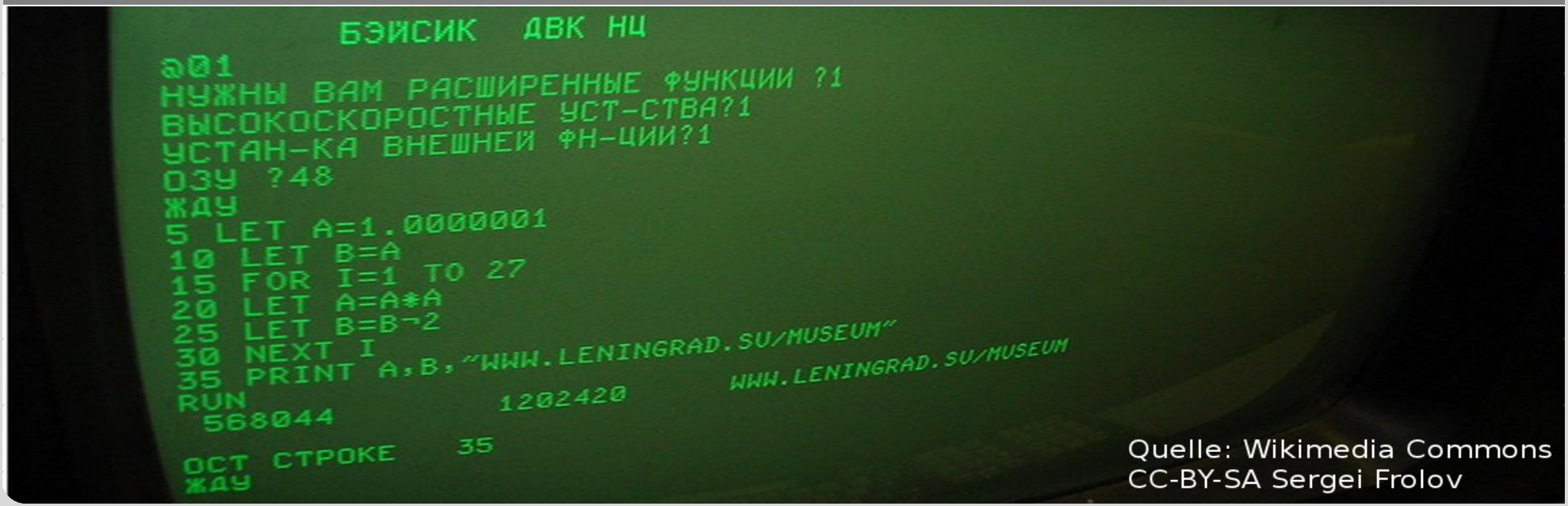
Karlsruher Institut für Technologie

C++ Workshop

08. Block, 22.06.2012

Markus Jung, Oliver Schneider, Robert Schneider | 22. Juni 2012



Gliederung

1. const
   - const pointer
   - const class
   - mutable
   - const cast
2. template basics
   - Über templates
   - Essentials zur Verwendung
   - Mehr zu templates
3. Komplexitätstheorie
4. STL-Datenstrukturen
   - Überblick
   - sequence containers
5. Praxis

1. const
   - const pointer
   - const class
   - mutable
   - const cast

const variablen

const für Konstanten

```
1  const int answer = 42;
2  answer = 99;              // wird nicht kompiliert
3  int j = answer+5;         // ist erlaubt
```

const pointer

const pointer

```
1  // Pointer zu einem konstanten MyClass Objekt
2  const MyClass * blub;
3  MyClass const * blub2;
4  const MyClass const * blub3;
5
6  // konstanter Pointer zu einem MyClass Objekt
7  MyClass * const blub4;
8
9  // konstanter Pointer zu einem konstanten MyClass Objekt
10 MyClass const * const blub5;
11 const MyClass * const blub6;
12 const MyClass const * const blub7;
```

const Funktions-Parameter

const als Veränderungsschutz

```
1  void drucker(int arg)
2  {
3      std::cout << arg << std::endl;
4  }
5
6  void testfun(int *arg)
7  {
8      *arg = 42;
9  }
10
11 void myfun(const int *arg)
12 {
13     *arg = 99;      // wird nicht kompiliert
14     testfun(arg);   // wird nicht kompiliert
15     drucker(*arg); // erlaubt
16 }
```

const Funktions-Parameter

const als Absicherung

```
1  void druckeRechnung(const Rechnung& rech)
2  {
3      std::cout << rech->menge << std::endl;
4      rech->menge++; // wird nicht kompiliert
5  }
6
7  void setzeRechnung(Rechnung& rech)
8  {
9      rech->menge = 100;
10 }
```

const member funktionen

const als Absicherung

```
1   class MyClass
2   {
3   private:
4       int myInt;
5
6   public:
7       int getInt() const
8       {
9           return this->myInt;
10      }
11
12      void bad() const
13      {
14          this->myInt = 99;
15      }
16
17      void setInt(int value)
18      {
19          if(value < 0) { this->myInt = 0; }
20          else          { this->myInt = value; }
21      }
22  };
```

const member funktionen

const als Veränderungsschutz

```
1  void drucker(const MyClass& arg)
2  {
3      std::cout << arg.getInt() << std::endl;
4      arg.setInt(99); // wird nicht kompiliert
5  }
6  void testfun(MyClass& arg)
7  {
8      std::cout << arg.getInt() << std::endl;
9      arg.setInt(99);
10 }
11 void myfun()
12 {
13     MyClass pObj;
14     pObj.setInt(42);
15     testfun(pObj);
16     drucker(pObj);
17 }
```

const overloading

const overloading

```
1  class MyClass
2  {
3  private:
4      int * m_iBlub;
5  public:
6      // sinnvoller overload
7      const int * get() const { return m_iBlub; }
8            int * get()       { return m_iBlub; }
9
10     // Rueckgabetyp kann frei gewaehlt werden
11     // Sinnhaftigkeit ist allerdings fragwuerdig
12     std::string seltsam()       { return "blubber"; }
13     double***   seltsam() const { return NULL; }
14 };
```

const aushebeln

mutable

```
1  class MyClass
2  {
3  private:
4      mutable unsigned int m_uCounter;
5  public:
6      unsigned int getCounter() const
7      {
8          return this->m_uCounter;
9      }
10
11     int compute(int value) const
12     {
13       m_uCounter++;
14       return value*42;
15     }
16 };
```

const aushebeln

mutable

```
1  MyClass test;
2  const MyClass& const_test = test;
3
4  std::cout << test.compute(1)       << std::endl;
5  std::cout << test.compute(-3)      << std::endl;
6  std::cout << const_test.compute(-5) << std::endl;
7  std::cout << test.getCounter()     << std::endl; // gibt 3 aus
```

const aushebeln

mutable

```
1  void bad_function(const int& const_ref)
2  {
3      const_ref++; // kompiliert nicht
4      int& ref = const_cast<int&>(const_ref);
5      ref++;       // kompiliert
6  }
7
8  int original = 5;
9  bad_function(original);
10 std::cout << original << std::endl; // gibt 6 aus
```

2. template basics
   - Über templates
   - Essentials zur Verwendung
   - Mehr zu templates

Was sind templates?

templates sind: Vorlagen (Schablonen) für Klassen und Funktionen

Was sind templates?

templates sind: Vorlagen (Schablonen) für Klassen und Funktionen

```
1     int max(int p0, int p1) { return (p0>p1) ? (p0) : (p1); }
2   short max(short, short)   { return (p0>p1) ? (p0) : (p1); }
3  double max(double, double) { return (p0>p1) ? (p0) : (p1); }
4  MyType max(MyType, MyType) { return (p0>p1) ? (p0) : (p1); }
```

Was sind templates?

templates sind: Vorlagen (Schablonen) für Klassen und Funktionen

```
1     int max(int p0, int p1) { return (p0>p1) ? (p0) : (p1); }
2   short max(short, short)   { return (p0>p1) ? (p0) : (p1); }
3  double max(double, double) { return (p0>p1) ? (p0) : (p1); }
4  MyType max(MyType, MyType) { return (p0>p1) ? (p0) : (p1); }
```

```
1  class MyRingbuffer_int { int* p; /*...*/ };
2  class MyRingbuffer_short;
3  class MyRingbuffer_double;
4  class MyRingbuffer_MyType;
```

Zum Einstellen

templates sind

- mächtiger als der Präprozessor
- schwieriger als der Rest von C++ zusammen

Zum Einstellen

templates sind

- mächtiger als der Präprozessor
- schwieriger als der Rest von C++ zusammen

Beides nicht ganz wahr!

- man kann ganz wenige Dinge nur mit dem Präprozessor tun (und viele Dinge nur mit templates)
- die grundlegende Verwendung von templates ist einfach

Für den Hinterkopf

Wichtig bei templates ist:

- templates erzeugen Klassen und Funktionen
- aber keinen echten Quelltext
- templates werden unmittelbar vor der Übersetzung verarbeitet
- Fokus: statische Typsicherheit und Umgang mit Typen

Funktions-templates: Beispiel

Ziel: Anhand einer Vorlage all diese Funktionen erzeugen *lassen*.
D.h.: Vorlage + Typ $\Longrightarrow$ Funktion

```
1     int max(int p0, int p1) { return (p0>p1) ? (p0) : (p1); }
2   short max(short, short)   { return (p0>p1) ? (p0) : (p1); }
3  double max(double, double) { return (p0>p1) ? (p0) : (p1); }
4  MyType max(MyType, MyType) { return (p0>p1) ? (p0) : (p1); }
```

Funktions-templates: Beispiel

Ziel: Anhand einer Vorlage all diese Funktionen erzeugen *lassen*.
D.h.: Vorlage + Typ $\Longrightarrow$ Funktion

```
1     int max(int p0, int p1) { return (p0>p1) ? (p0) : (p1); }
2   short max(short, short)   { return (p0>p1) ? (p0) : (p1); }
3  double max(double, double) { return (p0>p1) ? (p0) : (p1); }
4  MyType max(MyType, MyType) { return (p0>p1) ? (p0) : (p1); }
5
6
7     Type max(Type p0, Type p1) {
8       return (p0>p1) ? (p0) : (p1);
9     }
```

Funktions-templates: Beispiel

Ziel: Anhand einer Vorlage all diese Funktionen erzeugen *lassen*.
D.h.: Vorlage + Typ $\Longrightarrow$ Funktion

```
1     int max(int p0, int p1) { return (p0>p1) ? (p0) : (p1); }
2   short max(short, short)   { return (p0>p1) ? (p0) : (p1); }
3  double max(double, double) { return (p0>p1) ? (p0) : (p1); }
4  MyType max(MyType, MyType) { return (p0>p1) ? (p0) : (p1); }
5
6  template < typename Type >
7    Type max(Type p0, Type p1) {
8      return (p0>p1) ? (p0) : (p1);
9    }
```

Funktions-templates: Syntax

Definition eines Funktions-templates

```
0  template < typename T >     // template mit Typ-Parameter T
1  T                           // return type
2  max(T p0, T p1)             // Funktionsname und Parameter
3  { return (p0>p1) ? (p0) : (p1); }  // Definition (Koerper)
```

Ein template liefert dem Compiler eine Konstruktionsvorschrift:
Vorlage + Parameter $\Longrightarrow$ Funktion

Funktions-templates: Syntax

Definition eines Funktions-templates

```
0  template < typename T >     // template mit Typ-Parameter T
1  T                           // return type
2  max(T p0, T p1)             // Funktionsname und Parameter
3  { return (p0>p1) ? (p0) : (p1); }  // Definition (Koerper)
```

Ein template liefert dem Compiler eine Konstruktionsvorschrift:
Vorlage + Parameter $\Longrightarrow$ Funktion

Veranschaulichung: int als Parameter T

```
3  int
4  max(int p0, int p1)
5  { return (p0>p1) ? (p0) : (p1); }
```

Verwendung von Funktions-templates

Ein Funktions-template ist eine Vorlage, und keine Funktion! (selbes gilt für class templates)

$\Longrightarrow$ es bedarf des Konstruktionsschrittes *function template* $\rightarrow$ *function*

Nennt sich: Instanziierung (des Funktions-templates)

Warum nicht gleich für alle möglichen Parameter instanziieren?

Verwendung von Funktions-templates

Ein Funktions-template ist eine Vorlage, und keine Funktion! (selbes gilt für class templates)

$\Longrightarrow$ es bedarf des Konstruktionsschrittes *function template* $\rightarrow$ *function*

Nennt sich: Instanziierung (des Funktions-templates)

Warum nicht gleich für alle möglichen Parameter instanziieren?

code bloat!

Verwendung von Funktions-templates

Ein Funktions-template ist eine Vorlage, und keine Funktion! (selbes gilt für class templates)
$\Longrightarrow$ es bedarf des Konstruktionsschrittes *function template* $\rightarrow$ *function*
Nennt sich: Instanziierung (des Funktions-templates)

Warum nicht gleich für alle möglichen Parameter instanziieren?
code bloat!

Also nur Instanziieren, wenn tatsächlich benötigt. Funktioniert wie?

Verwendung von Funktions-templates

Ein Funktions-template ist eine Vorlage, und keine Funktion! (selbes gilt für class templates)
$\Longrightarrow$ es bedarf des Konstruktionsschrittes *function template* $\rightarrow$ *function*
Nennt sich: Instanziierung (des Funktions-templates)

Warum nicht gleich für alle möglichen Parameter instanziieren?
code bloat!

Also nur Instanziieren, wenn tatsächlich benötigt. Funktioniert wie?

Geschieht automagisch bei der Verwendung! Hier liegt der Unterschied zu macros usw.!

Implizite Instanziierung

Aufrufen einer template-Funktion

```
0  int c = max<int>(4, 42);
```

Implizite Instanziierung

Aufrufen einer template-Funktion

```
0  int c = max < int > (4, 42);
```

- Compiler erkennt: hier wird eine Schablone benannt und eine Instanz benötigt.
- $\Longrightarrow$ Compiler instanziiert das template, Funktionsaufruf wird mit Instanz verbunden.
- Das fertige Programm wird an dieser Stelle die entstandene Funktion aufrufen.

Implizite Instanziierung

Aufrufen einer template-Funktion

```
0  int c = max < int > (4, 42);
```

- Compiler erkennt: hier wird eine Schablone benannt und eine Instanz benötigt.
- $\Longrightarrow$ Compiler instanziiert das template, Funktionsaufruf wird mit Instanz verbunden.
- Das fertige Programm wird an dieser Stelle die entstandene Funktion aufrufen.

```
1  double c = max < double > (42.0, 21.0);
```

Eine andere Funktion wird erzeugt!
Es wird nur eine Instanz je Satz von template-Parametern erzeugt pro *translation unit* und nicht für jeden Aufruf eine.

Sichtbarkeit und Instanziierung

Bisher:

- kann eine Funktion aufrufen, wenn die Deklaration vorhanden ist
- die Funktion kann (external linkage) in einer anderen translation unit definiert sein

Sichtbarkeit und Instanziierung

Bisher:

- kann eine Funktion aufrufen, wenn die Deklaration vorhanden ist
- die Funktion kann (external linkage) in einer anderen translation unit definiert sein

templates:

- Instanziierung erfordert vorherige *Definition*!
- $\Longrightarrow$ Funktionsaufrufe von noch nicht instanziierten templates erfordern die Definition (den Körper) der Funktion weiter oben in *derselben* translation unit

Sichtbarkeit und Instanziierung

Bisher:

- kann eine Funktion aufrufen, wenn die Deklaration vorhanden ist
- die Funktion kann (external linkage) in einer anderen translation unit definiert sein

templates:

- Instanziierung erfordert vorherige *Definition*!
- $\Longrightarrow$ Funktionsaufrufe von noch nicht instanziierten templates erfordern die Definition (den Körper) der Funktion weiter oben in *derselben* translation unit

common solution templates inkl. Definitionen in die Header!

Problem One definition rule (darf nicht mehr als eine Definition im gesamten Programm haben)

Umgehung internal linkage (`static`) oder `inline`

templates & headers

header.h

```
1  template < typename T >
2  inline T max(T p0, T p1)
3  {
4      return (p0>p1) ? p0 : p1;
5  }
6
7  template < typename Type >
8  static void print(Type p)
9  {
10     std::cout << p;
11 }
12
13
14 void foobar();
```

templates & headers

header.h

```
1  template < typename T >
2  inline T max(T p0, T p1)
3  {
4      return (p0>p1) ? p0 : p1;
5  }
6
7  template < typename Type >
8  static void print(Type p)
9  {
10     std::cout << p;
11 }
12
13
14 void foobar();
```

foobar.cpp

```
0  #include "header.h"
1  void foobar()
2  {
3      print < int > (42);
4  }
```

templates & headers

header.h

```
1  template < typename T >
2  inline T max(T p0, T p1)
3  {
4      return (p0>p1) ? p0 : p1;
5  }
6
7  template < typename Type >
8  static void print(Type p)
9  {
10     std::cout << p;
11 }
12
13
14 void foobar();
```

foobar.cpp

```
0  #include "header.h"
1  void foobar()
2  {
3      print < int > (42);
4  }
```

main.cpp

```
0  #include "header.h"
1  int main()
2  {
3      int x, y;
4      std::cin >> x >> y;
5
6      int m = max < int > (x, y);
7      print(m);
8      foobar();
9  }
```

template parameters

Die template-declaration enthält wie Funktions-Deklarationen eine Menge von Parametern:

```
1  void foo(int, double, MyType, int);
2
3  template < typename T, typename MyTemplateParam,
4             class x >
5  void bar();
```

Im Kontext einer template-declaration sind `typename` und `class` semantisch identisch.

template parameters

Die template-declaration enthält wie Funktions-Deklarationen eine Menge von Parametern:

```
1  void foo(int, double, MyType, int);
2
3  template < typename T, typename MyTemplateParam,
4             class x >
5  void bar();
```

Im Kontext einer template-declaration sind `typename` und `class` semantisch identisch.
Ein template-Parameter empfängt bei der Instanziierung entweder

- einen Typen, d.h. den Namen einer `class`, `struct`, `union` $\Rightarrow$ template type-parameter
- oder einen Wert ähnlich wie bei einem Funktionsaufruf $\Rightarrow$ template non-type-parameter (später!)

class templates

Grundsätzlich: Wie function templates

```
1  template < typename T >
2  class MyRingbuffer {
3      T* data_member;
4      T memfun(int);
5      /* work with T */
6  };
```

```
1
2  class MyRingbuffer_double {
3      double* data_member;
4      double memfun(int);
5      /* work with double */
6  };
```

class templates

Grundsätzlich: Wie function templates

```
1  template < typename T >
2  class MyRingbuffer {
3      T* data_member;
4      T memfun(int);
5      /* work with T */
6  };
```

```
1
2  class MyRingbuffer_double {
3      double* data_member;
4      double memfun(int);
5      /* work with double */
6  };
```

Gleiche Probleme:

- Implizite Instanziierung bei Verwendung (z.B. Deklaration eines ≫Dinges≪)
- Definition in den Header
- linkage der member functions??

class template member functions

```
1  template < typename T >
2  class MyRingbuffer {
3      inline T* memfun(int);
4
5  };
```

class template member functions

```
1  template < typename T >
2  class MyRingbuffer {
3      inline T* memfun(int);
4
5  };
6
7  namespace /* anonymous */ {
8      template < typename T >
9      class MyRingbuffer {
10         T* memfun(int);
11     };
12 }
```

class template member functions

```
1  template < typename T >
2  class MyRingbuffer {
3      inline T* memfun(int);
4
5  };
6
7  namespace /* anonymous */ {
8      template < typename T >
9      class MyRingbuffer {
10         T* memfun(int);
11     };
12 }
13
14
15 F*
16
17 memfun(int)
18 { /* ... */ }
```

class template member functions

```
1  template < typename T >
2  class MyRingbuffer {
3      inline T* memfun(int);
4
5  };
6
7  namespace /* anonymous */ {
8      template < typename T >
9      class MyRingbuffer {
10         T* memfun(int);
11     };
12 }
13
14 template < typename F >
15 F*
16 MyRingbuffer < F > ::
17 memfun(int)
18 { /* ... */ }
```

class template member functions

```
1  template < typename T >
2  class MyRingbuffer {
3      inline T* memfun(int);
4
5  };
6
7  namespace /* anonymous */ {
8      template < typename T >
9      class MyRingbuffer {
10         T* memfun(int);
11     };
12 }
13
14 template < typename F >
15 F*
16 MyRingbuffer < F > ::
17 memfun(int)
18 { /* ... */ }
```

```
22 template < typename T >
23 class MyRingbuffer {
24     /* inline */ T* memfun(int)
25     { /* ... */ }
26 };
```

class template member function templates

```
30  template < typename T >
31  class MyRingbuffer
32  {
33      template < typename F >
34      inline F* memfun2(T);
35  };
```

class template member function templates

```
30  template < typename T >
31  class MyRingbuffer
32  {
33      template < typename F >
34      inline F* memfun2(T);
35  };
36
37
38  template < typename T > // class template-parameters
39  template < typename F > // function template-parameters
40  MyRingbuffer < T > ::
41  F*
42  memfun2(T p)
43  { /* ... */ }
```

intricacies: qualified-id, template

unqualified name lookup = weird

Generell eine gute Idee: alles qualifiziert (explizit) hinschreiben, z.B.

```
mynamespace::MyClass::MyStuff
```

Bei templates ist dies extrem wichtig!

intricacies: qualified-id, template

unqualified name lookup = weird

Generell eine gute Idee: alles qualifiziert (explizit) hinschreiben, z.B.
`mynamespace::MyClass::MyStuff`
Bei templates ist dies extrem wichtig!

keyword `template` (Standard, 14.2/4)

Folgt auf ein `.` `->` oder `::` der Name eines templates, so muss vor diesem Namen das Keyword `template` eingefügt werden (14.2/4).

```
1  struct X {
2      template < typename T >
3      void foo(T);
4  };
5  namespace N {
6      template < typename F >
7      class A { /* ... */ };
8  };
```

intricacies: qualified-id, template

unqualified name lookup = weird

Generell eine gute Idee: alles qualifiziert (explizit) hinschreiben, z.B.
`mynamespace::MyClass::MyStuff`
Bei templates ist dies extrem wichtig!

keyword `template` (Standard, 14.2/4)

Folgt auf ein `.` `->` oder `::` der Name eines templates, so muss vor diesem Namen das Keyword `template` eingefügt werden (14.2/4).

```
1  struct X {
2      template < typename T >
3      void foo(T);
4  };
5  namespace N {
6      template < typename F >
7      class A { /* ... */ };
8  };
```

```
11 X x;
12 x.foo < int > (42); // Fehler!
13
14
15 N::A < double > n;  // Fehler!
16
```

intricacies: qualified-id, template

unqualified name lookup = weird

Generell eine gute Idee: alles qualifiziert (explizit) hinschreiben, z.B.
`mynamespace::MyClass::MyStuff`
Bei templates ist dies extrem wichtig!

keyword `template` (Standard, 14.2/4)

Folgt auf ein `.` `->` oder `::` der Name eines templates, so muss vor diesem Namen das Keyword `template` eingefügt werden (14.2/4).

```
1  struct X {
2      template < typename T >
3      void foo(T);
4  };
5  namespace N {
6      template < typename F >
7      class A { /* ... */ };
8  };
```

```
11 X x;
12 x.foo < int > (42); // Fehler!
13 x.template foo < int > (42);
14
15 N::A < double > n;   // Fehler!
16 N::template A < double > n;
```

intricacies: dependent names, typename

Nur innerhalb von template-Definitionen!

keyword `typename` (Standard, 14.6/2-3)

Wird ein Typ mittels *qualified-id* benannt (mit `::`), und es taucht ein template Parameter (auch implizit) auf, so muss vor den ganzen Ausdruck ein `typename` gestellt werden.

intricacies: dependent names, typename

Nur innerhalb von template-Definitionen!

keyword `typename` (Standard, 14.6/2-3)

Wird ein Typ mittels *qualified-id* benannt (mit `::`), und es taucht ein template Parameter (auch implizit) auf, so muss vor den ganzen Ausdruck ein `typename` gestellt werden.

```
21  template < typename T >
22  class X {
23      typedef T* P;
24      void foobar() {
25          T a;                    // OK
26          T::A* x;                // multipliziere?
27          typename T::A y;        // Definition
28          typename X<T>::P p;     // Definition
29      }
30  };
```

3 Komplexitätstheorie

Vergleich von Algorithmen

Problem

Nach welchen Kriterien vergleicht man Algorithmen?

Vergleich von Algorithmen

Problem

Nach welchen Kriterien vergleicht man Algorithmen?

- Rechenzeit
- Speicherbedarf
- I/O-Bandbreitenbedarf
- (Parallelisierbarkeit)

Vergleich von Algorithmen

Problem

Nach welchen Kriterien vergleicht man Algorithmen?

- Rechenzeit
- Speicherbedarf
- I/O-Bandbreitenbedarf
- (Parallelisierbarkeit)

Das sind alles von der ausführenden Maschine abhängige Metriken!
Ziel: Ein davon unabhängiger Bewertungsmaßstab

(Mit dem ferner Beweise und theoretische Analysen möglich sind)

Big O: Das algorithmische Komplexitätsmaß

O-Kalkül (Landau-Symbole)

- Beschreibt das asymptotische Verhalten
- Reduktion auf das Wesentliche
- Vernachlässigung konstanter Faktoren (!) und asymptotisch unbedeutender Terme
  - Manchmal sind konstante Faktoren aber **nicht** zu vernachlässigen!

Big O: Das algorithmische Komplexitätsmaß

O-Kalkül (Landau-Symbole)

- Beschreibt das asymptotische Verhalten
- Reduktion auf das Wesentliche
- Vernachlässigung konstanter Faktoren (!) und asymptotisch unbedeutender Terme
  - Manchmal sind konstante Faktoren aber **nicht** zu vernachlässigen!
- $f \in \mathcal{O}(g)$: f wächst asymptotisch höchstens so schnell wie g
- $f \in \Theta(g)$: f wächst asymptotisch genau so schnell wie g
- $f \in \Omega(g)$: f wächst asymptotisch mindestens so schnell wie g

O-Kalkül: Anwendung und Beispiele

Notation

- Landau-Symbole beschreiben eigentlich Mengen von Funktionen
- „Korrekte“ Element-Notation: $f \in \mathcal{O}(g)$
- Verbreitete Notation: $f = \mathcal{O}(g)$

O-Kalkül: Anwendung und Beispiele

Notation

- Landau-Symbole beschreiben eigentlich Mengen von Funktionen
- „Korrekte“ Element-Notation: $f \in \mathcal{O}(g)$
- Verbreitete Notation: $f = \mathcal{O}(g)$

Meist Angabe der Zeitkomplexität in Abhängigkeit der Eingabegröße, aber auch für andere (kritischen) Ressourcen (Speicher, Bandbreite etc.) einsetzbar. Beispiele (Eingabegröße: n):

- Arrayzugriffe: $\Theta(1)$
- Binäre Suche: $\mathcal{O}(log(n))$
- Iterieren durch eine Liste: $\Theta(n)$
- Sortieren (vergleichsbasiert): $\Omega(n * log(n))$

4 STL-Datenstrukturen

- Überblick
- sequence containers

Standard Template Library

Aufbau: building blocks

- containers, adaptors
- streams
- algorithms
- functional programming
- helpers (e.g. `reference_wrapper`)
- metaprogramming support

Standard Template Library

Alexander Stepanov, Mitbegründer des generic programming und der STL: aus: Forward to „STL Tutorial and Reference Guide, Second Edition“

Fundamental ideas behind STL

- Generic programming
- Abstractness without loss of efficiency
- Von Neumann computational model (pointers)
- Value semantics (whole-part ownership)

Standard Template Library

Alexander Stepanov, Mitbegründer des generic programming und der STL: aus: Forward to „STL Tutorial and Reference Guide, Second Edition“

Fundamental ideas behind STL

- Generic programming
- Abstractness without loss of efficiency
- Von Neumann computational model (pointers)
- Value semantics (whole-part ownership)

„Fun fact“ *;)*
STL $\subset$ Standard-Bibliothek

Überblick

Die STL stellt für die geordnete Speicherung von „Dingen“ drei Containertypen bereit:

- `vector`
- `list`
- `deque`

Überblick

Die STL stellt für die geordnete Speicherung von „Dingen“ drei Containertypen bereit:

- `vector`
- `list`
- `deque`

Außerdem gibt es noch drei Adapter-Typen, die auf Basis dieser Container arbeiten:

- `queue`
- `priority_queue`
- `stack`

Überblick

Die STL stellt für die geordnete Speicherung von „Dingen“ drei Containertypen bereit:

- `vector`
- `list`
- `deque`

Außerdem gibt es noch drei Adapter-Typen, die auf Basis dieser Container arbeiten:

- `queue`
- `priority_queue`
- `stack`

Diese werden in einem der nächsten Workshops besprochen.

Gemeinsame Eigenschaften/Funktionalität

- Speichern Sequenzen von „Dinge“
  - In der vorgegebenen Reihenfolge (Überraschung!)

Gemeinsame Eigenschaften/Funktionalität

- Speichern Sequenzen von „Dinge“
  - In der vorgegebenen Reihenfolge (Überraschung!)
  - Dynamisch, sie passen ihre Größe den Anforderungen an

Gemeinsame Eigenschaften/Funktionalität

- Speichern Sequenzen von „Dinge“
  - In der vorgegebenen Reihenfolge (Überraschung!)
  - Dynamisch, sie passen ihre Größe den Anforderungen an

Funktionsumfang:

- Zugriff über Iteratoren (mehr dazu später)
  - Sowohl vorwärts als auch rückwärts
  - Generell je Inkrement in $\mathcal{O}(1)$

Gemeinsame Eigenschaften/Funktionalität

- Speichern Sequenzen von „Dinge“
  - In der vorgegebenen Reihenfolge (Überraschung!)
  - Dynamisch, sie passen ihre Größe den Anforderungen an

Funktionsumfang:

- Zugriff über Iteratoren (mehr dazu später)
  - Sowohl vorwärts als auch rückwärts
  - Generell je Inkrement in $\mathcal{O}(1)$
- Elementzählung in $\mathcal{O}(1)$

Gemeinsame Eigenschaften/Funktionalität

- Speichern Sequenzen von „Dinge“
  - In der vorgegebenen Reihenfolge (Überraschung!)
  - Dynamisch, sie passen ihre Größe den Anforderungen an

Funktionsumfang:

- Zugriff über Iteratoren (mehr dazu später)
  - Sowohl vorwärts als auch rückwärts
  - Generell je Inkrement in $\mathcal{O}(1)$
- Elementzählung in $\mathcal{O}(1)$
- Elementzugriff
  - Anfang und Ende in $\mathcal{O}(1)$
  - `vector` und `deque` auch Wahlfrei (in $\mathcal{O}(1)$)

Gemeinsame Eigenschaften/Funktionalität

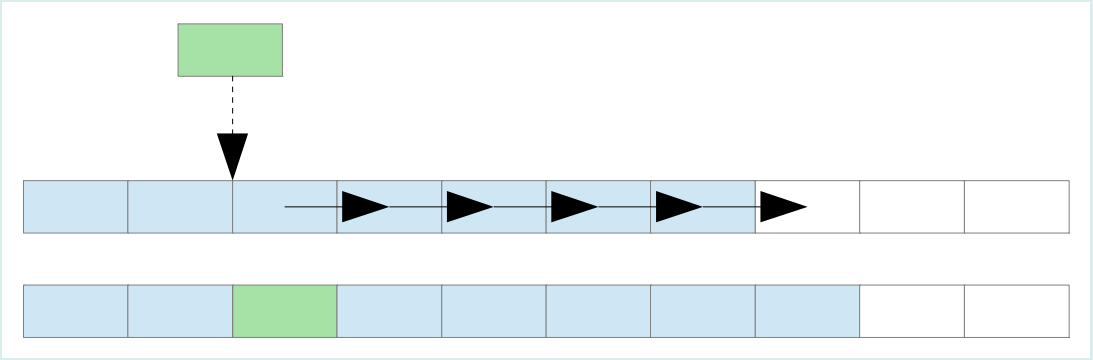
- Speichern Sequenzen von „Dinge“
  - In der vorgegebenen Reihenfolge (Überraschung!)
  - Dynamisch, sie passen ihre Größe den Anforderungen an

Funktionsumfang:

- Zugriff über Iteratoren (mehr dazu später)
  - Sowohl vorwärts als auch rückwärts
  - Generell je Inkrement in $\mathcal{O}(1)$
- Elementzählung in $\mathcal{O}(1)$
- Elementzugriff
  - Anfang und Ende in $\mathcal{O}(1)$
  - `vector` und `deque` auch Wahlfrei (in $\mathcal{O}(1)$)
- Modifikation: Einfügen, löschen und leeren

Container: `vector`

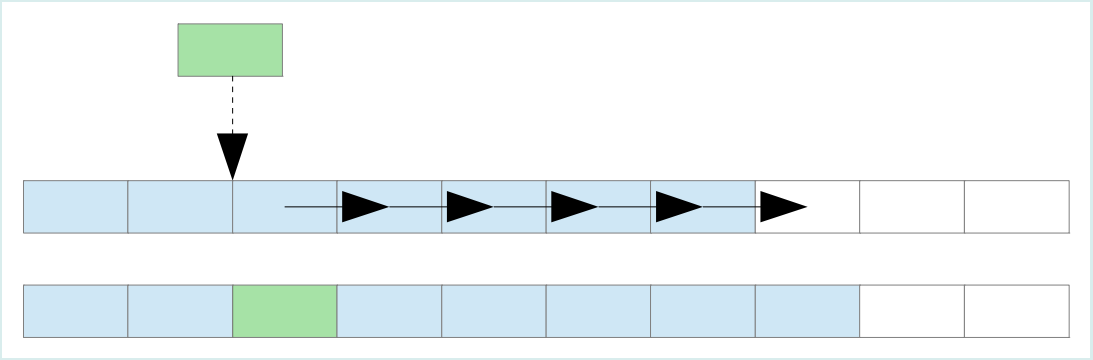
Abbildung: Struktur



- Array-basiert

Container: `vector`

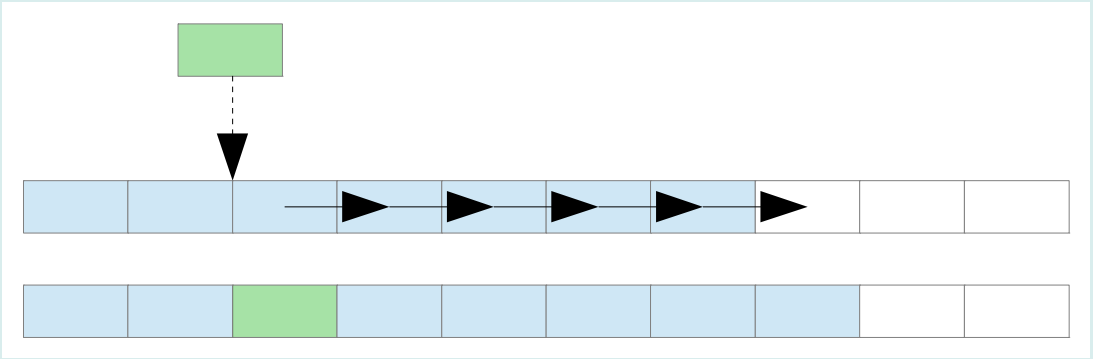
Abbildung: Struktur



- Array-basiert
  - \+ Cache-effizient
  - \+ (Direkt-)Zugriffe sehr schnell möglich: $\mathcal{O}(1)$

Container: `vector`

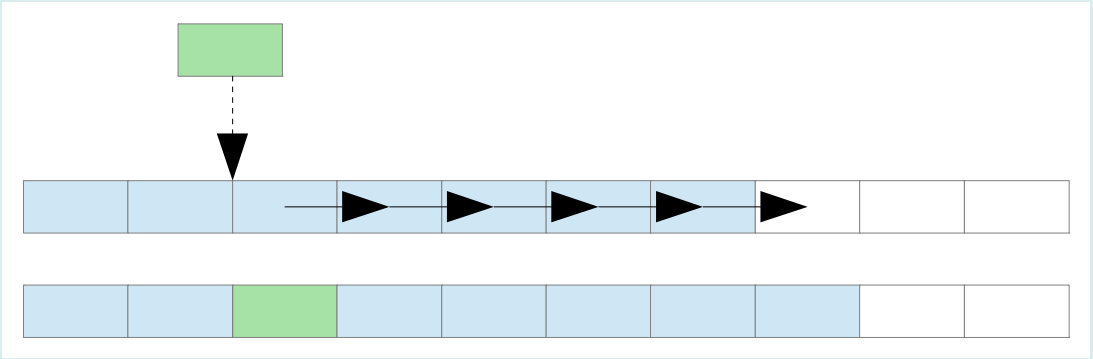
Abbildung: Struktur



- Array-basiert
  - \+ Cache-effizient
  - \+ (Direkt-)Zugriffe sehr schnell möglich: $\mathcal{O}(1)$
  - Wachstum erfordert Kopiervorgänge!
  - \- Einfügen/Entfernen ist teuer: $\mathcal{O}(n)$ (!)
    - Ausnahme: Am Ende des `vectors` (mit `push_back`, `pop_back`): $\mathcal{O}(1)$

Container: `vector`

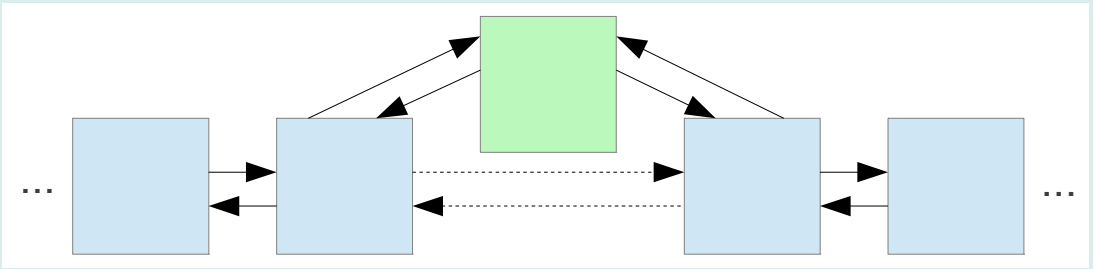
Abbildung: Struktur



- Array-basiert
  - \+ Cache-effizient
  - \+ (Direkt-)Zugriffe sehr schnell möglich: $\mathcal{O}(1)$
  - Wachstum erfordert Kopiervorgänge!
  - \- Einfügen/Entfernen ist teuer: $\mathcal{O}(n)$ (!)
    - Ausnahme: Am Ende des `vectors` (mit `push_back`, `pop_back`): $\mathcal{O}(1)$
- Vor allem geeignet für Aufgaben mit wahlfreiem Zugriff und wenigen Änderungsoperationen die nicht am Ende stattfinden.

Container: `list`

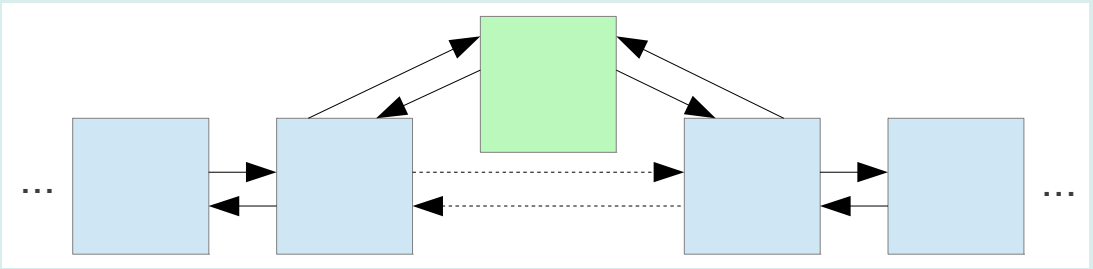
Abbildung: Struktur



- Zugrundeliegende Datenstruktur: Doppelt verkettete Liste

Container: `list`

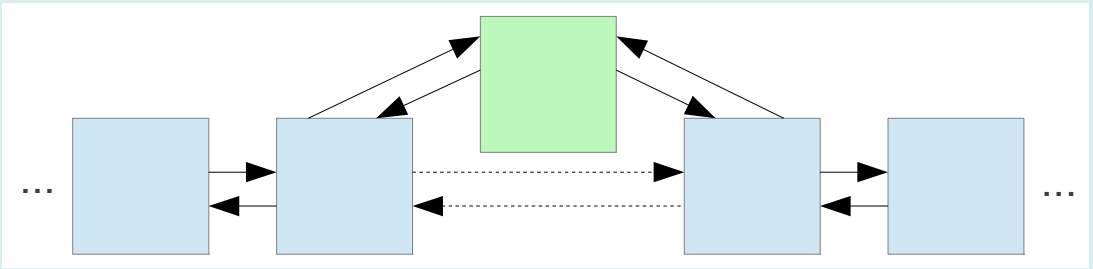
Abbildung: Struktur



- Zugrundeliegende Datenstruktur: Doppelt verkettete Liste
  - Schlechte Cache-Effizienz (nichtkontinuierliche Speicherbelegung)
  - Kein Direktzugriff! Über Iteration in $\mathcal{O}(n)$ (!)

Container: `list`

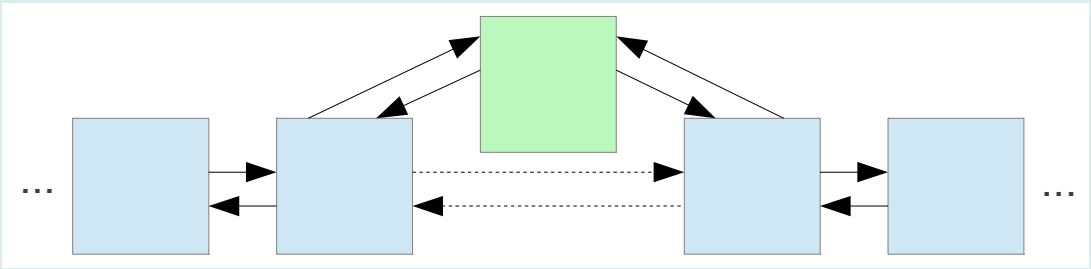
Abbildung: Struktur



- Zugrundeliegende Datenstruktur: Doppelt verkettete Liste
  - \- Schlechte Cache-Effizienz (nichtkontinuierliche Speicherbelegung)
  - \- Kein Direktzugriff! Über Iteration in $\mathcal{O}(n)$ (!)
  - \+ Einfügen/Entfernen ist (bei gegebener Position) sehr schnell: $\mathcal{O}(1)$
  - \+ Verschieben von Elementen in eine andere `list` ist (in zwei Fällen) sehr schnell: `splice` in $\mathcal{O}(1)$

Container: `list`

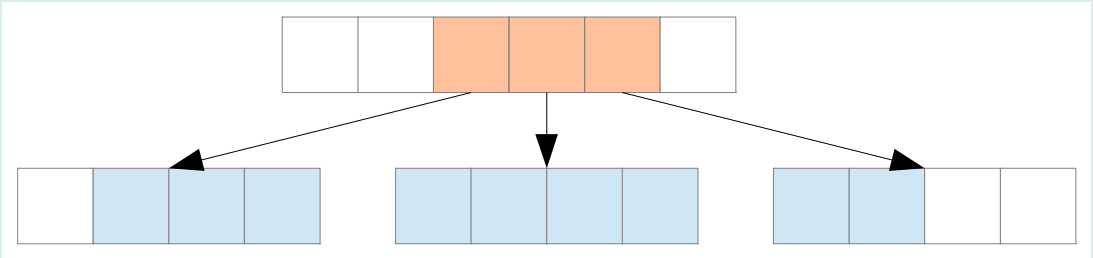
Abbildung: Struktur



- Zugrundeliegende Datenstruktur: Doppelt verkettete Liste
  - \- Schlechte Cache-Effizienz (nichtkontinuierliche Speicherbelegung)
  - \- Kein Direktzugriff! Über Iteration in $\mathcal{O}(n)$ (!)
  - \+ Einfügen/Entfernen ist (bei gegebener Position) sehr schnell: $\mathcal{O}(1)$
  - \+ Verschieben von Elementen in eine andere `list` ist (in zwei Fällen) sehr schnell: `splice` in $\mathcal{O}(1)$
- Vor allem geeignet für Algorithmen mit einer großen Anzahl nichttrivialer Listenmodifikationen, solange kein wahlfreier Zugriff erforderlich ist.

Container: `deque`

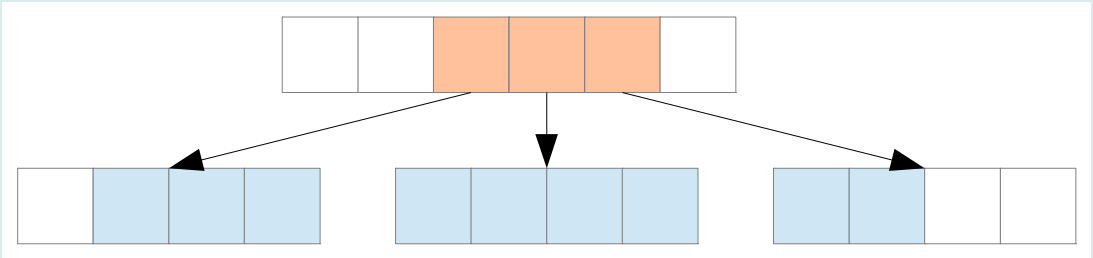
Abbildung: Struktur



- Datenstruktur: Array-basiert ähnlich wie `vector`, aber in kleine Blöcke unterteilt

Container: `deque`

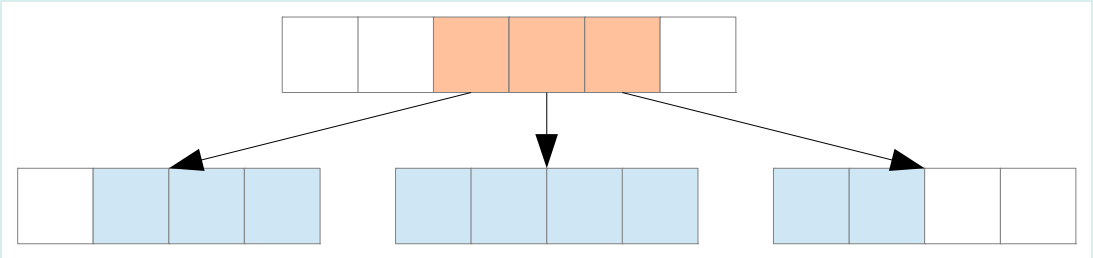
Abbildung: Struktur



- Datenstruktur: Array-basiert ähnlich wie `vector`, aber in kleine Blöcke unterteilt
  - \+ Cache-Effizienz: Zusammenhängende Blöcke $\rightarrow$ vermutlich gut
  - \+ Direktzugriff: $\mathcal{O}(1)$

Container: `deque`

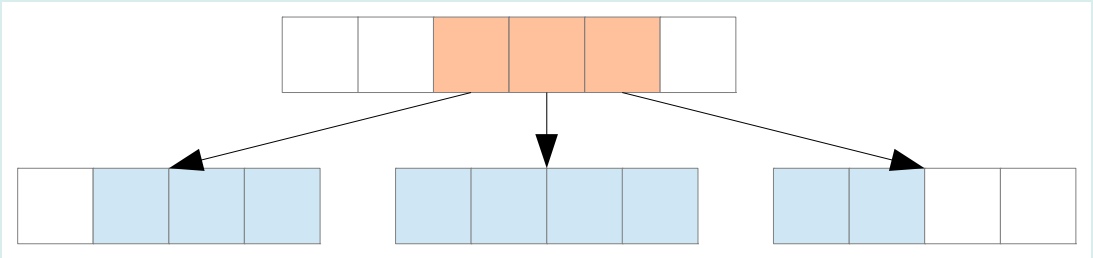
Abbildung: Struktur



- Datenstruktur: Array-basiert ähnlich wie `vector`, aber in kleine Blöcke unterteilt
  - \+ Cache-Effizienz: Zusammenhängende Blöcke $\rightarrow$ vermutlich gut
  - \+ Direktzugriff: $\mathcal{O}(1)$
  - Wieder: Wachstum erfordert Kopieroperationen
  - \- Beliebiges Einfügen/Entfernen: $\mathcal{O}(n)$ (wie bei `vector`)
    - Ausnahme: An Anfang oder Ende der `deque` (`push/pop_front/back`): $\mathcal{O}(1)$

Container: `deque`

Abbildung: Struktur



- Datenstruktur: Array-basiert ähnlich wie `vector`, aber in kleine Blöcke unterteilt
  - \+ Cache-Effizienz: Zusammenhängende Blöcke $\rightarrow$ vermutlich gut
  - \+ Direktzugriff: $\mathcal{O}(1)$
  - Wieder: Wachstum erfordert Kopieroperationen
  - \- Beliebiges Einfügen/Entfernen: $\mathcal{O}(n)$ (wie bei `vector`)
    - Ausnahme: An Anfang oder Ende der `deque` (`push/pop_front/back`): $\mathcal{O}(1)$
- `deque` ist etwas flexibler als `vector`, hat aber auch einen größeren Verwaltungsoverhead. (Konstante Faktoren!)

Codebeispiel

```
1  #include <list>
2  #include <vector>
3
4  std::vector<int> v = {1, 2, 3};              // initialization-list, C++11
5  std::list  <int> l = {1, 2, 3};              // initialization-list, C++11
6
7  v.push_back(42);       l.push_back (42);     // Element anhaengen
8  /*     n/a     */      l.push_front(21);     // Element voranstellen
9
10 v.pop_back();          l.pop_back ();        // von hinten entfernen
11 /*     n/a     */      l.pop_front();        // von vorne entfernen
12
13
14 int f;
15 f = v.front();         f = l.front();        // erstes Element erhalten
16 f = v.back();          f = l.back();         // letztes Element erhalten
17
18 v[2] = 42;            /*     n/a     */     // Element 1 setzen
19 f    = v[2];          /*     n/a     */     // Element 1 erhalten
20
21 // list: mit Iteratoren!
```

Iteratoren: nächster Workshop!

Ausblick:

- „Verallgemeinerung“ von Pointern
- Container-unabhängiger Zugriff auf Elemente
- Iterator $\Longrightarrow$ Zugriff auf benachbartes Element

Wichtig für Algorithmen und essentiell für `list`!

Hilfe zur Selbsthilfe

Der C++-Standard ist die offizielle Referenz.
Die unten verlinkte Dokumentation ist in der Praxis aber

- mehr als ausreichend
- außerdem übersichtlich strukturiert
- bietet umfangreiche Erläuterungen
- und jede Menge Beispiele

`http://www.cplusplus.com/reference/stl/`

Etwas formaler (und mehr C++11):

`http://en.cppreference.com/w/cpp/container`

5 Praxis

Praxis!

- Aufgabe 1: Ringpuffer (const)
- Aufgabe 2: Eine generische Look-Up-Table
- Daueraufgabe: Schach

`https://github.com/kit-cpp-workshop/workshop-ss12-08`

Aufgabenbeschreibungen und Hinweise: Siehe `README.md`