BERGISCHE UNIVERSITÄT
GESAMTHOCHSCHULE WUPPERTAL
GAUSS-STRASSE 20
42097 WUPPERTAL
(Korrespondenzanschrift)
42119 WUPPERTAL
(Lieferanschrift)
TELEX 8 592 262 bughw
TELEFAX (0202) 439-2901
TELEFON (0202) 439-1

Fachbereich 7
MATHEMATIK

Prof. Dr. Hans-Jürgen Buhl
Praktische Informatik / Numerik

e-mail: Juergen.Buhl@math.uni-wuppertal.de

Informatik I

WS98/99 – Übungsblatt 14

— optional —

Aufgabe 1. *Vektoren als Klasse, Forts.: Zusammenfügen von Vektoren*

Benutzen Sie das Ergebnis der Aufgabe 5 des vorherigen Übungsblattes zur Konstruktion eines Zusammenfügeoperators

```
vektor operator|(const vektor&, const vektor&)
```

und testen Sie Ihr Ergebnis.

Aufgabe 2. *Vorbereitung: Matrizen als Klasse, hier: Indexarithmetik*

In der folgenden Klasse `matrix`

```
//////////////////////////////////
// Datei:   matrix.cc
// Version: 1
// Zweck:   matrix als Klasse
// Autor:   Hans-Juergen Buhl
// Datum:   10.02.99
//////////////////////////////////

#include <iostream>
#include <bool.h>

//using namespace std;

class matrix{

  int low1;        // v(low1..high1, low2..high2)
  int high1;               // high1 >= low1
  int low2;
  int high2;               // high2 >= low2

  double* v;         // Startadresse fuer dyn. verwaltetes Exemplar

public:
```



```
  matrix(const int h1, const int l1, const int h2, const int l2,
         const double d = 0.0);

  ~matrix(){ delete []v; };

  int lo1() const { return low1;  };
  int hi1() const { return high1; };
  int lo2() const { return low2;  };
  int hi2() const { return high2; };

  double& operator()(const int i, const int j);
  double  operator()(const int i, const int j) const;

};

matrix::matrix(const int h1, const int l1, const int h2, const int l2,
               const double d) : low1(l1), high1(h1), low2(l2), high2(h2)
{
  int size1(h1-l1+1);
  if (size1 < 1) throw "falscher Matrix-Zeilenindex in Konstruktor";

  int size2(h2-l2+1);
  if (size2 < 1) throw "falscher Matrix-Spaltenindex in Konstruktor";

  v = new double[size1*size2];
  if (v == 0) throw "kein freier Speicherplatz mehr verfügbar";

  for (int j=l1; j <= h1; j++)
    for (int k=l2; k <= h2; k++)
      v[(j-l1)+(k-l2)*size1] = d;
};

double& matrix::operator()(const int i, const int j)
{
  if ( (i < low1) || (i > high1)) throw "Indexverletzung bei Komponentenzugriff"
  if ( (j < low2) || (j > high2)) throw "Indexverletzung bei Komponentenzugriff"
  int size1(high1-low1+1);

  return v[(i-low1)+(j-low2)*size1];
};

double  matrix::operator()(const int i, const int j) const
{
  if ( (i < low1) || (i > high1)) throw "Indexverletzung bei Komponentenzugriff"
  if ( (j < low2) || (j > high2)) throw "Indexverletzung bei Komponentenzugriff"
  int size1(high1-low1+1);

  return v[(i-low1)+(j-low2)*size1];
};
```

```
int main()
{
  matrix m1(5,2, 7,0);

  for (int i=m1.lo1(); i <= m1.hi1(); i++)
    for (int j=m1.lo2(); j <= m1.hi2(); j++)
      cout << i << "  " << j << " :       " << m1(i,j) << endl;

  return 0;
}
```

wird eine (high2-low2+1)-fache Zusammenfügung der gleich langen, mit gleichem Indexlaufbereich versehenen Spaltenvektoren durchgeführt.

Testen Sie das Programm (auch auf Zugriffe und Speicherlecks).

Wie ist es zu ändern, wenn die Matrix nicht — wie oben — spaltenweise, sondern zeilenweise abgepeichert werden soll? Testen Sie auch hier.

Aufgabe 3. *Matrizen als Klasse*

Konzipieren Sie in UML-Diagrammen die Ihnen nötig erscheinenden weiteren Konstruktoren (z.B. den Kopierkonstruktor, ...), Observatoren, Modifikatoren (z.B. `matrix operator=(const matrix&)`, ...) und Iteratoren. Welche zusätzlichen Funktionen sind sinnvoll (z.B. `bool operator==(const matrix&) const`, `friend ostream& operator<<(ostream& os, const matrix& m)`, Zusammenfügen von Matrizen mit Matrizen, von Matrizen mit Vektoren, ...)?

Aufgabe 4. *Matrizen als Klasse, Forts.: Implementierung*

Schreiben und testen Sie die in Aufgabe 3 konzipierten Operationen. Beachten Sie dabei insbesondere, wann `delete []v` nötig ist, daß auch in Methoden der sichere Komponentenzugriff durch den operator `()` verwendet werden sollte, ...

Aufgabe 5. *Matrizen und Vektoren: Multiplikation*

Schreiben Sie Multiplikationsoperatoren

`matrix operator*(const matrix&, const matrix&)`

`vektor operator*(const matrix&, const vektor&)`

`matrix operator||(const vektor& x, const vektor& y)  //` $x \cdot y^t$

und testen Sie. Dokumentieren Sie dabei gut die nötigen Kompatibilitätsregeln und benuten Sie in der Implementierung Exceptions bei deren Verletzung.

Aufgabe 6. *Ausblick: Einfach gekettete Listen*

Testen Sie das folgende Programm und versuchen Sie, es mittels zeilenweiser Ausführung zu verstehen:

```
//////////////////////////////////
// Datei:   linkedList.cc
// Version: 1.0
// Autor:   Hans-Juergen Buhl
// Datum:   10.02.199
```



```
//////////////////////////////////

#include <iostream>
#include <string>
#include <assert.h>

#include <stdlib.h>

//using namespace std;

template<class T>
class linkedList{

  struct Knoten{
    T Daten;
    Knoten *next;
    Knoten(const T& daten, Knoten *p):
        Daten(daten), next(p) {};
  } *Anfang;

  void clear_all(Knoten* abStelle){
    if (abStelle != 0) {
      clear_all(abStelle->next);
      delete abStelle;
    };
  };

public:

  linkedList<T>(): Anfang(0) {};

  ~linkedList<T>() { clear_all(Anfang); };

  void push_front(const T& daten){
    Knoten* temp = new Knoten(daten, Anfang);
    assert(temp != 0);
    Anfang = temp;
  };

  void pop_front() {
    if (Anfang != 0) {
      Knoten* temp(Anfang);
      Anfang = Anfang->next;
      delete temp;
    };

  }

  class iterator;
```

```
  friend class iterator;

  class iterator {
    Knoten* aktuell;
  public:
    iterator(Knoten* init = 0): aktuell(init) {};
    T& operator*() { return aktuell->Daten; };
    const T& operator*() const {return aktuell->Daten; };
    iterator& operator++(){ // praefix
      if (aktuell != 0) aktuell = aktuell->next;
      return *this;
    };
    iterator operator++(int){ // postfix
      iterator temp = *this;
      ++*this;
      return temp;
    };
    bool operator!=(const iterator& x) const {
      return (aktuell != x.aktuell);
    };
  };

  iterator begin() const { return iterator(Anfang); };
  iterator end() const   { return iterator(); };
};

int main()
{
 {
  linkedList<string> stringList;
  stringList.push_front("Erster eingefuegter Listenknoten");
  stringList.push_front("Zweiter eingefuegter Listenknoten");
  stringList.pop_front();
  stringList.push_front("Dritter eingefuegter Listernknoten");
  // ...
  stringList.push_front("ENDE!");

  linkedList<string>::iterator pos(stringList.begin());
  for (; pos != stringList.end(); pos++)
    cout << *pos << endl;

 }

 return 0;
}
```

Einfach gekettete Listen sind der erste Prototyp dynamischer „Container-Datentypen“, die wir ausführlich in der *Informatik III* besprechen werden.

Nicht immer braucht man diese „Container-Datentypen“ jedoch selbst mühsam zu

programmieren; die STL (C++ standard template library) stellt eine Grundmenge zur Verfügung:

```
/////////////////////////////////
// Datei:   STLlist.cc
// Version: 1.0
// Autor:   Hans-Juergen Buhl
// Datum:   10.02.1999
/////////////////////////////////

#include <iostream>
#include <string>
#include <assert.h>
#include <list>

//using namespace std;

int main()
{
 {
  list<string> stringList;
  stringList.push_front("Erster eingefuegter Listenknoten");
  stringList.push_front("Zweiter eingefuegter Listenknoten");
  // ...
  stringList.push_front("ENDE!");

  list<string>::iterator pos(stringList.begin());
  for (; pos != stringList.end(); pos++)
    cout << *pos << endl;
 };
 {
  typedef list<double> rliste;
  rliste rl;
  rl.push_front(1.1);
  rl.push_front(1.2);
  rl.push_front(1.3);
  rl.push_front(1.4);
  rl.push_front(1.5);
  rliste::iterator pos(rl.begin());
  for(; pos != rl.end(); pos++)
    cout << *pos << endl;
 };

  return 0;
}
```