor	O(1)	O(n)	O(1)	O(n)
Mimimum	O(1)	O(n)	O(1)	O(n)
Maximum	O(1)	O(n)	O(1)	O(n)

Eine Beispielklasse für eine Warteschlange

- Queues zeichnen sich durch folgende Eigenschaften aus:
  - Elemente können nur am Anfang eingefügt werden und
  - nur am Ende entnommen werden (FIFO: first in – first out)
- Zur Implementierung verwenden wir die Listenklasse.
- Das Listenobjekt L wird privat angelegt und die Element-funktionen der Klasse Queue rufen die öffentlichen Elementfunktionen des Objekts L auf.
- Die Klasse Queue delegiert somit Aufgaben an die Klasse List (Prinzip: Delegation).

Eine Beispielklasse für eine Warteschlange


```
#include “list.t“

template<class T>
class Queue {
  public:
    bool       empty( )   const      { return L.empty(); }
    int        size( )    const      { return L.size(); }
    void       push( const T& x)     { L.push_back(x); }
    void       pop()                 { L.pop_front(); }
    T&         front()               { return L.front(); }
    const T&   front()    const      { return L.front(); }
    T&         back()                { return L.back(); }
    const T&   back()     const      { return L.back(); }
    void       apply(void(*f)(T&) )  { L.apply(f); }

  private:
    List<T>  L;
};
```

Eine Beispielklasse für eine Warteschlange

- Man beachte:
  - Jede Queue enthält ein Objekt L der Klasse „List“.
  - Wir müssen keinen Kopierkonstruktor implementieren, da der vom Compiler zur Verfügung gestellte Kopier-konstruktor diese Aufgabe schon perfekt erledigt.
  - Der Default-Kopierkonstruktur ruft nämlich den Kopierkonstruktor der Klasse List für das einzige Datenelement L der Klasse Queue auf.
  - Analoge Aussagen gelten für den Zuweisungsoperator und den Destruktor der Klasse Queue.

Laufzeiten für Wörterbuch-Operationen

	sortiertes Feld	nicht sortiertes Feld	sortierte doppelt-verkettete Liste	nicht sortierte doppelt verkettete Liste	(nicht sortierter) Stack/ Queue
Search	O(log n)	O(n)	O(n)	O(n)	O(n)
Insert	O(n)	O(1)	O(n)	O(1)	O(1)
Delete	O(n)	O(1)	O(1)	O(1)	O(1)
Successor	O(1)	O(n)	O(1)	O(n)	O(n)
Predecessor	O(1)	O(n)	O(1)	O(n)	O(n)
Mimimum	O(1)	O(n)	O(1)	O(n)	O(n)
Maximum	O(1)	O(n)	O(1)	O(n)	O(n)

Iteratoren

- Ein Iterator ist kein Typ und kein Objekt, sondern ein Konzept.
- Iteratoren dienen dazu, auf Elemente eines Containers zuzugreifen.
- Wir betrachten zunächst einen einfachen Iterator „MyIterator“ vom Typ „int*“ (Zeiger auf int).
- Dann studieren wir eine Klasse „Iterator“ für Listen.

Iteratoren

- Mittels „typedef“ können neue Namen für komplexe Deklarationen eingeführt werden:

```
typedef  float  real;

int main( ) {
        real Zahl = 1.756;      // Zahl ist vom Typ float
}
```

- Wenn wir dasselbe Programm mit double Zahlen rechnen lassen wollen, brauchen wir nur die typedef-Zeile zu ändern:

```
typedef double real;
```

Zeiger als Iterator


```
// include's und namespace lassen wir aus Platzgründen weg
typedef  int* MyIterator;            // neuer Name MyIterator
typedef  void (*Function) (int&); // Function = Zeiger auf eine Funktion

void print( int & x)  {  cout << x; }

void apply( MyIterator start, MyIterator end, Function f ) {
    while (start != end)  f(*start++);
}

int main( ) {
    const int number = 10;
    int array[number];
    for (size_t  i=0; i < number; i++)  array[i] = i * i;

    MyIterator  start = array, end = array + number;
    apply(start, end, print);
}
```

Zeiger als Iterator


```
void apply( MyIterator start, MyIterator end, Function f ) {
      while (start != end)  f(*start++);
}
```

- Oft „durchwandert“ man eine Datenstruktur oder einen Container, um auf jedes Element eine bestimmte Operation anzuwenden.
- Die obige Funktion „apply“ ähnelt der Funktion „for_each()“ aus der C++ Bibliothek:

```
template<class Iterator, class Function>
Function for_each(Iterator start, Iterator end, Function f ) {
   while (start != end) f(*start++);
   return f;
}
```

Zeiger als Iterator


```
template<class Iterator, class Function>Function
for_each(Iterator start, Iterator end, Function f ) {
   while (start != end) f(*start++);
   return f;
}
```

- Die obige Funktion zeigt, welche Operationen (Operatoren) für einen Iterator benötigt werden:
  - Dereferenzieroperator *
  - Referenzieroperator &
  - Vergleichsoperatoren !=, ==
  - Inkrementoperatoren ++, -- und einige andere

Breymann_Folien

Zeiger als Iterator

- Beim trivialen Iterator „MyIterator“ sind diese Operatoren natürlich vorhanden und implementiert.
- Betrachten wir jedoch komplexe Datenstrukturen wie Listen oder (später) Bäume, so muss der Iterator einiges über den internen Aufbau der Datenstruktur wissen.
- Anwendungsbeispiel : Iterator für Listen
- Im Beispielprogramm ist „ListIter“ der Iterator, der mit einer Liste L verknüpft wird.

Iterator für Listen


```
// include's und namespace lassen wir aus Platzgründen weg

void print( int& x) { cout << x; }

int main( ) {
   List<int> L;
   for (int i=0; i < 10; i++)  L.push_front(i*i);

   List<int>::Iterator  ListIter;
   for ( ListIter = L.begin(); ListIter != L.end(); ++ListIter) {
         if (*ListIter == 36) {     // Listenelement mit Wert 36 löschen
               L.erase(ListIter);
               cout << *ListIter << “an aktueller Position\n“;
               ListIter = L.end();
         }
   }
   for_each( L.begin(), L.end(), print);          //  #include<algorithm>
}
```

Eine Beispielklasse für Listen


```
template<class T>
class List {
   public:
            /*  öffentliche Methoden: siehe Folien  39-45 */
   private:
            /*  Hier fügen wir die Klasse Iterator ein:
                    Siehe folgende Folien  ! */
            class ListElement  {       // „nested classes"
                    friend class List<T>;
                    T data;
                    ListElement *next, *prev;
                    ListElement(const T& dat)
                    :  data(dat), next(NULL), prev(NULL) { }
            };          // Ende der Deklaration von ListElement

   ListElement  *start, *end;          // Anfang und Ende der Liste
   int  number;                        // Anzahl der Listenelemente
};
```

Iterator für Listen


```
// die Klasse Iterator in List :

class Iterator {
     public:
            friend class List<T>;
            /* die öffentlichen Elementfunktionen
                folgen auf den nächsten Folien.   */
     private:
            //  das einzige Datenelement
            ListElement* current;

};  //  Ende der Klasse Iterator
```

Iterator für Listen


```
//  zunächst die Konstruktoren
Iterator(ListElement*  Init = NULL}
:  current(Init) {
}

// setze auf Anfang der Liste L
Iterator(const List& L) {
        current = L.begin();
}

//  Präfix-Inkrementoperator
Iterator& operator++( ) {
        if (current) current = current->next;
        return *this;
}
```

Iterator für Listen


```
// Postfix-Inkrementoperator
Iterator& operator++(int) {
        Iterator temp = *this;
        ++*this;
        return temp;
   }

//  Vergleichsoperatoren
bool operator==( const Iterator& x) const {
        return current == x.current;
}

bool operator!=(const Iterator& x )  const {
        return current != x.current;
}
```

Iterator für Listen


```
// begin()
Iterator begin() const {
        return Iterator(start);
}

// end()
Iterator end() const {
        return Iterator();
 }
```

Iterator für Listen


```
void erase(Iterator& pos) {
        if (pos.current == start) {
                pop_front();
                pos.current = start;   // neuer Anfang
        }
        else if (pos.current == end) {
                pop_back();
                pos.current = end;   //  neues Ende
        }
        else  {  // zwischen zwei Elementen
                pos.current->next->prev = pos.current->prev;
                pos.current->prev->next = pos.current->next;
                ListElement *temp = pos.current;
                pos.current = pos.current->next;
                delete temp;
                --number;
        }  // Ende else
} //  Ende  erase()
```

Container

- Sei X der Datentyp eines Beispiel-Containers, z.B.,
  X = vector<T> size_type = int, long, size_t, ... usw.
- Die folgenden Methoden werden von jedem Container zur Verfügung gestellt:

Rückgabetyp Methode	Bedeutung
X( )	Standardkonstruktor für leeren C.
X(const X&)	Kopierkonstruktor
~X( )	Destruktor
iterator begin()	Anfang des Containers
const_iterator begin( )	Anfang des Containers
iterator end()	Position nach Ende des C.
const_iterator end( )	Position nach Ende des C.
size_type size( )	Aktuelle Größe des C.
size_type max_size( )	Maximal mögliche Größe des C.
bool empty( )	size == 0 ?
void swap(X&)	Vertauschen mit Argumentcont.

Container

Rückgabetyp Methode	Bedeutung
X& operator=(const X&)	Zuweisungsoperator =
bool operator==(const X&)	Vergleichsoperator ==
bool operator!=(const X&)	Vergleichsoperator !=
bool operator<(const X&)	Vergleichsoperator <
bool operator>(const X&)	Vergleichsoperator >
bool operator<=(const X&)	Vergleichsoperator <=
bool operator>=(const X&)	Vergleichsoperator >=

Container: Deque

- Der Name Deque ist eine Abkürzung für „double ended queue“, also eine Warteschlange, die das Hinzufügen und Entnehmen sowohl am Anfang als auch am Ende erlaubt.
- Die Deklaration der Klasse ist

```
template<class T> class deque;
```

- Die interessantesten „zusätzlichen“ Funktionen sind:

Rückgabetyp Methode	Bedeutung
reverse_iterator rbegin()	Reverser Iterator, der beim letzten Element beginnt.
const_reverse_iterator rbegin()	Fiktive Position vor dem ersten Element
reverse_iterator rend()	Referenz auf erstes Element
const_reverse_iterator rend()	Referenz auf erstes Element
T& front()	Referenz auf das letzte Element
const T& front( ) const	Referenz ...
T& back( )	
const T& back( ) const	

Container: Deque

Rückgabetyp Methode	Beschreibung
T& operator[ ](size_type n)	Referenz auf das n-te Element
T& at(size_type n)	Referenz auf das n-te Element
void push_front(const T& t)	Fügt t am Anfang ein
void push_back(const T& t)	Fügt t am Ende ein
void pop_front()	Löscht das erste Element
void pop_back()	Löscht das letzte Element
iterator insert(iterator p,const T& t)	Fügt eine Kopie von t vor p ein
iterator erase(iterator q)	Löscht das Element, auf das q zeigt, und zeigt dann auf das Element nach q
iterator erase(iterator s, iterator e)	Löscht den Bereich [s,e)
void clear( )	Löscht alle Elemente
void resize(size_type n, T t = T())	Dequegröße ändern

Container: Liste

Rückgabetyp Methode	Beschreibung
list(size_type n, const T& t)	Erzeugt Liste mit n Kopien von t
void assign(size_type n, const T& t = T())	Liste löschen und anschließend n Kopien von t einfügen
void remove(const T& t)	Alle Elemente entfernen, die gleich t sind.
void reverse( )	Reihenfolge umkehren
void sort( )	Sortiert die Liste mit dem Vergleichsoperator < von T
template<class Compare> void sort(Compare cmp)	Sortiert mit dem Sortierkriterium des Compare-Objekts cmp S.342
void unique()	Entfernt gleiche aufeinander-Folgende Objekte bis auf das erste
void merge(list& L)	Mischt sortierte Listen <
void splice(iterator p, list& x)	Fügt x vor p ein

Funktionsobjekte

- Funktoren sind Objekte, die sich wie Funktionen verhalten, aber alle Eigenschaften von Objekten haben.
- Diese Technik wird in den Klassen der Algorithmen und Klassen der C++-Standardbibliothek häufig eingesetzt.
- Bei Funktoren wird der Funktionsoperator () mit der Operatorfunktion operator()() überladen.
- Das Objekt kann dann wie eine Funktion aufgerufen werden.
- Als Beispiel definieren eine Klasse „less_Point2D“ für Vergleiche von Punkten.

Funktionsobjekte


```
// Klassendeklaration von less_Point2D
class less_Point2D {
   public:
     bool operator( ) (const Point2D&  a, const Point2D& b) {
                if (a.X() < b.X())  return true;
                else if (a.X() > b.X()) return false;
                else if (a.Y() < b.Y()) return true;
                      else return false;
      }
};

//  mögliches Hauptprogramm mit Anwendung von less ohne Header
int main() {
   list<Point2D> L;
   //  L wird irgend wie mit Punkten gefüllt.
   less_Point2D  cmp;
   L.sort(cmp);
}
```

Container: Map

- Der Header <map> definiert die Klasse map<Key, T>, die Paare von Schlüsseln und zugehörigen Daten speichert.
- Hierbei ist der Schlüssel eindeutig (es gibt keine zwei Datensätze mit dem gleichen Schlüssel).
- Die Deklaration der Klasse ist

```
template<class Key,          // Schlüssel
                class T,          // Daten
                class Compare = less<Key> >  // Standardvergleich
class map;
```

- Eine ausführliche Beschreibung dieses Containers finden Sie im Breymann-Buch ab Seite 494.
- Mit der map-Klasse lässt sich leicht ein Wörterbuch realisieren:

```
map<string, string> Woerterbuch;
```

Weitere Container

- Queues <queue> und Stacks <stacks> wurden in der Vorlesung bereits ausführlich diskutiert.
- Diesen beiden Klassen stellen natürlich nur einen Teil der Funktionalität von Deque zur Verfügung.
- Eine Priority-Queue ist eine prioritätsgesteuerte Warteschlange. Priority-Queues werden wir später in der Vorlesung behandeln.
- Die Klasse set<Key, T> entspricht der Klasse map, nur dass Schlüssel und Daten zusammenfallen, d.h., es werden nur Schlüssel gespeichert.
- Die Klasse <vector> haben wir bereits ausführlich diskutiert und verwendet.

Algorithmen

- Alle im Header <algorithm> vorhandenen Algorithmen sind unabhängig von der speziellen Implementierung der Container.
- Sie kennen nur Iteratoren, über die sie auf die Datenstrukturen in den Containern zugegriffen werden kann.
- Die Iteratoren müssen nur wenigen Kriterien genügen.
- Aus Zeitgründen verzichten wir hier auf eine Diskussion der vorhandenen Algorithmen (siehe Breymann-Skript Seite 430).

ADTs und ihre Implementierungen

- Abstrakte Datentypen (ADT) kapseln Daten und Funktionen.
- Eine Klasse ist ein ADT, der in einer Programmiersprache formuliert ist.
- Eine ADT wird ausschließlich über die öffentliche Schnittstelle spezifiziert.
- Die STL erlaubt für manche ADTs verschiedene Implementierungen.

ADTs und ihre Implementierungen


```
template< T, class Containertyp  =  deque<T>>
class stack {
    public:
            bool empty() const {
                    return c.empty();
            }
            // und weitere öffentliche Methoden
    private:
            Containertyp  c;
};
```

- Mögliche Implementierungen von „stack“:

```
stack<int> intStack1;
stack<int,  list<int>>  intStack2;
stack<int,  vector<int>>  intStack3;
```

Container: Bitset

- Der Header <bitset> definiert eine Template-Klasse und zugehörige Funktionen zur Darstellung und Bearbeitung von Bitfolgen fester Größe.
- Die Deklaration der Klasse ist

```
template<size_t N> class bitset;
```

- Eine ausführliche Beschreibung dieses Containers finden Sie im Breymann-Buch auf den Seiten 485-488.