

---

# Container

Ein Container ist eine Behälterklasse, d.h. eine Datenstruktur, die mehrere Objekte aufnehmen kann. Wir werden hier selbst definierte Container behandeln und einige Probleme, die dabei auftreten diskutieren.

## Inhaltsverzeichnis

1. Einleitung.....	2
2. Container für doppelt verkettete Listen.....	6
3. Container für Warteschlangen.....	17
4. Iteratoren.....	23
5. typename.....	40

## 1. Einleitung

Der Entwurf eines Containers ist stets **problemabhängig**. Folgende Aspekte beeinflussen dabei die Entwurfsentscheidung:

- Soll ein Container **Objekte** oder **Zeiger** auf Objekte enthalten?
- Sollen **Zugriffe** auf Elemente über **Werte** oder **Referenzen** erfolgen?
- Soll die Behälterklasse **statisch** in der Anzahl der möglichen Elemente sein oder soll **dynamisches** Wachstum möglich sein?
- Wie kann auf jedes Element im Container zugegriffen werden, ohne dass die **interne Struktur** (die Realisierung) bekannt sein muss?
- Welche **Organisationsform** soll der Container aufweisen (Stack, Keller, ...)?

## Inhalt eines Containers

Ein Container beinhaltet eine Menge von Objekten. Die Organisation des Containers kann unterschiedlich ausfallen.

Betrachten wir zwei Container `c1` und `c2` und ein Objekt `obj`. Die Container sollen eine Elementfunktion `ein fuegen(obj)` zur Einfügen von Objekten haben; die Basisklasse der Objekte soll eine Methode `veraendern()` besitzen, mit der der Zustand eines Objektes änderbar ist.

Was passiert nach folgenden Schritten:

```
OBJ_Container c1;  
OBJ obj;  
c1.ein fuegen(obj);  
obj.veraendern();
```

Soll das Objekt **im Container verändert** sein oder ist im Container eine **Kopie** von `obj`?

```
OBJ_Container c1, c2;  
OBJ obj;  
c1.ein fuegen(obj);  
c2.ein fuegen(obj);  
obj.veraendern();
```

Soll nun das Objekt in `c1` **und** `c2` geändert sein?

```
OBJ_Container c;  
{  
    OBJ obj;  
    c.einfuegen(obj);  
}
```

Das Objekt `obj` wird beim Verlassen des Blocks gelöscht. Es kann im Container `c` nur dann existieren, wenn es als Kopie eingefügt wurde und nicht nur die Referenz durch `c.einfuegen()` gespeichert wurde.

Die o.a. Probleme lassen zwei Möglichkeiten der Implementierung von Container zu:

1. Im Container wird immer eine **Kopie eines Objektes** angelegt. Dann ist die **Arraysemantik** umgesetzt: in ein Arrayfeld ist stets ein Wert kopiert.
2. Die **Objektidentität** bleibt erhalten, wenn im Container lediglich ein **Zeiger** auf ein Objekt abgelegt ist.

Wenn die **Elemente** eines Containers **unterschiedliche Typen** haben sollen zwischen denen Vererbungsbeziehungen existieren, so ist normalerweise ein Speichern von Zeigern erforderlich:

Nehmen wir an wir hätten einen Container für `GrpObj`-Objekte. Dann kann kein `Rechteck` gespeichert werden, da nur Attribute für den Bezugspunkt, aber keine für Höhe und Breite verfügbar sind.

In diesem Fall sollte der Container **Zeiger** auf die **Basisklasse** `GrpObj` speichern und wegen Polymorphie auch auf Rechtecke zeigen.

**Container** werden i.A. als **Templates** realisiert, um über den Parameter den Typ der Objekte bestimmen zu können.

## Zugriff auf die Elemente

Wenn auf ein Element des Containers **zugegriffen** werden soll, muss unterschieden werden, ob der Container eine **Kopie des Elementes** oder eine **Referenz** auf das Element zurück gibt.

Wird **häufig** auf das selbe Element **zugegriffen**, ist es besser eine **Referenz** zurück zu geben, da dies gerade bei großen Objekten den Kopieraufwand minimiert. Weiterhin kann die **Änderung im Container** erfolgen und ist von **außen** sichtbar.

Andererseits stellt dies eine **Fehlerquelle** dar, da jede Änderung im Container automatisch auf alle Zugriffsstellen wirkt.

## Wachstum

Die Realisierung eines Behälters sollte so sein, dass der **Container sich selbst** bzgl. des benötigten Speicherplatzes **organisiert**, ohne dass der Aufrufer dies explizit durch Methodenaufrufe tun muss.

Wir werden im Folgenden einige Container mit unterschiedlicher Organisationsform realisieren und Besonderheiten dabei zeigen.

## 2.Container für doppelt verkettete Listen

Hier nun ein Container, der als doppelt verkettete Liste organisiert ist. Der Elementtyp ist `variable`, weil Templates verwendet werden.

Die Entwurfsentscheidung „Kopie oder Zeiger“ wurde zu Gunsten der **Kopie** getroffen, als Zugriffsmethode sind Funktionen realisiert, die eine **Referenz** auf die enthaltenen Daten zurück liefern. Das Wachstum wird selbst organisiert, da der benötigte Speicherplatz dynamisch (`new`, `delete`) verwaltet wird.

In der Klasse für den Container `Liste` wird eine innere Klasse `Listenelement` verwendet, in dem die eigentlichen Daten und die Verkettung gespeichert sind.

```
$ cat liste.t
// Template für doppelt verkettete Liste
// T ist Platzhalter für Datentyp eines Listenelements
// Listenelemente sind echte Kopien, keine Verweise
```

```
template<class T>
class Liste {
public:
    Liste();
    Liste(const Liste&);
    virtual ~Liste();
    Liste& operator=(const Liste&);    // Zuweisungsoperator

    bool empty() const { return Anzahl == 0;}
    int size() const { return Anzahl;}

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

    // am Anfang bzw. Ende löschen
    void pop_front();
    void pop_back();

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

    // anwenden der Funktion f() auf alle Elemente
    void anwenden(void(*f)(T&)) const;
```

```
private:
    class Listenelement {
    public:
        T Daten;
        Listenelement *Naechstes, *Vorgaenger;
        Listenelement(const T& dat)
            : Daten(dat), Naechstes(NULL), Vorgaenger(NULL) {}
    };
    Listenelement *Ende, *Anfang;
    int Anzahl;
};

template<class T>
inline Liste<T>::Liste() // Konstruktor
:Ende(NULL), Anfang(NULL), Anzahl(0)
{}
}
```

```
template<class T>
inline Liste<T>::Liste(const Liste<T>& L) // Kopierkonstruktor
:Ende(NULL), Anfang(NULL), Anzahl(0) {
    Listenelement *temp = L.Ende;
    while(temp) {
        push_front(temp->Daten);
        temp = temp->Vorgaenger;
    }
}

template<class T>
inline Liste<T>::~~Liste() { // Destruktor
    while(!empty()) pop_front();
}
```

```
template<class T>
inline Liste<T>& Liste<T>::operator=(const Liste<T>& L) { // Zuweisungsoperator
    if (this != &L) {
        while(!empty())
            pop_front(); // alles löschen

        // ... und neu aufbauen
        Listenelement *temp = L.Ende;
        while(temp) {
            push_front(temp->Daten);
            temp = temp->Vorgaenger;
        }
    }
    return *this;
}

template<class T>
inline void Liste<T>::push_front(const T& Dat) {
    Listenelement *temp = new Listenelement(Dat);
    // temp->Vorgaenger = NULL;
    // automatisch durch Konstruktor
    temp->Naechstes = Anfang;
    if (!Anfang) Ende = temp; // falls einziges Element
    else Anfang->Vorgaenger = temp;
    Anfang = temp;
    Anzahl++;
}
```

```
template<class T>
inline void Liste<T>::push_back(const T& Dat) {
    Listenelement *temp = new Listenelement(Dat);
    temp->Vorgaenger = Ende;
    temp->Naechstes = NULL;
    if (!Ende) Anfang=temp;          // einziges Element
    else temp->Vorgaenger->Naechstes = temp;
    Ende = temp;
    Anzahl++;
}
template<class T>
inline void Liste<T>::pop_front() {
    assert(!empty());
    Listenelement *temp = Anfang;
    Anfang = temp->Naechstes;
    if (!Anfang) // d.h. kein weiteres Element vorhanden
        Ende = NULL;
    else Anfang->Vorgaenger = NULL;
    delete temp;
    Anzahl--;
}
```

```
template<class T>
inline void Liste<T>::pop_back() {
    assert(!empty());
    Listenelement *temp = Ende;
    Ende = temp->Vorgaenger;
    if (!Ende) // d.h. kein weiteres Element vorhanden
        Anfang = NULL;
    else
        Ende->Naechstes = NULL;
    delete temp;
    Anzahl--;
}

template<class T>
inline T& Liste<T>::front() {
    assert(!empty());
    return Anfang->Daten;
}
```

```
template<class T>
inline const T& Liste<T>::front() const {
    assert(!empty());
    return Anfang->Daten;
}

template<class T>
inline T& Liste<T>::back() {
    assert(!empty());
    return Ende->Daten;
}

template<class T>
inline const T& Liste<T>::back() const {
    assert(!empty());
    return Ende->Daten;
}
```

```
template<class T>
// Funktion f auf alle Elemente anwenden
inline void Liste<T>::anwenden(void(*f)(T&)) const {
    Listenelement *temp=Anfang;
    while(temp) {
        f(temp->Daten);    // Anwenden der Funktion f auf ALLE Elemente der Liste
        temp = temp->Naechstes;
    }
}
#endif
$
```

Eine Testprogramm, das den o.a. **Container** mit **Strings** verwendet, definiert eine Funktion drucken, die dann auf alle Elemente angewendet wird.

```
$ cat main.cpp
#include<string>
#include<iostream>
#include"liste.t"
using namespace std;

void drucken(string& s) {
    cout << s << endl;
}
```

```
int main() {  
    Liste<string> Stringliste;  
    Stringliste.push_front( string("eins"));  
    Stringliste.push_front( string("zwei"));  
    Stringliste.push_front( string("drei"));  
    Stringliste.anwenden(drucken);           // Liste anzeigen  
  
    string buf;           // String-Objekt  
  
    while(!Stringliste.empty()) {  
        buf = Stringliste.front();  
        Stringliste.pop_front();  
        cout << "\n Element " << buf << " entnommen";  
        cout << "\n noch " << Stringliste.size()  
            << " Element(e) vorhanden! Restliste:\n";  
        Stringliste.anwenden(drucken);  
    }  
    cout << "Liste ist leer!" << endl;  
}  
$
```

Verwendet wird ausschließlich der Container, d.h. das **Listenelement** bleibt **verdeckt**.

## Hörsaalübung

Die Funktion `drucken()` hat einen Referenzparameter.

```
void drucken(string& s) {  
    cout << s << endl;  
}
```

An welcher Stelle müsste der Container geändert werden, damit die Funktion mit Wertübergabe arbeitet?

```
void drucken(string s) {  
    cout << s << endl;  
}
```

### 3.Container für Warteschlangen

Den ADT Schlange (Queue) haben wir bereits diskutiert. Jetzt soll ein Container für eine Schlange realisiert werden, der den Container `Liste` verwendet.

Wir betrachten zwei Lösungsvarianten:

- Delegation
- Private Vererbung

#### Delegation

Eine Idee zur Realisierung ist es , dass wir die Aufgaben (Methoden) einer Schlange einfach an eine Liste **delegieren**, d.h der **Container** `queue` verwendet ein **privates Attribut vom Typ** `Liste` und benutzt die **öffentlichen Methoden** von `Liste`, die für die Funktionalität einer Schlange erforderlich sind, die anderen werden nicht verwendet.

```
$ cat Delegation/queue.t
// queue.t      Warteschlangen-Template
#ifdef queue_t
#define queue_t queue_t
#include"liste.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);}           // am Ende einfügen
    void pop()              { L.pop_front();}           // am Anfang entnehmen

    T&      front()         { return L.front();}         // am Anfang bzw. Ende lesen
    const T& front() const { return L.front();}
    T&      back()          { return L.back();}
    const T& back() const   { return L.back();}

    void anwenden(void(*f)(T)) const {           // f auf alle Elemente anwenden
        L.anwenden(f);
    }

private:
    ►Liste<T> L;
};
#endif
$

```

Durch die Delegation enthält jedes **Queue-Objekt ein privates Liste-Objekt**, das „die Arbeit macht“, d.h. insbesondere die Methoden `push()` und `pop()` aber auch die komponentenweise Zuweisung durch den **überladenen Zuweisungsoperator** der Liste.

Damit kann eine Warteschlange in einem Programm verwendet werden:

```
$ cat Delegation/main.cpp
#include"queue.t"
#include<iostream>
using namespace std;
void drucken(int x) {
    cout.width(4);
    cout << x ;
}
int main() {
    Queue<int> Q;

    cout << "push:\n";
    for(int i=0; i<10; ++i) {
        Q.push(i*i);
    }
    Q.anwenden(drucken); cout << endl;

    cout << "5 pop:\n";
    for(int i=0; i<5; ++i) {
        cout << "front()=" << Q.front() <<endl;
        Q.pop();
    }
    Q.anwenden(drucken); cout << endl;
}
$
```

## Implementierungsvererbung

Durch Delegation konnte Code wiederverwendet werden. Eine weitere Möglichkeit der Wiederverwendung ist die Implementierungsvererbung (auch private Vererbung genannt).

Bei der Implementierungsvererbung dürfen **öffentliche Methoden** der Oberklasse von der Unterklasse verwendet werden. Abgeleitete **Objekten** (der Unterklasse) können aber die Oberklassenmethoden **nicht direkt** verwenden; **vererbt wird nicht die Schnittstelle, sondern nur die Implementierung**.

Sollen Oberklassenmethoden von Objekten abgeleiteter Klassen nutzbar sein (hier für Queue-Objekte) **müssen sie mit `using` gekennzeichnet** werden (engl. using declaration).

Diese *Using-Deklaration* besteht nur aus dem Namen der Methode ohne Parameter und Returntyp. Somit sind nur die durch using gekennzeichneten Methoden der Oberklasse in der Unterklasse verwendbar, die anderen nicht, obwohl sie öffentlich sind.

```
$ cat Implementierungsvererbung/queue.t
// queue.t      Warteschlangen-Template mit Implementierungsvererbung
#include"liste.t"
template<class T>

class Queue : private Liste<T> {
    public:
        using Liste<T>::empty;
        using Liste<T>::size;
        void push(const T& x) { Liste<T>::push_back(x); } // am Ende einfügen
        void pop()           { Liste<T>::pop_front(); } // am Anfang entnehmen
        using Liste<T>::front;           // am Anfang bzw. Ende lesen
        using Liste<T>::back;
        using Liste<T>::anwenden;
};
$
```

Also: `push_back`, `pop_front` darf innerhalb der abgeleiteten Klasse verwendet werden, nicht aber von Objekten der abgeleiteten Klasse:

```
Queue<int> q; q.push_back(1); // nicht möglich
Queue<int> q; cout << q.front(); // möglich weil mit using gekennzeichnet
```

## Hörsaalübung

Realisieren Sie einen Container für den ADT `Stack` unter Verwendung des Containers für `Liste` und benutzen Sie die

1. Delegation
  2. private Vererbung
- als Mechanismus.

## 4. Iteratoren

Im Beispiel der Klasse `Liste` gab es eine Methode `anwenden()`, die **alle** Listenelemente bearbeitet hat (gemäß einer als Argument übergebenen Funktion).

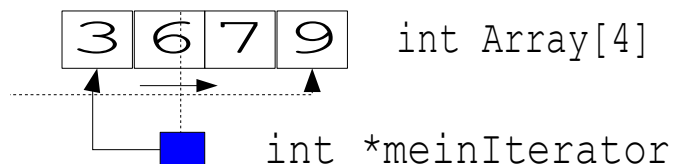
Sollen nur **einige Elemente** bearbeitet werden oder soll die Bearbeitung für einzelne Elemente unterschiedliche sein, so ist dies mit der Methode `anwenden()` so nicht möglich.

Benötigt wird ein Mechanismus, der **Elemente** eines **Containers sequentiell bearbeitet** und die **interne Struktur** vom Aufrufer **verbirgt**; also auch für Bäume oder andere Datenstrukturen geeignet ist. Ein solcher Mechanismus wird **Iterator** genannt.

Ein Iterator ist kein C++ Sprachmittel, es ist eine Art, die sequentielle Bearbeitung zu programmieren. Sie findet in der Realisierung der C++ Standard Template Library (STL) Anwendung.

Wir werden nun einen Iterator für Listen selbst programmieren, aber zuerst ein Beispiel ohne OO zur Einführung diskutieren.

Das folgende Beispiel verwendet einen `int`-Zeiger zur Realisierung eines Iterators:



```
$ cat einfacherZeiger/main.cpp
// Einfaches Iteratorbeispiel
typedef int* MeinIterator;           // MeinIterator ist ein Zeiger auf int
typedef void (*Funktion)(int&);     // Funktion ist Zeiger auf void-Funktion
                                     // mit int Argument

void drucken(int &x) {                // global
    cout.width(4);
    cout << x ;
}

void anwenden(MeinIterator Anfang, MeinIterator Ende, Funktion f) {
    while(Anfang != Ende) {
        f(*Anfang);
        Anfang++;
    }
}

int main() {
    const int Anz = 10;
    int Array[Anz];
    for (int i = 0; i < Anz; ++i) Array[i] = i*i;           // initialisieren

    MeinIterator von = Array,           // = &Array[0]
        bis = von + Anz;           // = Position NACH letztem Element
    anwenden(von, bis, drucken);
    cout << endl;
}
$
```

Im o.a. Beispiel wird also mittels `meinIterator`, also `int`-Zeigern eine Datenstruktur sequentiell durchlaufen. Dabei zeigt **von an den Anfang, bis an die Position direkt hinter die Datenstruktur.**

Die Funktion `anwenden` druckt ein Element und schaltet den Iterator (`Anfang`) um eins weiter.

In der **STL** gibt es eine Klasse, die einen Iterator bereit stellt, der so wie das letzte Beispiel funktioniert: die Klasse `vector`:

```
$ cat einfacherZeiger/main2.cpp
#include<iostream>
#include<vector>
using namespace std;

void drucken(int &x) { // global
    cout.width(4);
    cout << x ;
}
int main() {
    vector<int> v(10);
    for (size_t i = 0; i < v.size(); ++i)
        v[i] = i*i; // initialisieren
    for_each(v.begin(), v.end(), drucken);
}
$
```

Im `vector`-Beispiel sieht man, dass die Grenzen einfach durch `begin()` und `end()` geklammert sind, die Methode `for_each()` entspricht der Methode `anwenden()`.

Die Implementierung von `anwenden()` bzw. `for_each()` lassen erkennen, dass man zur Realisierung von Iteratoren folgende Operationen programmieren muss:

```
void anwenden(MeinIterator Anfang, MeinIterator Ende, Funktion f) {  
    while(Anfang != Ende) {  
        f(*Anfang);  
        Anfang++;  
    }  
}
```

Vergleichsoperator

Dereferenzierungsoperator

Inkrementoperator

**Also: Ein Iterator ist ein Zeiger, die die o.a. Operatoren überladen hat.**

## Iterator für die Klasse `Liste`

Die Realisierung eines Iterators für die Klasse `Liste` muss also die o.a. Operationen implementieren. Wir **erweitern** die **Klasse `Liste`** um eine **innere Klasse `Iterator`**, die die Operationen implementiert.

Die Klasse `Iterator` verwendet ein privates Attribut `aktuell` um auf das Listenelement zu zeigen, das der **Iterator bearbeiten** soll.

```
$ cat IteratorListe/liste.t
//      wie bisher, erweitert um Iterator, begin(), end()

// Listen-Template für doppelt verkettete Liste
// T = Platzhalter für Datentyp eines Listenelements
// Listenelemente sind echte Kopien, keine Verweise
```

```
/*Der g++-Compiler versteht keine public-definierten Typen ausserhalb
  der Klasse in einer Parameterliste. Deshalb sind alle Funktionen,
  die solche Typen in der Parameterliste haben,
  inline in der Klassendefinition enthalten. */

#include<cstddef>           // NULL
#include<cassert>

template<class T>
class Liste {
    public:
        Liste();
        Liste(const Liste&);
        virtual ~Liste();
        Liste& operator=(const Liste&);

        bool empty() const { return Anzahl == 0;}
        int size() const { return Anzahl;}

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

        void pop_front(); // am Anfang bzw. Ende entnehmen
        void pop_back();

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

```

const T& back() const;

private:
struct Listenelement {           // hier struct als Kurzschreibweise
    T Daten;
    Listenelement *Naechstes, *Vorgaenger;
    Listenelement(): Naechstes(NULL), Vorgaenger(NULL) {}
};

Listenelement *Ende, *Anfang;
int Anzahl;
public:
class Iterator {
private:
    Listenelement* aktuell;           // Zeiger auf aktuelles Element
public:
    friend class Liste<T>;           // wg. insert
    Iterator(Listenelement* Init = NULL)
    : aktuell(Init) {
    }

    Iterator(const Liste& L) {
        *this = L.begin();
    }

    const T& operator*() const {           // Dereferenzierung
        return aktuell->Daten;
    }
}

```

Im Konstruktor wird der Zeiger aktuell initialisiert.

Ein neuer Iterator wird auf den Beginn der Liste gesetzt.

```
T& operator*() { // Dereferenzierung
    return aktuell->Daten;
}

Iterator& operator++() { // prefix
    if(aktuell)
        aktuell = aktuell->Naechstes;
    return *this;
}

Iterator operator++(int) // postfix
{ Iterator temp = *this;
  ++*this;
  return temp;
}

bool operator==(const Iterator& x) const
{ return aktuell == x.aktuell;
}

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

}; // Iterator
```

```
// Methoden von Liste, die die Klasse Iterator benutzen:
```

```
Iterator begin() const {
    return Iterator(Anfang);
}

Iterator end() const {
    return Iterator(); // NULL-Iterator
}

// Vor pos einfügen
Iterator insert(Iterator pos, const T& Wert) {
    if(pos == begin()) {
        push_front(Wert);
        return Iterator(Anfang);
    }

    if(pos == end()) {
        push_back(Wert);
        return Iterator(Ende);
    }

    // zwischen 2 Elementen einketten
    Listenelement *temp = new Listenelement;
    temp->Daten = Wert;
    temp->Naechstes = pos.aktuell;
    temp->Vorgaenger = pos.aktuell->Vorgaenger;
    pos.aktuell->Vorgaenger->Naechstes=temp;
    pos.aktuell->Vorgaenger = temp;
    Anzahl++;
}
```

```
        return Iterator(temp);
    }
};

/* ===== Implementierung der Klasse Liste ===== */

template<class T>
inline Liste<T>::Liste() // Konstruktor
:Ende(NULL), Anfang(NULL), Anzahl(0) {
}

template<class T>
inline Liste<T>::Liste(const Liste<T>& L) // Kopierkonstruktor
:Ende(NULL), Anfang(NULL), Anzahl(0) {
    Listenelement *temp = L.Ende;
    while(temp) {
        push_front(temp->Daten);
        temp = temp->Vorgaenger;
    }
}

template<class T>
inline Liste<T>::~~Liste() // Destruktor
while(!empty()) pop_front();
}

template<class T> // Zuweisungsoperator
inline Liste<T>& Liste<T>::operator=(const Liste<T>& L) {
```

```
if (this != &L) {
    while(!empty()) pop_front(); // alles loechen
    // ... und neu aufbauen
    Listenelement *temp = L.Ende;
    while(temp) {
        push_front(temp->Daten);
        temp = temp->Vorgaenger;
    }
}
return *this;
}

template<class T>
inline void Liste<T>::push_front(const T& Dat) {
    Listenelement *temp = new Listenelement;
    temp->Daten = Dat;
    temp->Vorgaenger = NULL;
    temp->Naechstes = Anfang;
    if (!Anfang) Ende=temp; // einziges Element
    else Anfang->Vorgaenger = temp;
    Anfang = temp;
    Anzahl++;
}
```

```
template<class T>
inline void Liste<T>::push_back(const T& Dat) {
    Listenelement *temp = new Listenelement;
    temp->Daten = Dat;
    temp->Vorgaenger = Ende;
    temp->Naechstes = NULL;
    if (!Ende) Anfang=temp;          // einziges Element
    else temp->Vorgaenger->Naechstes = temp;
    Ende = temp;
    Anzahl++;
}

template<class T>
inline void Liste<T>::pop_front() {
    assert(!empty());
    Listenelement *temp = Anfang;
    Anfang = temp->Naechstes;
    if (!Anfang) Ende = NULL;      // d.h. kein weiteres Element vorhanden
    else Anfang->Vorgaenger = NULL;
    delete temp;
    Anzahl--;
}

template<class T>
inline void Liste<T>::pop_back() {
    assert(!empty());
    Listenelement *temp = Ende;
    Ende = temp->Vorgaenger;
```

```
    if (!Ende) Anfang = NULL;    // d.h. kein weiteres Element vorhanden
    else Ende->Naechstes = NULL;
    delete temp;
    Anzahl--;
}

template<class T>
inline T& Liste<T>::front() {
    assert(!empty());
    return Anfang->Daten;
}

template<class T>
inline const T& Liste<T>::front() const {
    assert(!empty());
    return Anfang->Daten;
}

template<class T>
inline T& Liste<T>::back() {
    assert(!empty());
    return Ende->Daten;
}
```

```

template<class T>
inline const T& Liste<T>::back() const {
    assert(!empty());
    return Ende->Daten;
}
$

```

Der so definierte Iterator kann nun in einer Anwendung verwendet werden; wir wollen eine `int-Liste` haben und einige Werte in sie schreiben. Wie in den letzten Beispielen programmieren wir eine **for\_each-Funktion**, die den **Iterator** verwendet, um die **Elemente** zu **drucken**.

```

$ cat IteratorListe/main.cpp
#include"liste.t" // erweitert um Iterator
#include<iostream>
using namespace std;

void drucken(int& x) { // wie vorher
    cout.width(4);
    cout << x ;
}

```

my als Präfix notwendig wg.  
Namenskollision in STL

```

template<class Iterator, class Funktion> // wie vorher
void my_for_each(Iterator Anfang, Iterator Ende, Funktion f) {
    while(Anfang != Ende) f(*Anfang++);
}

```

Verwendung der  
überschriebenen Operatoren

```

int main() {
    int i,x;
    Liste<int> L; // Liste mit int-Zahlen

    cout << "push_front:\n";
    for (i = 0; i < 10; ++i) {
        x = i*i;
        L.push_front(x); // mit Quadratzahlen 0..81 füllen
    }

    Liste<int>::Iterator ListIter(L);

    cout << "*ListIter =" << *ListIter << endl;
    cout << "ListIter++;" << endl;
    ListIter++;
    cout << "*ListIter =" << *ListIter << endl;

    // Ausgabe
    my_for_each(L.begin(), L.end(), drucken);
    cout << endl;
}
$

```

ListIter ist ein Iterator, der durch den Konstruktor mit der Liste L verknüpft wird.

hier wird my\_for\_each verwendet, um alle Elemente zu drucken. Die Methoden begin() und end() geben Iteratoren auf Anfang bzw. Ende zurück.

```

$ main
push_front:
*ListIter =81
ListIter++;
*ListIter =64
 81 64 49 36 25 16 9 4 1 0
$

```

## Hörsaalübung:

Schreiben Sie eine Anwendung, die mittels der Klasse `Liste` eine `long`-Liste erzeugt. Die Anwendung soll 5 `long`-Werte vom Benutzer einlesen und durch die Methode `insert` in der Liste jeweils am Ende speichern.

Dann soll die Liste mittels Iterator ausgegeben werden.

In vielen Anwendungsfällen ist eine Datenstruktur nicht nur sequentiell zu durchlaufen (durch Iteratoren), sondern man muss von der aktuellen Position vor- und rückwärts wandern können. In diesen Fällen braucht man keinen Iterator, es reicht aus, Operationen ++ und -- zu realisieren, die den Zeiger „aktuell“ entsprechend verschieben. Diesen Zeiger nennt man **Cursor**.

Es existiert dann **genau ein Cursor** für ein **Objekt**. Der **Vorteil** von **Iteratoren** ist es, dass es **beliebig viele** Iteratoren **für eine Datenstruktur** geben kann, die unabhängig voneinander gleichzeitig verwendbar sind.

Container werden auch verwendet, um die **Schnittstelle** von der **Implementierung trennen** zu können. So wäre es bei den letzten Testprogrammen denkbar, eine Klasse Baum mit den selben öffentlichen Methoden und einem Iterator zu realisieren und diese Klasse anstelle der Klasse Liste zu verwenden, ohne das Testprogramm ändern zu müssen (Schöne Aufgabe für zu Hause!).

## 5.typename

Programmiert man komplexere Container mit Templates, dann entstehen u.U. Mehrdeutigkeiten, die der Compiler nicht auflösen kann. Dann muss der Programmierer die Mehrdeutigkeit mittels „typename“ auflösen.

### Beispiel (einfache Liste)

```
template<class T>
class L {          // simple list
public:
    L() { start = NULL; }
    void insert(T t) { // insert front
        listElem *p;
        p = new listElem;
        p->data = t; p->next = NULL;
        if (start != NULL)
            p->next = start;
        start = p;
    }
protected:
    struct listElem {
        listElem * next;
        T data;
    };
    listElem *start;
};
```

```
template<class T>
class S : public L<T> {
public:
    void print() {
        L<T>::listElem *p;
        p = L<T>::start;
        while (p != NULL) {
            cout << p->data << " ";
            p = p->next;
        }
        cout << endl;
    }
};

int main(int argc, char** argv) {
    S<int> s;
    for (int i=1; i<=5; i++)
        s.insert(i);
    s.print();
    return 0;
}

$ g++ typename.cpp
main.cpp: In member function 'void S<T>::print()':
main.cpp:40: error: 'p' was not declared in this scope
main.cpp:40: error: dependent-name 'L<T>::listElem' is parsed as a non-type, but
instantiation yields a type
main.cpp:40: note: say 'typename L<T>::listElem' if a type is meant
```

Die Fehlermeldung besagt, dass abhängige Namen (Namen in Templates, die vom Template-Parameter abhängen) als Typ gekennzeichnet werden müssen.

Dazu dient das Schlüsselwort „typename“:

```
template<class T>
class S : public L<T> {
public:
    void print() {
        typename L<T>::listElem *p;
        p = L<T>::start;
        while (p != NULL) {
            cout << p->data << " ";
            p = p->next;
        }
        cout << endl;
    }
};
```