Informatik II

Oliver Jack

Fachhochschule Jena
Fachbereich Elektrotechnik und Informationstechnik

Sommersemester 2010

Inhalt

Die C++-Standard Template Library (STL)

- Motivation
- Vektoren
- Container-Klassen
- Basis-Sequenz-Operationen
- Der vector-Container

Vorlesung 12. Die C++-Standard Template Library (STL)

Die C++-Standard Template Library (STL)

- Motivation
- Vektoren
- Container-Klassen
- Basis-Sequenz-Operationen
- Der vector-Container

Die C++-Standard-Bibliothek

- Die C++-Standard-Bibliothek ist ein fundamentaler Teil des C++-Standards.
- Sie bietet eine Menge von effizient implementierten Werkzeugen und Möglichkeiten für die meisten Anwendungen.
- Teile der C++-Standard-Bibliothek sind bereits behandelt.
- Wichtiger noch nicht behandelter Teil: Standard Template Library (STL)
- Die STL baut auf Templates auf.
- STL stellt im Besonderen Container und Iteratoren bereit.

Die C++-Standard-Bibliothek (Forts.)

Inhalte (bisher behandelt)

- Speicherverwaltung, Laufzeit-Typinformationen
- Informationen über implementierungsspezifische Aspekte, z. B. größter float-Wert
- spezielle Funktionen, z. B. sqrt()
- Hilfsmittel, die Portabilität ermöglichen, z. B. Listen, Sortierfunktionen, I/O-Streams
- Framework zur Erweiterung der Hilfsmittel, z. B. Techniken, um Ein-/Ausgaben für eigene Datentypen im Stil der eingebauten Datentypen anzubieten

Motivation...

Größtes Element suchen

- Ausgangssituation: Menge von geordneten Elementen
- Algrorithmus: HIGH enthält nach Beenden der Schleife den größten Wert

HIGH = first element
current_element = second element
while current_element is within the group of elements
    if current_element > HIGH, then HIGH = current_element
        Advance to the next element
end while

Dieser Algorithmus macht keine Annahmen über die Art der Speicherung der Elemente.

...Motivation...

Implementierung 1

Speicherung in verketteter Liste

```
struct Element
{
    int value;
    struct Element *next;
};
int high = list->value; // erstes Element
struct Element *current = list->next;
                        // zeigt auf zweites Element
while (current != NULL) // noch innerhalb der Menge?
{
    if (current->value > high) { high = current->value; }
    current = current->next; // naechstes Element
}
```

...Motivation...

Implementierung 2

Speicherung in Feld

```
int high = *array;
int *one_past_end = array + size;
int *current = array + 1;    // zweites Element

while (current != one_past_end)
// noch innerhalb der Menge?
{
    if (*current > high) { high = *current; }
    current++;       // naechstes Element
}
```

. . .Motivation. . .

- Beide Implementierungen sind konzeptionell gleich.
- Die Menge der Elemente ist als Sequenz betrachtet.
- Gemeinsame Operationen
  1. zeige auf ein Element
  2. greife auf das gezeigte Element zu
  3. zeige auf das nächste Element
  4. prüfe, ob noch innerhalb der Sequenz
- Grundidee von Containern und Iteratoren: Generische Datentypen und Operationen auf diesen
- Wichtige Konzepte der Realisierung: Operator-Overloading und Templates

. . .Motivation

- Die STL bietet Datenstrukturen in zeiger-ähnlicher Art
- Basis-Operatoren sind
  - Zuweisung: zeige auf ein Element
  - Dereferenzierung: greife auf ein gezeigtes Element zu
  - Zeigerarithmetik: zeige auf nächstes oder voriges Element
  - Vergleich: Prüfe zwei Elemente auf Gleichheit
- Implementierung in Template-Klassen

Vektoren

- Die STL stellt einen Container `vector` zur Verfügung
- Zur Benutzung wird die Header-Datei `<vector>` eingebunden
- Vektor von ganzen Zahlen:

```
vector<int> values;
```

- Der Iterator ist container-spezifisch, d. h. Implementierung als Member-Funktion

```
vector<int>::iterator current;
```

Beispiel Vektor...

Implementierung des „Finde größtes Element“-Beispiels mit STL-Container vector

```
vector<int>::iterator current = values.begin();
int high = *current++;

while (current != values.end())
{
    if (*current > high)
    {
        high = *current;
    }
    current++;
}
```

...Beispiel Vektor

Unterschiede zur Feld-Implementierung

- Statt Zeiger auf das erste Element ist ein Iterator deklariert.
- Die Sequenz, d. h. der Vektor besitzt eine Member-Funktion, die einen Zeiger auf den Anfang (`begin()`) und das Ende (`end()`) liefert.
- Der Rest ist identisch.
- Die Member-Funktionen `begin()` und `end()` existieren für alle STL-Container.

Beispiel Liste...

Implementierung des „Finde größtes Element“-Beispiels mit STL-Container `list`

```
list<int>::iterator current = values.begin();
int high = *current++;

while (current != values.end())
{
    if (*current > high)
    {
        high = *current;
    }
    current++;
}
```

. . .Beispiel Liste. . .

Unterschiede zur Implementierung mit verketteter Liste

- Andere Art der Dereferenzierung
- Verkettete Liste: `current->value`
- STL-Container `list`: `*current`
- Achtung: in STL-Container `list` ist `current` vom Typ `list<int>::iterator`.

...Beispiel Liste

Unterschiede zur Implementierung mit verketteter Liste

- Der unäre Operator * hat eine spezielle Bedeutung für Iteratoren, ist aber intuitiv wie eine Zeiger-Dereferenzierungs-Operator zu benutzen.
- Zeiger werden statt mit `current = current->next` mit `current++` weitergesetzt.
- Prüfen auf Ende der Liste geschieht mit `end()` statt mit Verleich auf `NULL`-Zeiger

Wichtige Container-Klassen

Header-Datei	Container
`<vector>`	eindimensionales Feld von Ts
`<list>`	doppelt verkettete Liste von Ts
`<deque>`	„double-ended“ Schlange von Ts
`<queue>`	Schlange (Queue) von Ts
`<stack>`	Stapel (Stack) von Ts
`<map>`	Assoziatives Feld von Ts
`<set>`	Menge von Ts
`<bitset>`	Feld von Wahrheitswerten

Member-Funktionen der meisten Container...

- `push_front`: Element vorn einfügen (nicht für `vector`)
- `pop_front`: erstes Element entfernen (nicht für `vector`)
- `push_back`: Element hinten anfügen
- `pop_back`: letztes Element entfernen
- `empty`: Prüfung, ob der Container leer ist
- `size`: Anzahl der Elemente im Container

...Member-Funktionen der meisten Container

- `insert`: Element an einer speziellen Stelle einfügen
- `erase`: Element an einer speziellen Stelle entfernen
- `clear`: Enternen aller Elemente aus dem Container
- `resize`: Größe des Containers verändern
- `front`: Zeiger auf Anfang des Containers
- `back`: Zeiger auf Ende des Containers
- `vektor` und deque besitzen Index-Zugriff `[]` (ohne Bereichsprüfung), `at` (mit Bereichsprüfung)

Bemerkungen zu Containern

- Die Benutzung der verschiedenen STL-Container geschieht in gleicher Weise.
- Die Effizienz kann sehr unterschiedlich sein.
- `list` ist gut geeignet für Einfügen und Entfernen von Elementen an beliebigen Stellen, `vector` nicht.
- Einfügen in einen `vector` bedeutet das Verschieben aller Elemente hinter der Einfügestelle und ist exterm ineffizient.
- `list` besitzt keinen Index-Zugriff, ist also nicht geeignet für binäre Suche, `vector` schon.

Benutzung von Containern...

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
int main() {
    vector<string> names;
    while (more_data()) {
        string temp = get_more_data();
        names.push_back(temp);
    }
// Ineffizient. Falls Anzahl der Elemente bekannt, besser
    names.resize (num_elements);
    for (int i = 0; i < num_elements; i++) {
        names.at(i) = get_more_data();
    }// Alternativ names[i] statt names.at(i)
```

...Benutzung von Containern...

```
    // Sortiere Elemtent, Voraussetzung:
    // Existenz einer Funktion
    // template <class T>
    // void sort (vector<T> &);
    sort(names);
    // Ausgabe
    for (int i = 0; i < names.size(); i++) {
        cout << names[i] << endl;
    }
    // Alternativ: Ausgabe mit Iterator
    vector<string>::iterator i;
    for (i = names.begin(); i != names.end(); ++i) {
        cout << *i << endl;
    }
    return (EXIT_SUCCESS);
}
```

...Benutzung von Containern...

```
#include <iostream>
#include <list>
#include <string>
using namespace std;
class Student
{
public:
    // bestanden, falls Leistung min. 50%
    bool passed () { return (mark >= 50); }
    ostream& operator << (ostream& os, const Student& s);

private:
    string name, ID;
    int mark;
};
```

...Benutzung von Containern...

```
int main()
{
    list<Student> students;

    // Lese "sutdents" ein

    while (more_students())
    {
        Student temp;
        temp.read ();
        students.push_back(temp);
    }
```

...Benutzung von Containern...

```
// gebe Studenten aus, die nicht bestanden haben
// benutzt member-Funktion passed ()
   list<Student>::iterator i;
   for (i = students.begin(); i != students.end(); ++i)
   {
// iteratoren besitzen operator ->
      if (! i->passed())
      {
         cout << "Student " << *i
              << " hat nicht bestanden."
              << endl;
// Voraussetzung: Klasse Student besitzt operator <<
      }
   }
```

...Benutzung von Containern

```
// Entferne Studenten, die nicht bestanden haben

   i = students.begin ()
   while (i != students.end ())
   {
      if (! i->passed ())
      {
         i = students.erase (i);
      }
      else
      {
         ++i;
      }
   }
   // ...
   return (EXIT_SUCCESS);
}
```

Basis-Sequenz-Operationen...

Grundlegende Sequenzen

- `vector`,
- `list`,
- `deque`.

Effizienzbetrachtung

- Sequenz-Container haben unterschiedliche Effizienz für verschiedene Einsatzgebiete.
- Im Folgenden werden die Eigenschaften im Detail betrachtet.

...Basis-Sequenz-Operationen...

```
#include <deque>
#include <iostream>
#include <list>
#include <vector>
using namespace std;

template<typename Container>
void print(Container& c, char* title = "") {
  cout << title << ':' << endl;
  if(c.empty()) {
    cout << "(empty)" << endl;
    return;
  }
```

. . .Basis-Sequenz-Operationen. . .

```
  typename Container::iterator it;
  for(it = c.begin(); it != c.end(); it++)
    cout << *it << " ";
  cout << endl;
  cout << "size() "       << c.size()
       << " max_size() " << c.max_size()
       << " front() "     << c.front()
       << " back() "      << c.back()
       << endl;
}
```

. . .Basis-Sequenz-Operationen. . .

Die Funktion `print()` liefert über jeden Sequenz-Container

- ob der Container leer ist
- die aktuelle Größe
- die maximal mögliche Größe
- das Element am Anfang
- das Element am Ende

. . .Basis-Sequenz-Operationen. . .

Konstruktoren

- Standard-Konstruktor,
- Mengen-Konstruktor
  erzeugt ein Objekt, das eine festgelegte Menge von Elementen enthält
- Start-Ende-Konstruktor
  erzeugt ein Objekt, das durch Zeiger auf erstes und letzes Element bestimmt ist
- Kopier-Konstruktor

. . .Basis-Sequenz-Operationen. . .

```
template<typename ContainerOfInt> void basicOps(char* s) {
  cout << "-------␣" << s << "␣-------" << endl;
  typedef ContainerOfInt Ci;
  Ci c;
  print(c, "c␣after␣default␣constructor");
  Ci c2(10, 1); // 10 Elemente, Werte alle 1
  print(c2, "c2␣after␣constructor(10,1)");
  int ia[] = { 1, 3, 5, 7, 9 };
  const int IASZ = sizeof(ia)/sizeof(*ia);
  // Initialisierung mit begin und end iterator:
  Ci c3(ia, ia + IASZ);
  print(c3, "c3␣after␣constructor(iter,iter)");
  Ci c4(c2); // Copy-constructor
  print(c4, "c4␣after␣copy-constructor(c2)");
```

. . .Basis-Sequenz-Operationen. . .

Zuweisungen

- Standard-Zuweisung (=)
- Mengen-Zuweisung (`assign()`)
  weist eine vorgebene Anzahl von Elementen mit einem vorgegebenen Startwert zu
- Start-Ende-Zuweisung (`assign()`)
  weist eine Menge von Elementen, die durch Start- und Ende-Iteratoren gegeben ist, zu

...Basis-Sequenz-Operationen...

```
c = c2; // Zuweisungs-Operator
print(c, "c after operator=c2");
c.assign(10, 2); // 10 Elemente, Werte alle 2
print(c, "c after assign(10, 2)");
// Zuweisung mit begin und end iterator:
c.assign(ia, ia + IASZ);
print(c, "c after assign(iter, iter)");
```

. . .Basis-Sequenz-Operationen. . .

Umkehrbarkeit

- Sequenz-Container sind umkehrbar.
- Die Bearbeitung geschieht in umgekehrter Reihenfolge (`rbegin()`, `rend()`).

Anpassbarkeit

- Die Größe von Sequenz-Containern kann verändert werden (`resize()`).
- Wenn der Sequenz-Container vergrößert wird, dann wird für die neuen Elemente der Standard-Konstruktor des Element-Typs verwendet, bei Standard-Typen wird mit „Null“-Werten initialisiert.

...Basis-Sequenz-Operationen...

```
cout << "c␣using␣reverse␣iterators:" << endl;
typename Ci::reverse_iterator rit = c.rbegin();
while(rit != c.rend())
  cout << *rit++ << "␣";
cout << endl;
c.resize(4);
print(c, "c␣after␣resize(4)");
c.push_back(47);
print(c, "c␣after␣push_back(47)");
c.pop_back();
print(c, "c␣after␣pop_back()");
```

. . .Basis-Sequenz-Operationen. . .

Einfügen und Löschen

- Einfügen eines Elementes an einer bestimmten Stelle, gegeben durch einen Iterator (`insert()`)
- Einfügen von mehreren Elementen an einer Stelle, gegeben durch einen Iterator und der Elementanzahl (`insert()`)
- Einfügen einer Sequenz von Elemmenten an einer bestimmten Stelle, gegeben durch einen Iterator und Start- und Ende-Iteratoren der einzufügenden Sequenz (`insert()`)

...Basis-Sequenz-Operationen...

```
typename Ci::iterator it = c.begin();
++it; ++it;
c.insert(it, 74);
print(c, "c␣after␣insert(it,␣74)");
it = c.begin();
++it;
c.insert(it, 3, 96);
print(c, "c␣after␣insert(it,␣3,␣96)");
it = c.begin();
++it;
c.insert(it, c3.begin(), c3.end());
print(c, "c␣after␣insert("
  "it,␣c3.begin(),␣c3.end())");
```

. . .Basis-Sequenz-Operationen. . .

Löschen und Vertauschen

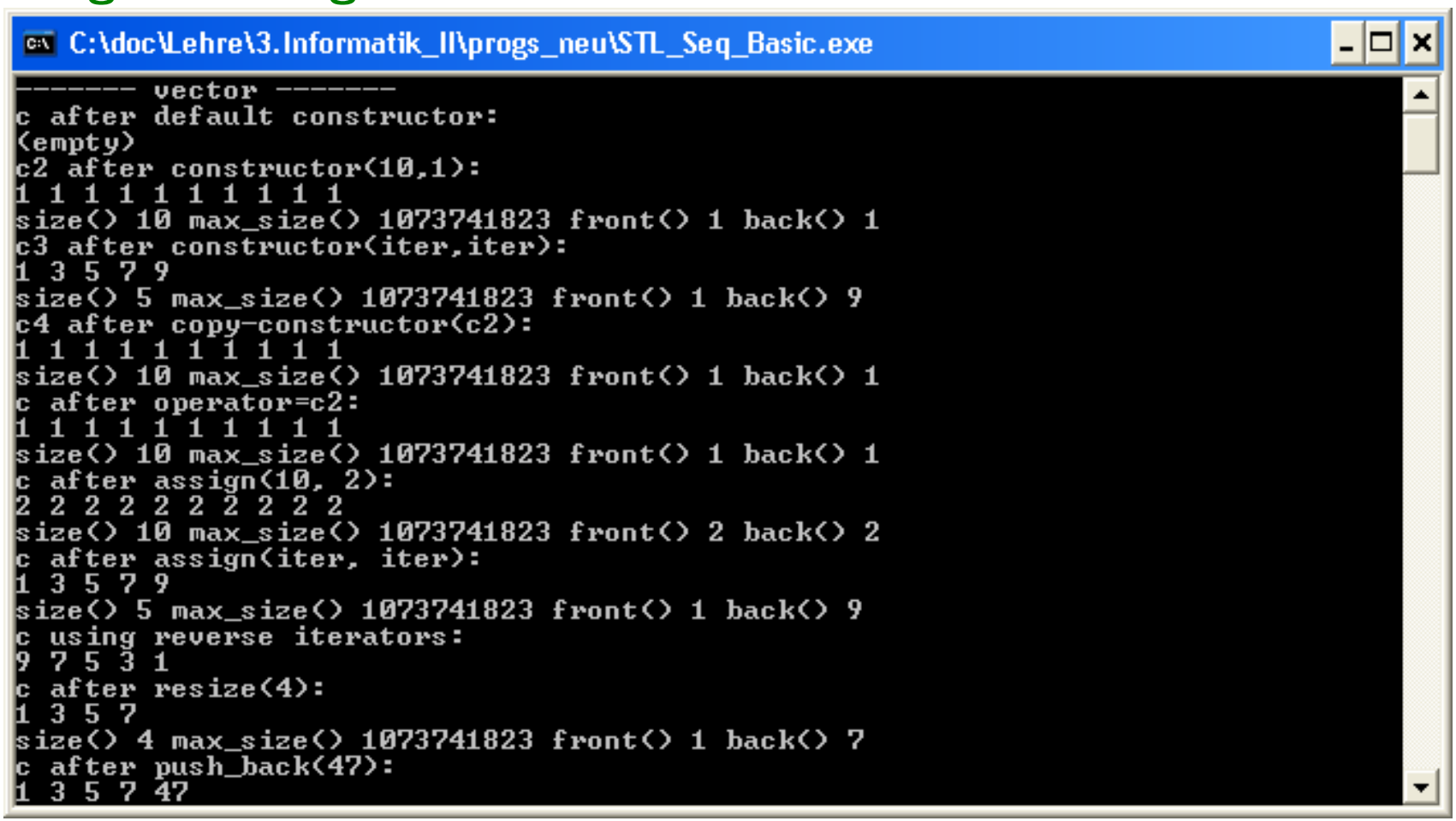
- Löschen eines Elements an einer bestimmten Stelle, gegeben durch einen Iterator (`erase()`)
- Löschen einer Sequenz von Elementen, gegeben durch Start- und Ende-Iteratoren (`erase()`)
  - Bei list-Container muss hierfür Inkrement (++) und Dekrement(–) verwendet werden.
  - Bei vector und deque kann der Index auch mit `operator`+ oder `operator`- angegeben werden.
- Vertauschen des Inhalts zweier Container (`swap()`), funktioniert nur bei Containern gleichen Element-Typs
- Löschen des gesamten Container-Inhalts (`clear()`)

. . .Basis-Sequenz-Operationen. . .

```
  it = c.begin();
  ++it;
  c.erase(it);
  print(c, "c after erase(it)");
  typename Ci::iterator it2 = it = c.begin();
  ++it;
  ++it2; ++it2; ++it2; ++it2; ++it2;
  c.erase(it, it2);
  print(c, "c after erase(it, it2)");
  c.swap(c2);
  print(c, "c after swap(c2)");
  c.clear();
  print(c, "c after clear()");
}
```

. . .Basis-Sequenz-Operationen. . .

Programmausgabe



Der vector-Container

Eigenschaften

- Der vector-Container speichert die Elemente in hintereinanderliegenden Speicherbereichen.
- vector ist dynamisch vergößerbar.
- Einfügen in vector ist nur effizient, wenn am Ende angefügt wird.
- Bei Vergrößerung eines vector-Containers muss in der Regel
  - ein neuer zusammenhängender Speicherbereich reserviert werden,
  - die vorhandenen Elemente umkopiert werden,
  - die alten Objekte zertört werden,
  - der alte Speichbereich freigegeben werden.
- Das ist bei komplexen Objekten teuer.

Der vector-Container

- Beispiel, was bei Objekt-Erzeugung und -Zerstörung passiert: „Noisy“-Klasse
- Jedes Noisy-Objekt hat einen Identifikator (statc-Variable).
- Alle Erzeugungen und Zuweisungen werden verfolgt.
- Bei Zuweisungen wird der alte Wert des Objekts auf der rechten Seite vor dem Überschreiben ausgegeben.
- Am Ende gibt es einen „NoisyReport“, der über den Destruktor eine Statistik der statischen Variablen der Klasse Noisy druckt.

Der vector-Container (Noisy.h)...

```
// Verfolgung von Objekt-Aktiviaeten.
#include <iostream>
using std::endl;
using std::cout;
using std::ostream;
class Noisy {
  static long create, assign, copycons, destroy;
  long id;
public:
  Noisy() : id(create++) {
    cout << "d[" << id << "]" << endl;
  }
  Noisy(const Noisy& rv) : id(rv.id) {
    cout << "c[" << id << "]" << endl;
    ++copycons;
  }
```

...Der vector-Container (Noisy.h)...

```
Noisy& operator=(const Noisy& rv) {
  cout << "(" << id << ")=[" << rv.id << "]" << endl;
  id = rv.id;
  ++assign;
  return *this;
}
friend bool operator<(const Noisy& lv, const Noisy& rv) {
  return lv.id < rv.id;
}
friend bool operator==(const Noisy& lv,const Noisy& rv) {
  return lv.id == rv.id;
}
~Noisy() {
  cout << "~[" << id << "]" << endl;
  ++destroy;
}
```

...Der vector-Container (Noisy.h)...

```
  friend ostream& operator<<(ostream& os, const Noisy& n) {
    return os << n.id;
  }
  friend class NoisyReport;
};

struct NoisyGen {
  Noisy operator()() { return Noisy(); }
};
```

...Der vector-Container (Noisy.h)

```
// Gibt bei Programm-Ende eine Statistik aus.
class NoisyReport {
  static NoisyReport nr;
  NoisyReport() {} // Private constructor
  NoisyReport & operator=(NoisyReport &);  // Disallowed
  NoisyReport(const NoisyReport&);         // Disallowed
public:
  ~NoisyReport() {
    cout << "\n--------------------\n"
         << "Noisy creations: " << Noisy::create
         << "\nCopy-Constructions: " << Noisy::copycons
         << "\nAssignments: " << Noisy::assign
         << "\nDestructions: " << Noisy::destroy << endl;
  }
};
```

Der vector-Container (Noisy.cpp)

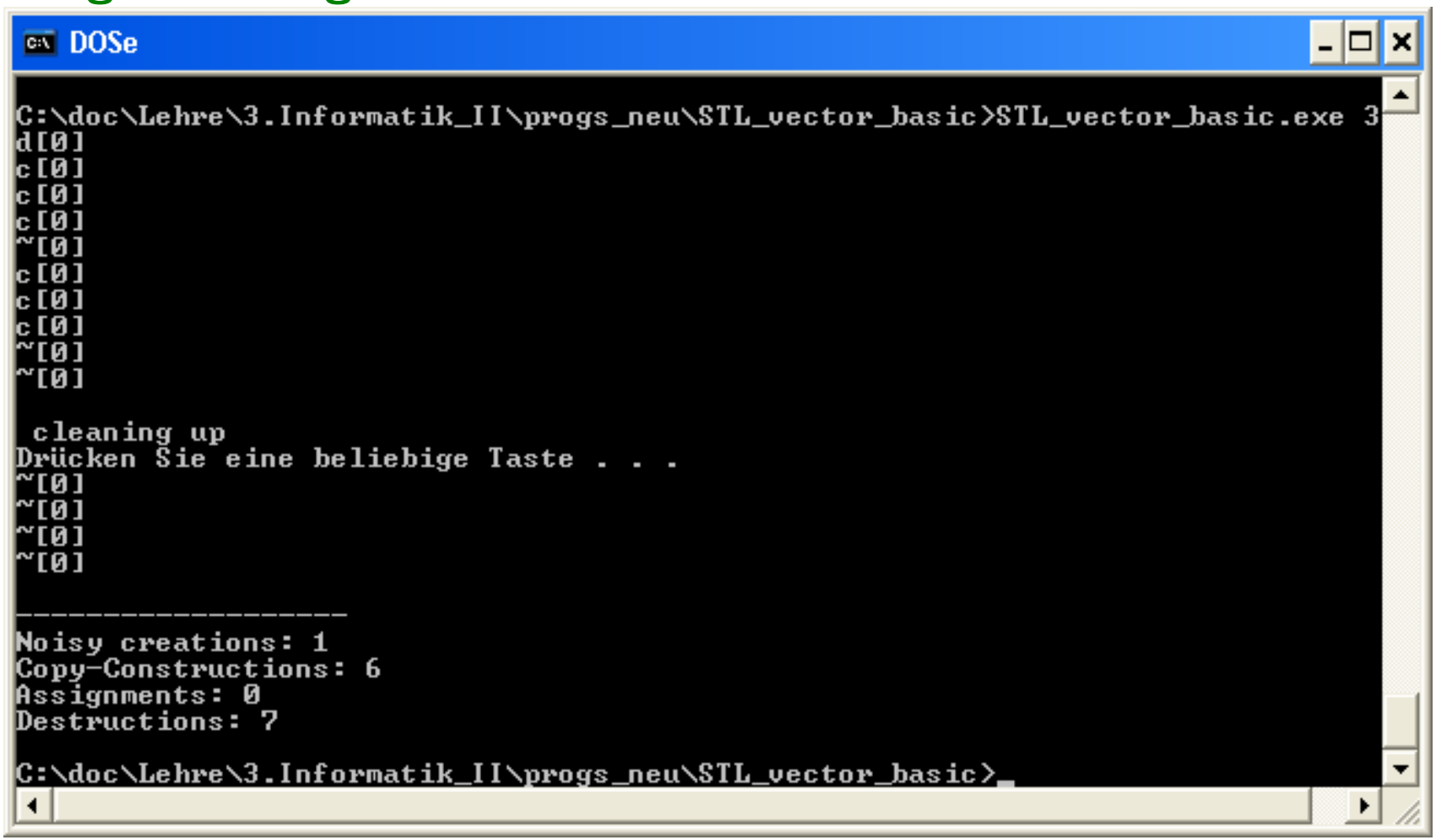
```
#include "Noisy.h"
long Noisy::create = 0, Noisy::assign = 0,
  Noisy::copycons = 0, Noisy::destroy = 0;
NoisyReport NoisyReport::nr;
```

Der vector-Container (main.cpp)

```
// Zeigt Kopier-Konstruktion und Destruktion, wenn
// ein vector Groessenangepasst wird.
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include "Noisy.h"
using namespace std;
int main(int argc, char* argv[]) {
  int size = 1000;
  if(argc >= 2) size = atoi(argv[1]);
  vector<Noisy> vn;
  Noisy n;
  for(int i = 0; i < size; i++)
    vn.push_back(n);
  cout << "\n cleaning up " << endl;
}
```

Der vector-Container (Noisy)

Programmausgabe



Der vector-Container...

- Was passiert mit dem Wert eines Iterators, wenn umkopiert wird?
- Umkopieren eines vector erfolgt z. B. bei Größenanpassung
- Iterator zeigt auf falschen Speicher!
- Grund: Efiiziente Zeiger-Implementierung von Iteratoren von vector.

...Der vector-Container...

```
// Ungueltigmachen eines Iterators.
#include <iterator>
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> vi(10, 0);
  ostream_iterator<int> out(cout, "␣");
  vector<int>::iterator i = vi.begin();
  *i = 47;
  copy(vi.begin(), vi.end(), out);
  cout << endl;
```

...Der vector-Container

```
  // Erzwinge Speicherbewegung
  vi.resize(vi.capacity() + 1);
  // Jetzt zeigt i auf einen falschen Speicher
  *i = 48;  // Zugriffsverletzung
  copy(vi.begin(), vi.end(), out); // No change to vi[0]
}
```

Der vector-Container (Invalider Iterator)

Programmausgabe