# Einführung in die Programmierung mit C++

## Dynamische Datenstrukturen (Vektoren und Listen)

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

# Inhalt

# Outline

# Dynamische Vektoren

## Definition



```
1  #pragma once
2
3  #include <cstdlib>
4
5  template<typename T>
6  class vector {
7    T* v; size_t n;
8    void grow();
9    void shrink();
10  public:
11   vector(size_t, const T&);
12   ~vector();
13   void insert(const T&, size_t);
14   void erase(size_t);
15   void print() const;
16 };
17
18 #include<vector.cpp>
```

- Wir betrachten eine (naive) Implementierung dynamisch wachsender / schrumpfender Vektoren mit Elementen variabler Typen.
- Bei Konstruktion für die gegebene Größe wird mit einem gegebenen Wert initialisiert.
- Ein Destruktor wird benötigt.
- Einfügen (insert) von Elementen führt zum Wachsen (grow) um jeweils ein Element.
- Entfernen (erase) von Elementen führt zum Schrumpfen (shrink) um jeweils ein Element.
- Vektoren werden mittels print auf den Bildschirm ausgegeben.

# Dynamische Vektoren

## Konstruktion. Destruktion. Ausgabe



```
1  template<typename T>
2  vector<T>::vector(size_t n, const T &d)
3      : v(new T[n]), n(n) {
4    for (size_t i=0;i<n;i++) v[i]=d;
5  }
6
7  template<typename T>
8  vector<T>::~vector() {
9    delete [] v;
10 }
11
12 template<typename T>
13 void vector<T>::print() const {
14   std::cout << "[ ";
15   for (size_t i=0;i<n;i++)
16     std::cout << v[i] << " ";
17   std::cout << "]" << std::endl;
18 }
```

- Der Konstruktor alloziert dynamischen Speicher der gegebenen Größe für den gegebenen Elementtyp, speichert die Größe des entsprechenden dynamischen Feldes und initialisiert die Elemente mit `d`.
- Der Destruktor gibt den belegten dynamischen Speicher frei.
- Die Elemente von Vektoren werden durch `print` in eckige Klammern eingeschlossen und durch Leerzeichen getrennt auf den Bildschirm ausgegeben.

# Dynamische Vektoren

## Wachsen und Schrumpfen



```
1  template<typename T>
2  void vector<T>::grow() {
3    T* d=new T[n+1];
4    for (size_t i=0;i<n;i++) d[i]=v[i];
5    delete [] v;
6    v=d;
7    n++;
8  }
9
10 template<typename T>
11 void vector<T>::shrink() {
12   T* d=new T[n-1];
13   for (size_t i=0;i<n-1;i++) d[i]=v[i];
14   delete [] v;
15   v=d;
16   n--;
17 }
```

- Bei Einfügen neuer Elemente muss der Vektor wachsen (grow) $\Rightarrow$ ineffiziente Reallozierung und Kopieren von Daten
- Bei Enfernen von Elementen muss der Vektor schrumpfen (shrink) $\Rightarrow$ ineffiziente Reallozierung und Kopieren von Daten
- Das Wachsen sollte besser nicht elementweise sondern mit Teilvektoren fixer Länge geschehen. Das Schrumpfen könnte durch eine entsprechende Komprimierungsfunktion implementiert werden, welche nur bei Bedarf aufgerufen wird.

# Dynamische Vektoren

## Einfügen und Entfernen



```
1  template<typename T>
2  void vector<T>::insert(
3        const T &d, size_t pos) {
4    if (pos>=n) throw std::runtime_error("insert:
       out of range");
5    else {
6      grow();
7      for (size_t i=n-1;i>pos;i--)
8        v[i]=v[i-1];
9      v[pos]=d;
10   }
11 }
12
13 template<typename T>
14 void vector<T>::erase(size_t pos) {
15   if (pos>=n) throw std::runtime_error("erase:
       out of range");
16   else {
17     for (size_t i=pos;i<n-1;i++) v[i]=v[i+1];
18     shrink();
19   }
20 }
```

- Einfügen und Entfernen von Elementen ist nur an den Positionen 0 ... n-1 möglich; sonst "Wurf" einer Ausnahme
- Bei Einfügen wächst das dynamische Feld gefolgt von der Generierung der benötigten Lücke und dem Einfügen des neuen Elements in diese.
- Entfernen eines Elements erfolgt durch Aufrücken der Folgeelemente gefolgt vom Schrumpfen des dynamischen Feldes durch Deallozierung des letzten Elements.

# Dynamische Vektoren

## Fallstudie



```
1  #include "vector.hpp"
2  #include <iostream>
3
4  int main(int argc, char* argv[]) {
5    if (argc!=2) throw "argc!=2";
6    int n=std::stoi(argv[1]);
7    if (n<0) throw "n<0";
8    try {
9      vector<int> v(n,1);
10     try {
11       for (int i=0;i<n;i++) v.insert(i,i);
12       v.print();
13       for (int i=0;i<n;i++) v.erase(n);
14       v.print();
15     } catch (const std::runtime_error &e) {
16       std::cerr << e.what() << std::endl;
17     }
18   } catch (const std::bad_alloc &e) {
19     std::cerr << e.what() << std::endl;
20   }
21   return 0;
22 }
```

- Einlesen der Vektorlänge von Tastatur
- Allozierung ⇒ ggf. Ausnahme std::bad_alloc
- Einfügen von Elementen ⇒ ggf. Ausnahme std::runtime_error
- Löschen von Elementen ⇒ ggf. Ausnahme std::runtime_error
- Ausgabe der Vektorelemente auf Bildschirm
- Beispiel:

```
:-) ./main.exe 4
[ 0 1 2 3 1 1 1 1 ]
[ 0 1 2 3 ]
```

# AUFWACHEN!

Welchen Wert hat die Summe der Elemente des Vektors kurz vor Beendigung des Programms?

```
1  #include "vector.hpp"
2
3  int main() {
4    vector<int> v(1,0);
5    for (int i=1;i<10;i++) v.insert(i,0);
6    v.erase(6);
7    // sum of all entries = ???
8    return 0;
9  }
```

# Wach?

## Klar!

```
1  #include "vector.hpp"
2
3  int main() {
4    vector<int> v(1,0);
5    for (int i=1;i<10;i++) v.insert(i,0);
6    v.erase(6);
7    // sum of all entries = 42
8    v.print();
9    return 0;
10 }
```

Ausgabe:

```
[ 9 8 7 6 5 4 2 1 0 ]
```

# Outline

# Dynamische Listen

## Definition



```
1  template<typename T>
2  class list {
3
4    int l;
5    struct list_element {
6      T data; list_element* next;
7      list_element(const T&);
8    } *first;
9
10   void free(const list_element *);
11 public:
12   list(int);
13   ~list(); // iterative / recursive
14   int length() const ;
15   void erase(int);
16   void insert(int, const T&);
17   void set(int, const T&);
18   const T& get(int) const;
19   void to_dot(const char*) const;
20 };
21
22 #include "list.cpp"
```

- Wir betrachten einfach verkettete Listen von Elementen generischen Typs.
- Listenelemente enthalten typgenerische Daten und Zeiger auf das jeweils folgende Element. Der Einstieg in die Liste erfolgt über einen Zeiger first auf das erste Element.
- Konstruktion einer Liste gegebener Länge; Destruktion iterativ vs. rekursiv
- Konstruktion von Listenelementen und Initialisierung mit gegebenen Daten.
- Schnittstelle ermöglicht Einfügen / Enfernen von Elementen in die / aus der Liste, Schreib- / Lesezugriff auf Elemente
- Ausgabe in Datei im `graphviz`-Format

# Dynamische Listen

## Konstruktion



```
1  template<typename T>
2  list<T>::list::list_element::list_element(
3        const T& d)
4    : data(d), next(nullptr) {}
5
6  template<typename T>
7  list<T>::list(int length) : l(length) {
8    if (length<=0) throw(std::runtime_error("
       list: length<=0"));
9    list_element* last=new list_element(T());
10   first=last;
11   for (int i=1;i<length;i++) {
12     last->next=new list_element(T());
13     last=last->next;
14   }
15 }
```

- Konstruktion von Listenelementen und Initialisierung mit gegebenen Daten; nullptr-Terminierung
- Konstruktion einer Liste der Länge l und potentielle Initialisierung durch Standardkonstruktor für Typ der in Listenelementen gespeicherten Daten; Iteratives Anhängen des nächsten Elements an das aktuell letzte.
- Mindestlänge gleich 1

# Dynamische Listen

## Destruktion (Iterativ vs. Rekursiv)



### Iterativ:

```
1  template<typename T>
2  list<T>::~list() {
3    for (int i=0;i<l;i++) {
4      list_element* current=first;
5      first=first->next;
6      delete current;
7    }
8  }
```

- Verschieben von first auf das zweite Element gefolgt vom Löschen des ersten Elements
- Elemente werden entsprechend ihrer Reihenfolge gelöscht.
- Alternativ: **while** (first)...

### Rekursiv:

```
1  template<typename T>
2  void list<T>::free(const list_element *e) {
3    if (e->next) free(e->next);
4    delete e;
5  }
6
7  template<typename T>
8  list<T>::~list() { free(first); }
```

- Rekursive Suche nach Ende der Liste $\Rightarrow$ letztes Element wird zuerst gelöscht.
- Elemente werden in umgekehrter Reihenfolge gelöscht.

# Dynamische Listen

## Schreib-/Lesezugriff



```
1  template<typename T>
2  void list<T>::set(int pos, const T& dat) {
3    if (pos<0||pos>=l) throw(std::runtime_error("
         erase: pos<0||pos>=l"));
4    list_element* current=first;
5    for (int i=1;i<=pos;i++)
6      current=current->next;
7    current->data=dat;
8  }
9
10 template<typename T>
11 const T& list<T>::get(int pos) const {
12   if (pos<0||pos>=l) throw(std::runtime_error("
         erase: pos<0||pos>=l"));
13   list_element* current=first;
14   for (int i=1;i<=pos;i++)
15     current=current->next;
16   return current->data;
17 }
18
19 template<typename T>
20 int list<T>::length() const { return l; }
```

- Schreib- / Lesezugriff erfolgt über Positionen 0<=pos<l.
- Ausnahme bei ungültiger Position
- Lesezugriff modifiziert den Zustand des aktuellen Objekts nicht → **const**
- Suche des Elements an Position pos iterativ von first (pos=0) beginnend
- Lesezugriff auf Länge der Liste mittels length()

# Dynamische Listen

## Einfügen von Elementen



```
1  template<typename T>
2  void list<T>::insert(int pos, const T& dat) {
3    if (pos<0||pos>=l) throw(std::runtime_error("
       erase: pos<0||pos>=l"));
4    list_element* new_element=
5      new list_element(dat);
6    if (pos==0) {
7      new_element->next=first;
8      first=new_element;
9    } else {
10     list_element* current=first;
11     for (int i=1;i<=pos-1;i++)
12       current=current->next;
13     new_element->next=current->next;
14     current->next=new_element;
15   }
16   l++;
17 }
```

- Einfügen an Stellen 0 ... l-1; sonst std::runtime_error
- Allozierung des neuen Listenelements und Initialisierung mit gegebenen Daten
- spezielles Einfügen am Anfang
- Einfügen an restlichen Positionen nach linearer Suche der Position und entsprechende Zeigermanipulation
- Inkrementierung der Länge der Liste

# Dynamische Listen

## Entfernen von Elementen. Ausgabe



```
1  template<typename T>
2  void list<T>::erase(int pos) {
3    if (pos<0||pos>=l) throw(std::runtime_error("
       erase: pos<0||pos>=l"));
4    list_element* current=first;
5    if (pos==0) {
6      first=first->next;
7      delete current;
8    } else {
9      for (int i=1;i<=pos-1;i++)
10       current=current->next;
11     list_element* old_element=current->next;
12     current->next=old_element->next;
13     delete old_element;
14   }
15    l--;
16 }
17
18 template<typename T>
19 void list<T>::to_dot(const char* filename) const;
```

- Entfernen an Stellen 0 ... l−1 sonst std::runtime_error
- spezielles Entfernen am Anfang
- Entfernen an restlichen Positionen nach linearer Suche der Position und entsprechende Zeigermanipulation
- Deallozierung des zu entfernenden Listenelements
- Dekrementierung der Länge der Liste
- Ausgabe im `graphviz`-Format

# Dynamische Listen

## Fallstudie

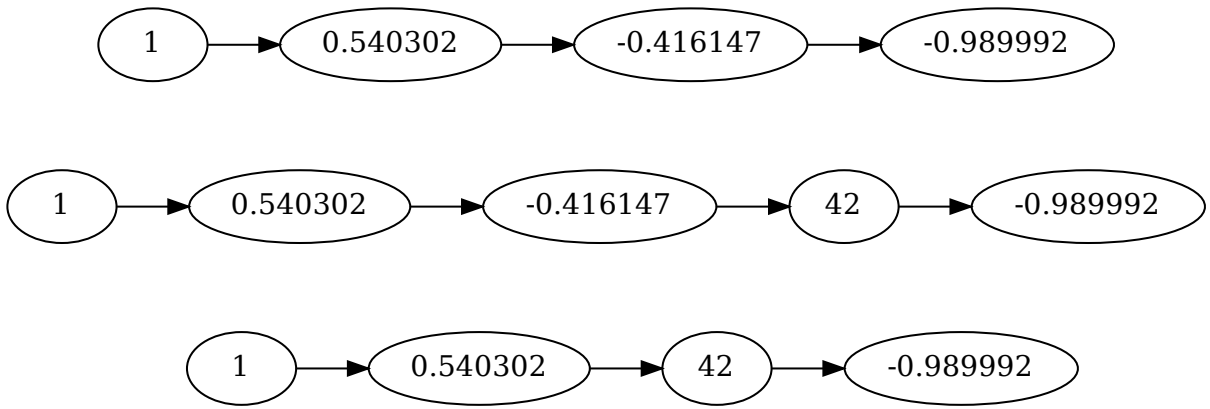


```
1  #include "list.hpp"
2
3  int main(int argc, char* argv[]) {
4    using namespace std;
5    if (argc!=4)
6      throw runtime_error("main: argc!=4");
7    try { // proper cleanup in case of error
8      list<float> l(stoi(argv[1]));
9      for (int i=0;i<l.length();i++)
10       l.set(i,cos(i));
11     l.to_dot("1.dot");
12     l.insert(stoi(argv[2]),42);
13     l.to_dot("2.dot");
14     l.erase(stoi(argv[3]));
15     l.to_dot("3.dot");
16   } catch (const runtime_error &e) {
17     cerr << "ERROR: "
18          << e.what() << endl;
19   }
20   return 0;
21 }
```

Bei Aufruf mit Kommandozeilenargumenten 4 3 2:

- Konstruktion einer Liste der Länge 4 und Initialisierung mit cos der jeweiligen Elementposition
- Einfügen des Elements 42 an Position 3 $\Rightarrow$ ggf. Ausnahme std::runtime_error (falls Länge $\leq 3$)
- Entfernen des Elements an Position 2 $\Rightarrow$ ggf. Ausnahme std::runtime_error (falls Länge $\leq 2$)
- Ausgabe der jeweiligen Zwischenzustände im `graphviz`-Format

- `./main.exe 4 3 2`




- `./main.exe 1 3 2`

  | ERROR: insert: pos<0||pos>=l

- `./main.exe 0 3 2`

  | ERROR: list: length<=0

# Dynamische Listen

## Doppelte Verkettung



```
1  struct l {
2
3    struct le {
4      int i=0;
5      le *prev=nullptr,*next=nullptr;
6      le()=default;
7      le(int i) : i(i) {}
8    } first,*last=&first;
9
10   l(int n) {
11     for (int i=1;i<n;i++) {
12       last->next=new le(i);
13       last->next->prev=last;
14       last=last->next;
15     }
16   }
17 ...
```

- Doppelt verkettete Listen ermöglichen effizienteres Navigieren (beliebige Kombinationen aus Vorwärts- und Rückwärtsschritten) auf Kosten eines höheren Speicherbedarfs (zusätzlicher Zeiger auf Vorgänger).
- Einstiegspunkte über erstes (first) bzw. über Zeiger auf letztes (last) Element
- Korrekte Verkettung bei Konstruktion einer Liste der Länge n

# Dynamische Listen

## Doppelte Verkettung und *pre-order* Navigation



```
1  ...
2    void forward(const le* e) {
3      // pre-order
4      std::cout << e << std::endl;
5      if (e->next) forward(e->next);
6    }
7    void forward() { forward(&first); }
8
9    void backward(const le* e) {
10     // pre-order
11     std::cout << e << std::endl;
12     if (e->prev) backward(e->prev);
13   }
14   void backward() { backward(last); }
15 };
16
17 int main() {
18   l x(3);
19   x.forward();
20   x.backward();
21   return 0;
22 }
```

- Rekursive Vorwärtsiteration über Einstiegspunkt &first
- Rekursive Rückwärtsiteration über Einstiegspunkt last
- Beispiel:

```
0x7fff321abeb0
0x55c28a216eb0
0x55c28a216ed0
0x55c28a216ed0
0x55c28a216eb0
0x7fff321abeb0
```

# Dynamische Listen

## Doppelte Verkettung und *post-order* Navigation



```
1  ...
2    void backward(const le* e) {
3      if (e->next) backward(e->next);
4      // post-order
5      std::cout << e << std::endl;
6    }
7    void backward() { backward(&first); }
8
9    void forward(const le* e) {
10     if (e->prev) forward(e->prev);
11     // post-order
12     std::cout << e << std::endl;
13   }
14   void forward() { forward(last); }
15 };
16
17 int main() {
18   l x(3);
19   x.forward();
20   x.backward();
21   return 0;
22 }
```

- Rekursive Rückwärtsiteration über Einstiegspunkt &first
- Rekursive Vorwärtsiteration über Einstiegspunkt last
- Beispiel:

```
0x7fff321abeb0
0x55c28a216eb0
0x55c28a216ed0
0x55c28a216ed0
0x55c28a216eb0
0x7fff321abeb0
```

# Outline

# Listen der Standardbibliothek

## Definition



```
1  #include<iostream>
2  #include<list>
3
4  template<typename T>
5  class list {
6    std::list<T> data;
7  public:
8    list(int);
9    int length() const ;
10   void erase(int);
11   void insert(int, const T&);
12   void set(int, const T&);
13   const T& get(int) const;
14   void algorithms();
15   void to_dot(const char*) const;
16 };
17
18 #include "list.cpp"
```

- Die Standardbibliothek stellt (element-)typgenerische doppelt verkettete Listen std::list inkl. einer reichhaltigen Schnittstelle zur Verfügung. Verwenden!
- Hier: Verpackung in nutzerdefinierter Klasse list
- Funktionalität analog der zuvor implementierten
- nützliche Algorithmen auf Listen in algorithms()

# Listen der Standardbibliothek

## Konstruktion und Lesezugriff auf Länge



```
1  template<typename T>
2  list<T>::list(int len) {
3    for (int i=0;i<len;i++)
4      data.push_back(T());
5  }
6
7  template<typename T>
8  int list<T>::length() const {
9    return data.size();
10 }
```

- dynamisches Wachstum einer initial leeren Liste mittels push_back(T()); potentielle Initialisierung der Listenelemente durch den Standardkonstruktor deren Datentyps
- Lesenzugriff auf Länge der Liste mittels Objektfunktion size()
- Weder Destruktor noch Kopierkonstruktor oder Zuweisungsoperator werden benötigt, da diese durch std::list bereitsgestellt werden $\Rightarrow$ *Rule of Zero* aus Nutzerperspektive.

# Listen der Standardbibliothek

## Einfügen / Entfernen von Elementen



```
1  template<typename T>
2  void list<T>::insert(int pos, const T& dat) {
3    if (pos<0||pos>=length())
4      throw(std::runtime_error("insert: pos<0||pos
           >=length"));
5    auto data_it=data.begin();
6    for (int i=1;i<=pos;i++,data_it++);
7    data.insert(data_it,dat);
8  }
9
10 template<typename T>
11 void list<T>::erase(int pos) {
12   if (pos<0||pos>=length())
13     throw(std::runtime_error("erase: pos<0||pos
           >=length"));
14   auto data_it=data.begin();
15   for (int i=1;i<=pos;i++,data_it++);
16   data.erase(data_it);
17 }
```

- std::list stellt eigene Methoden zum Einfügen und Entfernen von Elementen zur Verfügung.
- Positionen werden mittels Iteratoren implementiert.
- Umrechnung von **int** pos nach std::list<T>::iterator durch lineare (Vorwärts-)Suche vom ersten Element beginnend; alternativ: lineare (Rückwärts-)Suche vom letzten Element beginnend

# Listen der Standardbibliothek

## Elementzugriff



```
1  template<typename T>
2  void list<T>::set(int pos, const T& dat) {
3    if (pos<0||pos>=length())
4      throw(std::runtime_error("set: pos<0||
           pos>=length"));
5    auto data_it=data.begin();
6    for (int i=1;i<=pos;i++,data_it++);
7    *data_it=dat;
8  }
9
10 template<typename T>
11 const T& list<T>::get(int pos) const {
12   if (pos<0||pos>=length())
13     throw(std::runtime_error("get: pos<0||
           pos>=length"));
14   auto data_it=data.cbegin();
15   for (int i=1;i<=pos;i++,data_it++);
16   return *data_it;
17 }
```

- std::list verwendet Iteratoren zum Schreib- bzw Lesezugriff auf Listenelemente
- Beachte: Verwendung eines konstanten Iterators für Lesezugriff
- Wie zuvor: Umrechnung von **int** pos nach std::list<T>::iterator durch lineare Suche

# Listen der Standardbibliothek

## Fallstudie

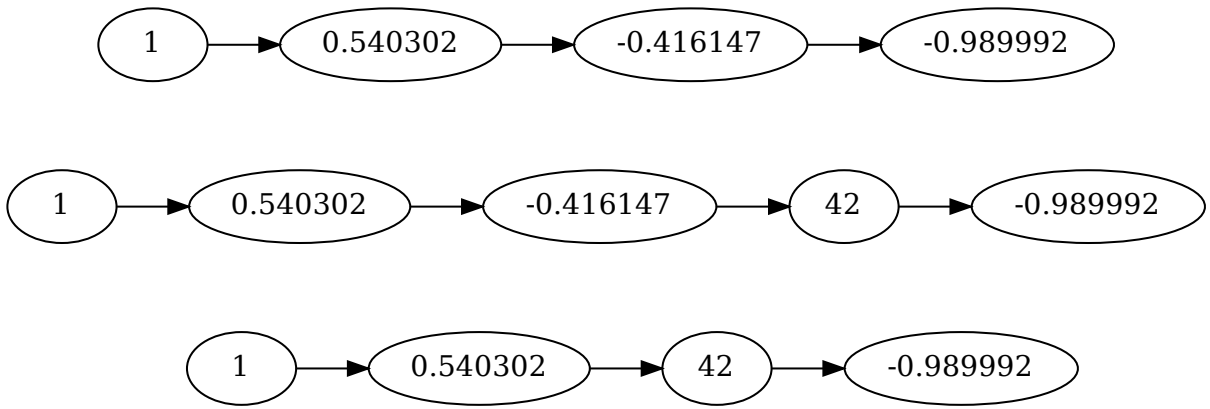


```
1  int main(int argc, char* argv[]) {
2    if (argc!=4)
3      throw std::runtime_error("main: argc!=4");
4    try { // proper cleanup in case of error
5      list<float> l(std::stoi(argv[1]));
6      for (int i=0;i<l.length();i++)
7        l.set(i,cos(i));
8      l.to_dot("1.dot");
9      l.insert(std::stoi(argv[2]),42);
10     l.to_dot("2.dot");
11     l.erase(std::stoi(argv[3]));
12     l.to_dot("3.dot");
13   } catch (const std::runtime_error &e) {
14     std::cerr << "ERROR: "
15               << e.what() << std::endl;
16   }
17   return 0;
18 }
```

Bei Aufruf mit Kommandozeilenargumenten 4 3 2:

- Konstruktion einer Liste der Länge 4 und Initialisierung
- Einfügen des Elements 42 an Position 3 $\Rightarrow$ ggf. Ausnahme std::runtime_error (falls Länge $\leq 3$)
- Entfernen des Elements an Position 2 $\Rightarrow$ ggf. Ausnahme std::runtime_error (falls Länge $\leq 2$)
- Ausgabe der jeweiligen Zwischenzustände im `graphviz`-Format

- `./main.exe 4 3 2`




- `./main.exe 1 3 2`

  ERROR: insert: pos<0||pos>=length

- `./main.exe 2 1 4`

  ERROR: erase: pos<0||pos>=length

# Listen der Standardbibliothek

## vs. Vektoren der Standardbibliothek



Die Wahl des falschen Containers kann teilweise dramatische Auswirkungen auf die Effizienz des Codes haben, z.B. Entfernen / Einfügen an beliebigen Positionen:

```
1  #include <vector>
2
3  int main(int argc, char* argv[]) {
4    assert(argc==2); int n=std::stoi(argv[1]);
5    std::vector<double> v;
6    double a=1,b=2;
7
8    v.push_back(a);
9    for (int j=0;j<n;j++) v.push_back(b);
10   v.push_back(a);
11   for (int j=0;j<n;j++) {
12     auto i=v.begin(); v.erase(++i);
13   }
14   return 0;
15 }
```

- `diff vector.cpp list.cpp`

```
4c4
< #include <vector>
---
> #include <list>
9c9
<   std::vector<double> v;
---
>   std::list<double> v;
```

- `vector.exe 200000` benötigt 3.7s
- `list.exe 200000` benötigt 0.1s
- Analoges Verhalten beobachtet man für insert.

# Listen der Standardbibliothek

## vs. Vektoren der Standardbibliothek



Vektoren sind für den effizienten Zugriff auf beliebige Elemente (*random access*) deutlich besser geeignet.

```
1  #include <iostream>
2  #include <list>
3  #include <cassert>
4
5  int main(int argc, char* argv[]) {
6    assert(argc==2);
7    int n=std::stoi(argv[1]);
8    std::list<double> l(n);
9    int i=0; for (auto &e:l) e=++i;
10   for (int i=0;i<n;i++) {
11     auto it=l.begin();
12     for (int j=0;j<i;j++,it++);
13     std::cout << *it << " ";
14   }
15   std::cout << std::endl;
16   return 0;
17 }
```

- Zugriff auf das i-te Listenelement über Iterator benötigt lineare Suche von l.begin() (alternativ: l.rbegin()) ausgehend.
- `list.exe 200000` benötigt 43s
- Zugriff auf das i-te Vektorelement über v[i]
- `vector.exe 200000` benötigt 0.4s

# Listen der Standardbibliothek

## Algorithmen



```
1  template<typename T>
2  void list<T>::algorithms() {
3    std::cout << "count(42): " << count(data.begin(),data.end(),42) << std::endl;
4    std::cout << "min: " << *min_element(data.begin(),data.end()) << std::endl;
5    std::cout << "max: " << *max_element(data.begin(),data.end()) << std::endl;
6    std::cout << "before sorting: ";
7    for (int i=0;i<length();i++) std::cout << get(i) << " ";
8    std::cout << std::endl;
9    data.sort();
10   std::cout << "after sorting: ";
11   for (int i=0;i<length();i++) std::cout << get(i) << " ";
12   std::cout << std::endl;
13   std::cout << "permutations: " << std::endl;
14   int c=0;
15   do {
16     std::cout << c++ << ": ";
17     for (auto i=data.cbegin();i!=data.cend();i++)
18       std::cout << *i << " ";
19     std::cout << std::endl;
20   } while (next_permutation(data.begin(),data.end()));
21 }
```

# Listen der Standardbibliothek

## Algorithmen: Fallstudie



```
1  class A {
2    size_t i;
3  public:
4    A(size_t i) : i(i) {};
5    // required by min/max_element
6    bool operator<(const A& a) { return i<a.i; }
7    size_t id() const { return i; }
8  };
9
10 std::ostream& operator<<(std::ostream& o, const A& a) {
11   o << a.id(); return o;
12 }
13
14 int main() {
15   using namespace std;
16   A a(1),b(2),c(3); list<A> l;
17   l.push_back(a); l.push_back(b); l.push_back(c);
18   std::cout << "min: " << *min_element(l.begin(),l.end()) << std::endl;
19   std::cout << "max: " << *max_element(l.begin(),l.end()) << std::endl;
20   return 0;
21 }
```

# Zusammenfassung

## Vektoren

- Konstruktion / Destruktion
- Einfügen / Entfernen von Elementen
- Fallstudie

## Listen

- Konstruktion / Destruktion
- Einfügen / Entfernen von Elementen
- Fallstudie

## Listen der Standardbibliothek

- Konstruktion
- Einfügen / Entfernen von Elementen
- Elementzugriff
- Fallstudie
- Listen vs. Vektoren
- Algorithmen