Parametervoreinstellungen

- Voreinstellung für Parameter ⟹ Parameter bei Aufruf optional
- Immer ganzes Endstück der Parameter mit Voreinstellung
- Keine Erwähnung anderer Parameter
- Nicht für Operatoren

```
int strtoint(string s, int basis=10);

strtoint("123");
strtoint("123", 16);
```

```
int strtoint(string s="", int basis);
```

```
double flaeche(double a, double b =
↪  a);
```

Überladene Funktionen

- Mehrere Funktionen dürfen selben Namen tragen
- Einschränkung: unterschiedliche Listen von Parameter-Typen

Einfache überladene Funktion

overloading_demo.cpp

```cpp
#include <iostream>

using namespace std;

int f(int a, int b) {
  return a + b;
}

int f(double x, double y) {
  return x * y;
}

int main() {
  cout << f(1, 2) << endl;
  cout << f(1.0, 2.0) << endl;
}
```

```
3
2
```

Auswahl überladener Funktionen

- Parameter des Aufrufs werden implizit konvertiert
- Implizite Konvertierung ist Folge beliebig vieler einzelner Konvertierungen
- *Eindeutige* „beste“ Konvertierung wird gewählt
- Keine implizite Konvertierung mit:
  - unnötigen Schritten
  - mehr als einer „benutzerdefinierten“ Konvertierung

Einzelne Konvertierungen in absteigender „Güte“:

-
  - Exakte Übereinstimmung
  - Triviale Konvertierung ($t \leftrightarrow t$`&`, t`[]` $\leftrightarrow t$`*`)
- Triviale Konvertierung mit const (t`*` $\longrightarrow$ `const` t`*`, t`&` $\longrightarrow$ `const` t`&`)
- Numerische Zahlbereichserweiterung
- Standard-Konvertierung (numerisch, Zeiger, `bool`)
- „Benutzerdefinierte“ Konvertierung

Interaktion von Überladung mit Templates

- Aufruf kann erfordern Parameter eines Templates zu bestimmen
- Bestimmung der Parameter *vor* Auswahl überladener Funktion
- Daher keine implizite Typumwandlung obwohl evtl. erwartet

tmpl_overload_f.cpp

```cpp
#include <iostream>
#include <complex>

using namespace std;

int main() {
  complex<double> i{0, 1.0};

  cout << (1 + i) << endl;
}
```

Benutzerdefinierte Konvertierungen

- Konstruktoren mit einem Argument
- Typumwandlungsoperatoren

Benutzerdef. Konvertierungen

```cpp
class Complex {
  Complex(double x_) : x(x_), y(0) {}

  operator complex<double>() {
    return complex<double>{x, y};
  }
};
```

Typumwandlung mit Konstruktor

- Konstruktor `C(T t)` nimmt Teil an impliziter Typumwandlung
- Attribut `explicit` verhindert Teilnahme
- Z.B. `explicit vector(size_type n)` aus STL

```
class Vektor {
  Vektor (int n, double x = 0);
}

int main() {
  Vektor v(3);
  v = 2;
  // v.operator=(2)
  // v.operator=(Vektor(2))
}
```

Typumwandlung mit Operator

- Motivation: Typumwandlung Klasse `C` zu eingebautem Datentyp `T` bisher nicht darstellbar
- Spezielle Syntax: `operator T()` wie Methode
- Expliziter Aufruf für `c` vom Typ `C` mit `static_cast<T>(c)`

```
class Vektor {
  private int len;

  operator int() {
    return len;
  }
}
```

Überladung von Operatoren

- Definition analog Funktionen bzw. Methoden, spezielle Syntax
- Für Operator `a∘b` definiere: `operator∘(`a`, `b`)` oder a`.operator∘(`b`)`
- Für Operator `∘a` bzw. `a∘`: `operator∘(`a`)` oder a`.operator∘()`
- Nur als Methode: `=`, `()`, `[]`, `->`
- Garnicht: `.`, `.*`, `::`, `sizeof`, `?:`
- *Principle Of Least Surprise*
- Annotation `operator∘ const (...)` möglich
- `const` präferiert wo möglich

Vergleichsoperatoren

- Satz von Operatoren `<`, `==`, `!=`, `>`, `>=`, `<=` für eigene Klassen oft nützlich
- Fehlerquelle: Inkonsistenz zwischen Operatoren (z.B. a `<` $b \not\Leftrightarrow b$ `>` `a`)
- Daher in `<utility>` und namespace `rel_ops` vordefinierte templates Annahmen:
  - `==` Äquivalenzrelation
  - `<` strenge Totalordnung auf Äquiv'klassen bzgl. `==`

```cpp
#include <utility>

using namespace rel_ops;

class Complex {
  double x, y;

  friend bool operator<(Complex x,
  ↪  Complex y);
  // damit autom. auch: >, <=, >=
};
```

Inkrementoperatoren

- `++`x übersetzt zu:
  - `operator++(`x`)`
  - x`.operator++()`
- x`++` übersetzt zu:
  - `operator++(`x`, 0)`
  - x`.operator++(0)`

Inkrementoperatoren

```
class Complex {
  public:
    double x, y;
    Complex(double x_, double y_) : x(x_), y(y_) {}

    Complex& operator++() {
      x++;
      return *this;
    }
    Complex operator++(int ignored) {
      return Complex{x++, y};
    }
};
```

Überladung des Funktionsaufrufs

x`(`a_1`, `a_2`, ...)` übersetzt zu x`.operator()(`a_1`, `a_2`, ...)`

Überladener Funktionsaufruf: Polynom

polynom_op.cpp

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

using namespace std;

class Polynom {
  private:
    vector<double> coeff;

  public:
    Polynom(const vector<double>& v) : coeff(v) {}

    double operator()(double x) {
      double s = 0, xpot = 1;
      for (vector<double>::size_type i = 0; i < coeff.size();
      ↪  i++) {
        s += coeff[i] * xpot;
        xpot *= x;
      }
      return s;
    }
};

int main() {
  Polynom p{vector<double>{1, 2, 3}};

  cout << "p(x) = " << p(2) << endl;

  return 0;
}
```

```
p(x) = 17
```

Überladener Funktionsaufruf: Matrix

matrix_op.cpp

```cpp
#include <iostream>
#include <iomanip>
#include <valarray>

using namespace std;

class Matrix {
  private:
    int m, n; // Dimension
    valarray<double> a; // Elemente

  public:
    Matrix (int m_ = 0, int n_ = 0, double x = 0)
      : m(m_), n(n_), a(x, m_ * n_) {}

    double operator()(int i, int j) const {
      return a[i*n + j];
    }
    double& operator()(int i, int j) {
      return a[i*n + j];
    }
};

int main() {
  const int m = 4, n = 3;

  Matrix a{m, n};
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      a(i, j) = i*n + j;

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      cout << setw(4) << a(i, j);
    }
    cout << endl;
  }

  cout << endl;

  const Matrix b{1, 1, 42};
  cout << b(0, 0) << endl;
}
```

```
   0   1   2
   3   4   5
   6   7   8
   9  10  11

42
```

Überladener Funktionsaufruf: Normalverteilung

normv_op.cpp

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

class NormV {
  private:
    double mw, stdabw;

  public:
    NormV (double mw_ = 0, double stdabw_ = 1)
      : mw(mw_), stdabw(stdabw_) {}

    double operator()(double x) const {
      return M_2_SQRTPI / (2 * M_SQRT2 * stdabw)
        * exp(-((x - mw) * (x - mw)) / (2*stdabw*stdabw));
    }
};

int main() {
  NormV n{0, 2};
  for (double x = -6; x <= 6.001; x += 12. / 18) {
    double fx = n(x);
    cout << setw(5) << fixed << setprecision(3) << fx << ' ';
    for (int k = 1; k < fx * 200; k++) cout << '#';
    cout << endl;
  }

  cout << endl;

  cout << defaultfloat << setprecision(6)
       << NormV{}(-0.5) << endl;
}
```

```
0.002
0.006 #
0.013 ##
0.027 #####
0.050 #########
0.082 ################
0.121 ########################
0.160 ###############################
0.189 #####################################
0.199 #######################################
0.189 #####################################
0.160 ###############################
0.121 ########################
0.082 ################
0.050 #########
0.027 #####
0.013 ##
0.006 #
0.002

0.352065
```

Funktionsobjekte

Funktionsobjekt ist Objekt einer Klasse, bei der:

- `operator()` überladen
- Aufruf von `operator()` „Primärzweck“ der Klasse