Referenzen als Funktionswerte

Um das Überladen des Indexoperators `[]`, der auf beiden Seiten einer Zuweisung vorkommen kann, zu ermöglichen, wurden Referenzen als Funktionswerte eingeführt. Solche Funktionswerte können sinnvoll auf der *linken* Seite einer Zuweisung verwendet werden.

Bsp.: Komponentenfunktion `front()` *für den selbstdef. Datentyp* `Vektor`

```
                 :

class Vektor {
  private:
    double *ap;
    int    len;

  public:
                 :

    double& front()       { return ap[0]; }
    double  front() const { return ap[0]; }

};

int main()
{
  int n;
  cout << "n: "; cin >> n;
  Vektor a(n);    // alle Komponenten haben den Wert 0
  a.front() = 5;  // Komponente 0 wird auf den Wert 5 gesetzt
  cout << "a.front()=" << a.front() << endl;
  return 0;
}
```

Überladen mit Komponentenfunktionen

Operatoren können auch mit Komponentenfunktionen überladen werden, dabei wird $x@y$ in x`.operator@`(y) und $@x$ in x`.operator@`$()$ umgesetzt.

Die Operatoren `[]`, `()`, `=` und `->` können nur durch Komponentenfunktionen überladen werden.

Komponentenfunktionen sind automatisch `inline`-Funktionen.

Bei mehreren Operatorfunktionen erfolgt die Auswahl der verwendeten Funktion nach den Regeln für die Auswahl gleichnamiger Funktionen. Nichtkonstante Komponentenfunktionen für eine Klasse T werden so behandelt, als ob die Parameterliste am Anfang um einen weiteren Parameter vom Typ T`&` ergänzt wäre.

Konstante Komponentenfunktionen werden durch das Attribut `const` gekennzeichnet, das direkt auf die Parameterliste folgt. Sie dürfen im aufgerufenen Objekt keine Komponenten verändern. Die Auswahl konstanter Komponentenfunktionen erfolgt wie bei nichtkonstanten Funktionen, nur daß der hypothetische erste Parameter vom Typ `const` T`&` ist.

Bsp: Vektoren v$\in \mathbb{R}^3$, *Komponentenzugriff über v.x,v.y,v.z und v[1],v[2],v[3]*

```
#include <iostream>

using namespace std;

class Vektor3d {
  public:
    double x,y,z;
    Vektor3d() : x(0),y(0),z(0) {}

    double operator[] (int i) const {
      switch(i) {
        case  1: return x;
        case  2: return y;
        case  3: return z;
      }
    }
    double& operator[] (int i) {
      switch(i) {
        case  1: return x;
        case  2: return y;
        case  3: return z;
      }
    }
};
```

Bsp.: Indexoperator für die Klasse `Vektor`

```
class Vektor {
  private:
    double *ap;
    int    len;
  public:
    double& operator[](int i)       { return ap[i]; }
    double  operator[](int i) const { return ap[i]; }
                           :
};
```

Bsp.: Vektoren mit Indexbereich m..n

```
#include <iostream>
#include <new>
#include <string>

using namespace std;

class fvektor {
  private:
    int m;      // min. Index
    int n;      // max. Index
    double *ap; // Zeiger auf die Komponenten
```

```
  public:
    fvektor(int n_=1, int m_=1, double x=0.0) : m(m_), n(n_) // Standardkonstr.def.
      { ap = new double[n-m+1];
        for (int i=m; i<=n; ++i)
          ap[i-m]=x; }

    fvektor(const fvektor &b);                      // Kopierkonstruktordeklaration
    fvektor operator=(const fvektor& b);            // Zuweisungsoperatordeklaration

    ~fvektor()                                      // Destruktordefinition
      { delete[] ap; }

    double operator[] (int i) const
      { return ap[i-m]; }
    double& operator[] (int i)
      { return ap[i-m]; }

    void ausgeben(string s);                        // Komponentenfunktionsdeklaration
};

fvektor::fvektor(const fvektor &b)                  // Kopierkonstruktordefinition
{                                                   // ausserhalb der Klasse fvektor
  m = b.m; n = b.n;
  ap = new double[n-m+1];
  for (int i=m; i<=n; ++i)
    ap[i-m] = b.ap[i-m];
}

fvektor fvektor::operator=(const fvektor &b)        // Zuweisungsoperatordefinition
{                                                   // ausserhalb der Klasse fvektor
  delete[] ap;
  m = b.m; n = b.n;
  ap = new double[n-m+1];
  for (int i=m; i<=n; ++i)
    ap[i-m] = b.ap[i-m];
  return *this;
}

void fvektor::ausgeben(string s)                    // Komponentenfunktionsdefinition
{                                                   // ausserhalb der Klasse fvektor
  for (int i=m; i<=n; ++i)
    cout << s << "[" << i << "]=" << ap[i-m] << " ";
  cout << endl;
}

int main()
{
  fvektor a(3), b(4,0), c(4,1,5.0), d(c),e;
  a[2]=b[2]=c[2]=1.0;
  a.ausgeben("a");
  b.ausgeben("b");
```

```
  c.ausgeben("c");
  d.ausgeben("d");
  e.ausgeben("e");
  e=c;
  e.ausgeben("e");
  return 0;
}
```

Ausgabe:

```
a[1]=0 a[2]=1 a[3]=0
b[0]=0 b[1]=0 b[2]=1 b[3]=0 b[4]=0
c[1]=5 c[2]=1 c[3]=5 c[4]=5
d[1]=5 d[2]=5 d[3]=5 d[4]=5
e[1]=0
e[1]=5 e[2]=1 e[3]=5 e[4]=5
```

Überladen des Funktionsaufrufs

Der Funktionsaufrufoperator `()` kann durch die Komponentenfkt. `operator()`$(x_1,x_2,\ldots,x_k)$ überladen werden, d.h. für ein Klassenobjekt f bewirkt $f(x_1,x_2,\ldots,x_k)$ den Aufruf f`.operator()`$(x_1,\ldots,x_k)$.

Bsp.: Funktionswerte für Polynome

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

using namespace std;

class Polynom {
  private:
    vector<double> a;

  public:
    Polynom() { }                                // Nullpolynom (Laenge 0, Grad:=-1)
    Polynom(int n): a(n+1) { a[n]=1; }           // Monom x^n
    Polynom(const vector<double>& v): a(v) { }  // Polynom initial. entspr. Vektor v

    int grad() const { return a.size()-1; }      // const-Attribut ergaenzt

    double operator()(double x) const            // geaendert:  value -> operator()
    {                                            //             const-Attribut ergaenzt
      double s=0, xpot=1;
      for (int i=0; i<a.size(); ++i) { s += a[i]*xpot; xpot *= x; }
      return s;
    }
};

int main()
{
  double x; Polynom p;
        :
  cout << "p(x) = " << p(x) << endl;             // geaendert:   p.value(x) -> p(x)
}
```