Universität des Saarlandes
FR 6.2 Informatik
Prof. Dr. Hans-Peter Lenhof
Dipl. Inform. Andreas Hildebrandt

Programmierung II, SS 2003
Musterlösung zur 3. Übung

Hinweis: Wie in der Vorlesung angekündigt, gehen die ersten drei Übungen nicht in die Wertung ein!

Zum Kompilieren der Musterlösungen befindet sich ein Makefile im Verzeichnis zur 3.Übung.

```
aufgabe1:
  g++ quicksort.cpp -o quicksort -g -Wall -pedantic

aufgabe2:
  g++ vorlesungsteilnehmer.cpp -o vorlesungsteilnehmer -g -Wall -pedantic

aufgabe3:
  g++ myVector.cpp -o myVector -g -Wall -pedantic
```

Mit folgenden Aufrufen kann man damit die Musterlösungen kompilieren.

- `make aufgabe1` Musterlösung zur Aufgabe 1 wird kompiliert.
- `make aufgabe2` Musterlösung zur Aufgabe 2 wird kompiliert.
- `make aufgabe3` Musterlösung zur Aufgabe 3 wird kompiliert.

Aufgabe 1: QuickSort
Der folgende Code befindet sich in der Datei **quicksort.cpp**

```
#include <iostream>
#include <vector>

using namespace std;

/** Diese Funktion uebernimmt das Partitionieren
 *  in Teilarrays V1 und V2, so dass alle Elemente
 *  von V1 kleiner als alle Elemente von V2 sind.
 */
unsigned int partition(vector<int>& v, int p, int q)
{
  // Wir nehmen das erste Element als Pivotelement
  int pivot = v[p];

  // Um unsere beiden Teilarrays zu schaffen, so dass
```

```
  // "links" alle Elemente kleiner oder gleich dem Pivot-
  // element stehen und rechts alle, die groesser sind,
  // durchsuchen wir gleichzeitig von links und rechts,
  // und tauschen die ersten gefundenen Elemente aus, die
  // "fehl am Platz" sind
  int links  = p;
  int rechts = q;

  while (links < rechts)
  {
    if (v[rechts] > pivot)
    {
      rechts--;
    }
    else
    {
      if (v[links] <= pivot)
      {
        links++;
      }
      else
      {
        int tmp   = v[links];
        v[links]  = v[rechts];
        v[rechts] = tmp;
      }
    }
  }

  // Jetzt muss nur noch das aktuell betrachtete Element
  // mit dem Pivot vertausch werden
  v[p]     = v[rechts];
  v[rechts] = pivot;

  // und die richtige Position zurueckgegeben werden
  return rechts;
}

/** Diese Funktion implementiert den Quicksort-
 *  Algorithmus fuer vector<int>.
 */
void quicksort(vector<int>& v, int p, int q)
{
  // sind wir vielleicht schon fertig? oder ist
  // unterwegs was seltsames passiert?
  if ( p < q )
  {
    // na gut, wir muessen doch noch was machen
```

```
    unsigned int r = partition(v, p, q);
    quicksort(v, p, r-1);
    quicksort(v, r+1, q);
  }
}

// Ein kleines Hauptprogramm, um unseren Algorithmus
// zu testen
int main(int argc, char **argv)
{
  // Ueberpruefen der Kommandozeilenargumente
  if (argc < 3)
  {
    cout << "Aufruf mittels " << argv[0] << " seed laenge" << endl;

    return 0;
  }

  // Erstellen eines passenden Vektors
  vector<int> v(atoi(argv[1]));

  // Initialisieren des Zufallszahlengenerators
  srand(atoi(argv[2]));

  // Fuellen mit zufaelligen Werten
  for (unsigned int i = 0; i < v.size(); i++)
  {
    v[i] = rand();
  }

  // Sortieren
  quicksort(v, 0, v.size()-1);

  // Ausgabe
  for (unsigned int i = 0; i < v.size(); i++)
  {
    cout << i << " " << v[i] << endl;
  }

  // Und fertig.
  return 0;
}
```

Aufgabe 2: Klassen, Teil 1
Der folgende Code befindet sich in der Datei **vorlesungsteilnehmer.h**

```
#include <string>
#include <vector>

using namespace std;

/** Die Klassendefinition */
class Vorlesungsteilnehmer
{
  public:

    // Ein Defaultkonstruktor
    Vorlesungsteilnehmer();

    // Ein etwas detaillierterer Konstruktor
    Vorlesungsteilnehmer(const string& name,
                         const long int matrikelnummer,
                         const string& studienfach);


    // Der Vollstaendigkeit halber implementieren wir
    // auch noch einen Copyconstructor und einen Assignmentoperator
    Vorlesungsteilnehmer(const Vorlesungsteilnehmer& vt);

    Vorlesungsteilnehmer& operator = (const Vorlesungsteilnehmer& vt);

    // Der Destruktor
    ~Vorlesungsteilnehmer();

    // Und wo wir schon dabei sind auch die Vergleichsoperatoren
    bool operator == (const Vorlesungsteilnehmer& vt) const;
    bool operator != (const Vorlesungsteilnehmer& vt) const;

    // Zugriff auf die Membervariablen.
    string liesName() const;
    long int liesMatrikelnummer() const;
    string liesStudienfach() const;

    void setzeName(const string& name);
    void setzeMatrikelnummer(const long int matrikelnummer);
    void setzeStudienfach(const string& studienfach);

    int  liesPunkte(const int blatt) const;
    void setzePunkte(const int blatt, const int punkte);

    void bestehensgrenze(const int bestehensgrenze_);
    bool bestanden() const;
  protected:
    // Alle unsere protected - Variablen bekommen
```

```
    // einen Unterstrich "_" verpasst, um sie deutlich
    // zu kennzeichnen und um Probleme mit globalen Variablen
    // gleichen Namens zu minimieren

    // Der Name
    string name_;

    // Die Matrikelnummer
    long int matrikelnummer_;

    // Das Studienfach
    string studienfach_;

    // Die erreichten Punktzahlen
    vector<int> punkte_;

    // Die Bestehensgrenze
    int bestehensgrenze_;
};
```

Der folgende Code befindet sich in der Datei **vorlesungsteilnehmer.cpp**

```
#include <iostream>

#include "vorlesungsteilnehmer.h"

using namespace std;

// In unserem Defaultkonstruktor koennen wir die Defaultkonstruktoren
// unserer Membervariablen aufrufen. Die wissen hier schon selbst
// am Besten, was "sinnvolle" Standardwerte sind.
Vorlesungsteilnehmer::Vorlesungsteilnehmer()
  : name_(),
    matrikelnummer_(),
    studienfach_(),
    punkte_(),
    bestehensgrenze_()
{
}

// Im detailierten Konstruktor koennen wir einfach die
// Copy-Konstruktoren der Membervariablen aufrufen
Vorlesungsteilnehmer::Vorlesungsteilnehmer(const string& name,
        const long int matrikelnummer, const string& studienfach)
  : name_(name),
    matrikelnummer_(matrikelnummer),
    studienfach_(studienfach),
    punkte_(),
    bestehensgrenze_()
{
```

```
}

// Der Copy-Konstruktor ist einfach...
Vorlesungsteilnehmer::Vorlesungsteilnehmer(const Vorlesungsteilnehmer& vt)
  : name_(vt.name_),
    matrikelnummer_(vt.matrikelnummer_),
    studienfach_(vt.studienfach_),
    punkte_(vt.punkte_),
    bestehensgrenze_(vt.bestehensgrenze_)
{
}

// Unser Destruktor braucht nichts zu tun. Wir haben
// ja keinen Speicher allokiert.
Vorlesungsteilnehmer::~Vorlesungsteilnehmer()
{
}

// Und nun der Assignmentoperator
Vorlesungsteilnehmer& Vorlesungsteilnehmer::operator = (const Vorlesungsteilnehmer& v
{
  // Self-assignment ist _immer_ eine problematische Sache
  if ( &vt == this )
  {
    return *this;
  }

  name_ = vt.name_;
  matrikelnummer_ = vt.matrikelnummer_;
  studienfach_ = vt.studienfach_;
  punkte_ = vt.punkte_;
  bestehensgrenze_ = vt.bestehensgrenze_;

  return *this;
}

// Und die Vergleiche
bool Vorlesungsteilnehmer::operator == (const Vorlesungsteilnehmer& vt) const
{
  return (    (name_ == vt.name_)
           && (matrikelnummer_ == vt.matrikelnummer_)
           && (studienfach_ == vt.studienfach_)
           && (punkte_ == vt.punkte_)
           && (bestehensgrenze_ == vt.bestehensgrenze_)
         );
}

bool Vorlesungsteilnehmer::operator != (const Vorlesungsteilnehmer& vt) const
```

```
{
  return (    (name_ != vt.name_)
           && (matrikelnummer_ != vt.matrikelnummer_)
           && (studienfach_ != vt.studienfach_)
           && (punkte_ != vt.punkte_)
           && (bestehensgrenze_ != vt.bestehensgrenze_)
         );
}

// Nun die Zugriffe
string Vorlesungsteilnehmer::liesName() const
{
  return name_;
}

long int Vorlesungsteilnehmer::liesMatrikelnummer() const
{
  return matrikelnummer_;
}

string Vorlesungsteilnehmer::liesStudienfach() const
{
  return studienfach_;
}

void Vorlesungsteilnehmer::setzeName(const string& name)
{
  name_ = name;
}

void Vorlesungsteilnehmer::setzeMatrikelnummer(const long int matrikelnummer)
{
  matrikelnummer_ = matrikelnummer;
}

void Vorlesungsteilnehmer::setzeStudienfach(const string& studienfach)
{
  studienfach_ = studienfach;
}

int Vorlesungsteilnehmer::liesPunkte(const int blatt) const
{
  if ((int)punkte_.size() > blatt)
  {
    return punkte_[blatt];
  }
  else
  {
```

```
    return -1;
  }
}

void Vorlesungsteilnehmer::setzePunkte(const int blatt, const int punkte)
{
  if ((int)punkte_.size() > blatt)
  {
    punkte_[blatt] = punkte;
  }
  else
  {
    // wir muessen erst noch Platz schaffen...
    punkte_.resize(blatt+1);
    punkte_[blatt] = punkte;
  }
}

void Vorlesungsteilnehmer::bestehensgrenze(const int punkte)
{
  bestehensgrenze_ = punkte;
}

bool Vorlesungsteilnehmer::bestanden() const
{
  // summiere alle Punkte auf
  int summe = 0;

  for (unsigned int i = 0; i < punkte_.size(); i++)
  {
    summe+=punkte_[i];
  }

  return (summe >= bestehensgrenze_);
}


// nun basteln wir noch ein paar Tests ins Hauptprogramm
int main(int argc, char** argv)
{
  Vorlesungsteilnehmer vt1, vt2, vt3("Karl Mustermann", 123456, "Informatik");

  vt1 = Vorlesungsteilnehmer("Karla Musterfrau", 123456, "Informatik");
  vt2.setzeName("Karla Musterfrau");
  vt2.setzeMatrikelnummer(123456);
  vt2.setzeStudienfach("Informatik");

  if (vt1 == vt3)
```

```
  {
    cerr << "Huch, da stimmt was nicht." << endl;
    return 1;
  }

  if (vt2 != vt1)
  {
    cerr << "Huch, da stimmt was nicht." << endl;
    return 1;
  }

  vt2.setzePunkte(1, 40);

  if (vt2 == vt1)
  {
    cerr << "Huch, da stimmt was nicht." << endl;
    return 1;
  }

  vt1.bestehensgrenze(39);
  vt2.bestehensgrenze(39);

  if (vt1.bestanden() || !(vt2.bestanden()))
  {
    cerr << "Huch, da stimmt was nicht." << endl;
    return 1;
  }

  cout << "Hat funktioniert!" << endl;

  return 0;
}
```

Aufgabe 3: Klassen, Teil 2
Der folgende Code befindet sich in Datei **myVector.h**

```
#include <vector>

using namespace std;

// Diese Musterloesung enthaelt nur die templatisierte
// Fassung. Die nicht-template Version unterscheidet sich
// davon allerdings nur unwesentlich
// Bei Templateklassen steht auch die Implementierung im Headerfile!

/** Unsere vector-Klasse */
```

```
template<typename T>
class myVector
{
  public:
    // Ein Defaultkonstruktor muss sein
    myVector();

    // Dieser Konstruktor uebernimmt einen vector<T>
    myVector(const vector<int>& v);

    // Dieser Konstruktor schafft Platz fuer n Elemente
    myVector(const unsigned int n);

    // Und dies ist schliesslich der copy-Konstruktor
    myVector(const myVector<T>& mv);

    // Und schliesslich ein Destruktor
    ~myVector();

    // Fuellt mit n Zufallszahlen
    void zufallswerte(const int n);

    // Haengt ein element an den vector an
    void elementAnfuegen(T element);

    // Zugriff von aussen auf die Elemente
    T& element(const int n);

    // Lesender Zugriff von aussen auf die Elemente
    const T& element(const int n) const;

    // Auswahl des Sortierverfahrens.
    //    0 steht fuer InsertionSort
    //    1 steht fuer MergeSort
    //    2 steht fuer QuickSort
    // Gibt "false" zurueck, falls eine Zahl ausser 0, 1 oder 2
    // angegeben wird.
    bool sortierverfahren(const int verfahren);

    // Sortiert den vector
    void sort();

    // Gibt die Laenge des Vektors zurueck
    unsigned int laenge() const;

    // Sucht nach dem Element T
    unsigned int suche(T element);
```

```
  protected:
    // InsertionSort. Wir verstecken die Funktion hier,
    // da ja alle Aufrufe ueber "sort" gekapselt werden sollen
    void insertionSort_();

    // Hilfsfunktion fuer MergeSort
    void merge_(unsigned int start1, unsigned int end1, unsigned int end2);

    // MergeSort.
    void mergeSort_(unsigned int p, unsigned int r);

    // Hilfsfunktion fuer QuickSort
    unsigned int partition_(unsigned int p, unsigned int q);

    // QuickSort
    void quickSort_(unsigned int p, unsigned int q);

    // BinarySearch
    unsigned int binarySearch_(int gesucht, int von, int bis);

    // Der eigentliche Datenvector
    vector<T> daten_;

    // Ein flag, dass uns sagt, ob wir schon sortiert haben
    bool istSortiert_;

    // Ein flag, dass angibt, dass der vector jederzeit
    // von aussen veraendert werden kann -> sortieren vor
    // _jeder_ suche ist noetig
    bool immerSortieren_;

    // Ein flag, dass uns sagt, welche Suchmethode wir verwenden sollen
    int methode_;
};

template<typename T>
myVector<T>::myVector()
  : daten_(),
    istSortiert_(false),
    immerSortieren_(false),
    methode_()
{
}

template<typename T>
myVector<T>::myVector(const vector<int>& v)
  : daten_(v),
    istSortiert_(false),
```

```
    immerSortieren_(false),
    methode_()
{
}

template<typename T>
myVector<T>::myVector(const unsigned int n)
  : daten_(n),
    istSortiert_(false),
    immerSortieren_(false),
    methode_()
{
}

template<typename T>
myVector<T>::myVector(const myVector<T>& mv)
  : daten_(mv.daten_),
    istSortiert_(mv.istSortiert_),
    immerSortieren_(false),
    methode_(mv.methode_)
{
}

template<typename T>
myVector<T>::~myVector()
{
}

template<typename T>
void myVector<T>::zufallswerte(const int n)
{
  // Zuerst passen wir die Laenge an
  daten_.resize(n);

  // Dann fuellen wir. Da auf dem Uebungsblatt nichts
  // von einem bestimmten seed gesagt war, nehmen wir irgendeinen.
  // Ein wenig trickreich ist hier, dass wir Daten vom Typ "T" brauchen
  srand(42);

  for (unsigned int i = 0; i < daten_.size(); i++)
  {
    daten_[i] = (T) rand(); // wir casten geeignet
  }

  istSortiert_ = false;
}

template<typename T>
```

```
void myVector<T>::elementAnfuegen(T element)
{
  daten_.push_back(element);
  istSortiert_ = false;
}

template<typename T>
T& myVector<T>::element(const int n)
{
  // Wir geben eine Referenz zurueck, d.h. man kann
  // von aussen den vector verändern. Wir koennen uns
  // ab jetzt nicht mehr darauf verlassen, dass der
  // vector noch sortiert ist, selbst wenn sort()
  // aufgerufen wird!
  immerSortieren_ = true;

  return daten_[n];
}

template<typename T>
const T& myVector<T>::element(const int n) const
{
  // Hier sieht die Sache anders aus: eine const - Referenz
  // verhindert die Veränderung von aussen! Wir muessen uns
  // also um die Sortierung keine Gedanken machen!

  return daten_[n];
}

template<typename T>
bool myVector<T>::sortierverfahren(const int verfahren)
{
  if ( (verfahren < 0) || (verfahren > 2) )
  {
    return false;
  }

  methode_ = verfahren;
  return true;
}

template<typename T>
void myVector<T>::sort()
{
  // Vielleicht muessen wir ja gar nichts mehr tun...
  if (!istSortiert_ || immerSortieren_)
  {
    // Leider doch.
```

```
    switch (methode_)
    {
      case 0: insertionSort_();
              istSortiert_ = true;
              break;

      case 1: mergeSort_(0, laenge()-1);
              istSortiert_ = true;
              break;

      case 2: quickSort_(0, laenge()-1);
              istSortiert_ = true;
              break;
    }
  }
}

template<typename T>
void myVector<T>::insertionSort_()
{
  T naechster;

  // Wir muessen einmal ueber den Vektor iterieren,
  // wobei wir das erste Element ueberspringen koennen
  // (ein einelementiger Vektor ist schon sortiert)
  for (int j=1; j<(int) daten_.size(); j++)
  {
    // betrachte das naechste noch nicht einsortierte
    // Element
    naechster = daten_[j];

    // und sortiere es an der passenden Stelle ein
    int i = j-1;

    while ((i >= 0) && (daten_[i] > naechster))
    {
      // Wir muessen alle Zahlen die groesser sind
      // als naechster um eine Position nach rechts
      // verschieben.
      daten_[i+1] = daten_[i];

      --i;
    }

    daten_[i+1] = naechster;
  }
}
```

```
// merge_ fuegt die beiden benachbarten sortierten Teilarrays v(start1,...,end1) und
// v(end1+1,...,end2) zu einem sortierten Array zusammen, und kopiert dieses dann
// in v zurueck
template<typename T>
void myVector<T>::merge_(unsigned int start1, unsigned int end1, unsigned int end2)
{
  unsigned int index1=start1;
  unsigned int index2=end1+1;

  vector<T> sortiert(end2-start1+1);

  for (int index3=0; index3<(int)sortiert.size(); index3++)
  {
    if ( index1 == end1+1 )
    {
      sortiert[index3] = daten_[index2];
      index2++;
      continue;
    }
    if ( index2 == end2+1 )
    {
      sortiert[index3] = daten_[index1];
      index1++;
      continue;
    }
    if ( daten_[index1] > daten_[index2] )
    {
      sortiert[index3] = daten_[index2];
      index2++;
    }
    else
    {
      sortiert[index3] = daten_[index1];
      index1++;
    }
  }

  // kopiert den neuen vektor in den alten vektor
  for (int i=0; i<(int)sortiert.size(); i++)
  {
    daten_[i+start1] = sortiert[i];
  }
}

template<typename T>
void myVector<T>::mergeSort_(unsigned int p, unsigned int r)
{
  // Sind wir vielleicht schon fertig? Dann hat unser zu betrachtendes
```

```
  // Array die Laenge 0
  if (p < r)
  { // nein. schade.
    int q = (p+r)/2; // halbiere den vector

    // und sortiere rekursiv
    mergeSort_(p, q);
    mergeSort_(q+1, r);

    // fehlt nur noch das Kombinieren
    merge_(p, q, r);
  }
}

/** Diese Funktion uebernimmt das Partitionieren
 *  in Teilarrays V1 und V2, so dass alle Elemente
 *  von V1 kleiner als alle Elemente von V2 sind.
 */
template<typename T>
unsigned int myVector<T>::partition_(unsigned int p, unsigned int q)
{
  // Wir nehmen das erste Element als Pivotelement
  T pivot = daten_[p];

  // Um unsere beiden Teilarrays zu schaffen, so dass
  // "links" alle Elemente kleiner oder gleich dem Pivot-
  // element stehen und rechts alle, die groesser sind,
  // durchsuchen wir gleichzeitig von links und rechts,
  // und tauschen die ersten gefundenen Elemente aus, die
  // "fehl am Platz" sind
  int links  = p;
  int rechts = q;

  while (links < rechts)
  {
    if (daten_[rechts] > pivot)
    {
      rechts--;
    }
    else
    {
      if (daten_[links] <= pivot)
      {
        links++;
      }
      else
      {
        T tmp   = daten_[links];
```

```
        daten_[links]  = daten_[rechts];
        daten_[rechts] = tmp;
      }
    }
  }

  // Jetzt muss nur noch das aktuell betrachtete Element
  // mit dem Pivot vertausch werden
  daten_[p]      = daten_[rechts];
  daten_[rechts] = pivot;

  // und die richtige Position zurueckgegeben werden
  return rechts;
}

/** Diese Funktion implementiert den Quicksort-
 *  Algorithmus fuer vector<int>.
 */
template<typename T>
void myVector<T>::quickSort_(unsigned int p, unsigned int q)
{
  // sind wir vielleicht schon fertig? oder ist
  // unterwegs was seltsames passiert?
  if ( p < q )
  {
    // na gut, wir muessen doch noch was machen
    unsigned int r = partition_(p, q);
    quickSort_(p, r-1);
    quickSort_(r+1, q);
  }
}

template<typename T>
unsigned int myVector<T>::laenge() const
{
  return daten_.size();
}

template<typename T>
unsigned int myVector<T>::suche(T element)
{
  if (!istSortiert_ || immerSortieren_)
  {
    sort();
  }
  return binarySearch_(element, 0, laenge()-1);
}
```

```
template<typename T>
unsigned int myVector<T>::binarySearch_(int gesucht, int von, int bis)
{
  // vielleicht sind wir ja schon fertig?
  if (von == bis)
  {
    // entweder wir haben
    // das element gefunden...
    if (daten_[von] == gesucht)
    {
      return von;
    }
    else // oder es kommt nicht vor.
    {
      return -1;
    }
  }

  // ansonsten: sehen wir uns die Mitte des zu durchsuchenden
  //            Bereichs mal an...
  int mitte = (int) floor((bis - von) / 2.) + von;
  T daten_mitte = daten_[mitte];

  if (daten_mitte == gesucht) // wir sind fertig...
  {
    return mitte;
  }
  else
  {
    if (daten_mitte < gesucht) // die untere Haelfte ist uninteressant
    {
      return (binarySearch_(gesucht, mitte+1, bis));
    }
    else // die obere Haelfte ist uninteressant
    {
      return (binarySearch_(gesucht, von, mitte-1));
    }
  }
}


// Mit einem Typedef koennen wir auch noch myIntegerVector basteln
typedef myVector<int> myIntegerVector;
```

Der folgende Code befindet sich in Datei **myVector.cpp**

```
#include <iostream>
```

```
#include "myVector.h"

// Ein kleiner Test unseres Vektors
int main(int argc, char** argv)
{
  myVector<float> v1, v2, v3;

  v1.zufallswerte(12);
  v1.sort();

  v2.zufallswerte(12);
  v2.sortierverfahren(1);
  v2.sort();

  v3.zufallswerte(12);
  v3.sortierverfahren(2);
  v3.sort();

  for (unsigned int i=0; i<v1.laenge(); i++)
  {
    cout << i << " " << v1.element(i)
              << " " << v2.element(i)
              << " " << v3.element(i) << endl;
  }

  cout << endl;

  myIntegerVector v4;
  v4.zufallswerte(123);

  int element = v4.element(42);

  unsigned int n = v4.suche(element);

  if (v4.element(n) == element)
  {
    cout << "Hat funktioniert!" << endl;
  }
  else
  {
    cerr << "Hupps! Da stimmt was nicht!" << endl;
    return 1;
  }

  return 0;
}
```