Überladen von Funktionen

Voreinstellungen für Parameter

Ein mit Gleichheitszeichen zugewiesener Ausdruck in der Parameterliste einer Funktionsvereinbarung bewirkt eine Voreinstellung für den Fall, dass für den entsprechenden Parameter beim Funktionsaufruf kein Argument angegeben wird.

Alle evtl. folgenden Parameter müssen dann ebenfalls Voreinstellungen besitzen (oder in Funktionsvereinbarungen davor mit Voreinstellungen versehen worden sein).

Bsp.: Wert einer Zahlzeichenkette in der Basis $B \in 2, 3, \ldots, 36$ *(Voreinst.:* $B = 10$*)*

```
#include <iostream>
#include <iomanip>
#include <cctype>
#include <string>

using namespace std;

int strtoint(string s, unsigned basis=10)
{
  int ziffer,zahl=0;
  if (s.size()==0 || basis<2 || basis>36) return -1;
  for (int i=0; i<s.size(); ++i) {
    if (isdigit(s[i]))
      ziffer = s[i]-'0';
    else if (islower(s[i]))
      ziffer = s[i]-'a'+10;
    else if (isupper(s[i]))
      ziffer = s[i]-'A'+10;
    else
      return -1;
    if (ziffer<0 || ziffer>=basis) return -1; // Unzulaessiges Ziffer
    zahl = basis*zahl+ziffer;
  }
  return zahl;
}

int main()
{
  string s,t;
  cout << "s: "; cin  >> s;
  cout << "Wert im Dezimalsystem:         " <<  strtoint(s)    << endl;
  cout << "Wert im Hexadezimalsystem:     " <<  strtoint(s,16) << endl;
  return 0;
}
```

Im zugewiesenene Ausdruck dürfen keine anderen Parameter vorkommen.

```
double flaeche(double a, double b=a)   // unzulaessig
{ return a*b; }
```

Parametervoreinstellungen haben auf den Funktionstyp keinen Einfluss.

Operatorfunktionen dürfen keine Voreinstellungen für Parameter enthalten.

Auswahl gleichnamiger Funktionen

Unter den Funktionen im Gültigkeitsbereich des Funktionsaufrufs wird diejenige ausgewählt, deren Parametertypen am besten zu den Argumenttypen passen. Diese muss eindeutig bestimmt sein. Die Argumenttypen müssen sich jeweils implizit in die Parametertypen umwandeln lassen. Dabei werden Standardumwandlungen, gefolgt von benutzerdefinierten Typumwandlungen, wiederum gefolgt von Standardumwandlungen betrachtet.

Die gleichnamigen Funktionen müssen sich hinreichend unterscheiden:
Die Parametertypen `const` T, T und T`&` gelten in diesem Zusammenhang im allg. als gleich. Jedoch werden `const` T`&` und T bzw. `const` T`*` und T`*` jeweils voneinander unterschieden. Funktionen, die sich nur im Ergebnistyp unterscheiden, gelten ebenfalls als gleich.

Standardumwandlungen

Beim Aufruf wird jeweils argumentweise zunächst nach genauer Übereinstimmung von Argumenttyp und Parametertyp gesucht, wozu auch die Umwandlungen $T \to$ `const` T und $T \leftrightarrow T$`&` gehören. Allerdings gelten Folgen von Umwandlungen, bei denen T`&` $\to$ `const` T`&` oder T`*` $\to$ `const` T`*` auftritt, als weniger genau als solche, bei denen diese Umwandlung nicht auftritt.
Dann werden Integererweiterungen und Erweiterungen von Gleitpunktzahltypen versucht, z.B. `char` $\to$ `int` und `float` $\to$ `double`.
Daraufhin werden die verbleibenden impliziten Typumwandlungen eingebauter Datentypen in Betracht gezogen, etwa `int` $\to$ `char`, `double` $\to$ `float` und `int` $\leftrightarrow$ `double`.

Benutzerdefinierte Umwandlungen

Dabei handelt es sich im Zusammenhang mit Klassen definierte Typumwandlungen, pro Argument allerdings nur eine einzige.

Bsp.:

```
typedef complex<double> Complex;
float abs(float)                              // <cmath>
double abs(double);                           // <cmath>     fabs  aus <math.h>
long double abs(long double);                 // <cmath>:
int abs(int);                                 // <cstdlib>   abs   aus <stdlib.h>
long abs(long);                               // <cstdlib>   labs  aus <stdlib.h>
double abs(const Complex&);                   // <complex>
float pow(float, float);                      // <cmath>
float pow(float, int);                        // <cmath>
double pow(double, double);                   // <cmath>     pow   aus <math.h>
double pow(double, int);                      // <cmath>
long double pow(long double, long double);    // <cmath>
long double pow(long double, long int);       // <cmath>
Complex pow(const Complex&, int);             // <complex>
Complex pow(const Complex&, const double&);   // <complex>
Complex pow(const Complex&, const Complex&);  // <complex>
Complex pow(const double&, const Complex&);   // <complex>
//                 :
//  entsprechend fuer complex<float> und complex<long double>
//                 :
```

Überladen von Operatoren

Operatorfunktionen

Durch Funktionen mit dem Namen `operator@` kann der angegebene Operator @ (außer `=` `()` `[]` `? :` `.` `->` `.*` `::`) für nicht eingebaute Datentypen überladen werden. Die Operandenzahl und die Zahl der Funktionsparameter stimmen dabei überein (Ausnahme: Postfixoperatoren `++` und `--`). Der binäre Ausdruck $x@y$ wird in `operator@(`x`,`y`)` und der unäre Ausdruck $@x$ in `operator@(`x`)` umgesetzt.

Bsp.: Überladen der arithmetischen Operatoren in $\mathbb{R}^3$

```
#include <iostream>

using namespace std;

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

Vektor3d operator+(const Vektor3d& a, const Vektor3d& b)
{
  Vektor3d c;
  c.x = a.x+b.x;
  c.y = a.y+b.y;
  c.z = a.z+b.z;
  return c;
}

Vektor3d& operator+=(Vektor3d& a, const Vektor3d& b)
{
  a.x += b.x;
  a.y += b.y;
  a.z += b.z;
  return a;
}

Vektor3d& operator*=(Vektor3d& a, double lambda)
{
  a.x *= lambda;
  a.y *= lambda;
  a.z *= lambda;
  return a;
}

Vektor3d operator*(const Vektor3d& a, double lambda)
{
  Vektor3d b(a);
  return b*=lambda;
}
```

```
Vektor3d operator*(double lambda, const Vektor3d& a)
{
  Vektor3d b;
  b.x = a.x*lambda;
  b.y = a.y*lambda;
  b.z = a.z*lambda;
  return b;
}

Vektor3d operator^(const Vektor3d& a, const Vektor3d& b)
{
  Vektor3d c;
  c.x = a.y*b.z-a.z*b.y;
  c.y = a.z*b.x-a.x*b.z;
  c.z = a.x*b.y-a.y*b.x;
  return c;
}

ostream& operator<<(ostream& stream, const Vektor3d& a)
{
  stream << "(" << a.x << "," << a.y << "," << a.z << ")";
  return stream;
}

istream& operator>>(istream& stream, Vektor3d& a)
{
  char c1,c2,c3,c4;
  stream >> c1 >> a.x >> c2 >> a.y >> c3 >>  a.z >> c4;
  if (c1!='(' || c2!=',' || c3!=',' || c4!=')')
    stream.setstate(ios::failbit);
  return stream;
}

int main()
{
  Vektor3d a,b,c;
  cout << "a: "; cin >> a;
  cout << "b: "; cin >> b;
  c = a^b; cout << "c = " << c << endl;
  c += (a*=2)^b;
  cout << "a = " << a << endl
       << "b = " << b << endl
       << "c = " << c << endl;
  return 0;
}
```

- Die Referenzrückgabe bei `+=` und `*=` entspricht den in der Standardbibliothek üblichen Definitionen für arithmetische Zuweisungen.
- Generell ist zu beachten, dass Referenzen auf lokale Variablen nicht als Rückgabewerte von Funktionen verwendet werden dürfen.