Parametervoreinstellungen

Bei der Vereinbarung von Funktionen kann ein Endstück der Parameterliste mit Voreinstellungen versehen werden. D.h. den Paramatern werden, abgetrennt jeweils durch =, in der Parameterliste Werte zugewiesen. Dies bewirkt, dass beim späteren Aufruf der Funktion jene Parameter, die Voreinstellungen haben, optional weggelassen werden dürfen. In diesem Fall wird für die weggelassenen Parameter automatisch die Voreinstellung eingesetzt.

Beispiel. Wir definieren eine Funktion `strtoint` die einen `string` von Ziffern umwandelt in eine Zahl vom Typ `int`. Die Funktion nimmt optional einen Parameter für die Basis der Zahl (z.B. Dezimal oder Hexadezimal). Die Basis 10, d.h. Dezimal, ist für den `basis`-Parameter voreingestellt.

strtoint.cpp

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

using namespace std;

int strtoint(string s, unsigned int basis = 10) {
  if (s.size() <= 0 || basis < 2 || basis >= 36)
    return -1;

  int zahl = 0;
  for (unsigned int i = 0; i < s.size(); i++) {
    unsigned int ziffer;

    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;

    zahl = basis*zahl + ziffer;
  }

  return zahl;
}

int main() {
  string s;
  cout << "s: ";
  cin >> s;

  cout << "Wert in dezimal:     " << strtoint(s)     << endl
       << "Wert in hexadezimal: " << strtoint(s, 16) << endl;

  return 0;
```

```
}
```

```
s: 42
Wert in dezimal:     42
Wert in hexadezimal: 66
```

Andere Parameter der Funktion stehen im zugewiesenen Ausdruck einer Parametervoreinstellung nicht zur Verfügung.

Bei überladenen Operatoren (z.B. `operator+`) dürfen Parametervoreinstellungen nicht verwendet werden.

Überladene Funktionen

In C++ dürfen mehrere Funktionen den selben Namen tragen. Sie müssen sich dann jedoch in der Anzahl und/oder den Typen ihrer Parameter unterscheiden.

Beispiel. Wir überladen eine Funktion `f` mit `f(int, int)` und `f(double, double)`.

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

Bei Aufruf einer Funktion wird aus den gleichnamigen Funktionen, in deren syntaktischen Gültigkeitsbereich der Aufruf liegt, eine konkrete Implementierung ausgewählt. Hierbei werden alle Parameter des Funktionsaufrufs implizit Konvertiert zu den Typen von Parametern, die die gewählte konkrete Implementierung erwartet.

Es wird stets die Implementierung gewählt für die die „besten“ impliziten Konvertierungen gewählt werden können. Diese muss eindeutig sein.

Implizite Konvertierungen bestehen aus folgen von einzelnen Konvertierungen. Es werden keine impliziten Konvertierungen in Betracht gezogen, die unnötige Schritte enthalten oder in denen mehr als eine benutzerdefinierte Konvertierung vorkommt.

Die „Güte“ einer impliziten Konvertierung ergibt sich grob aus der „Güte“ ihrer einzelnen Konvertierungen.

Einzelne Konvertierungen in absteigender „Güte“:

- – Exakte Übereinstimmung
  – Triviale Konvertierungen ($t \leftrightarrow t\texttt{\&}$, $t\texttt{[]} \leftrightarrow t\texttt{*}$)
- Triviale Konvertierung mit const ($t\texttt{*} \longrightarrow \texttt{const}\ t\texttt{*}$, $t\texttt{\&} \longrightarrow \texttt{const}\ t\texttt{\&}$)
- Numerische Zahlbereichserweiterung (z.B. `float` ⟶ `double`, `bool` ⟶ `int`, `char` ⟶ `int`)
- Standard-Konvertierung (numerisch, Zeiger, `bool`)
- „Benutzerdefinierte“ Konvertierung (Konstruktoren mit einem Argument, Typumwandlungsoperatoren)

Beispiel. `abs` und `pow` liegen in der Standardlibrary überladen vor:

- – `float abs(float)`
  – `double abs(double)`
  – `long double abs(long double)`
  – `int abs(int)`
  – `long int abs(long int)`
  – `double abs(const complex<double>&)`
  – …
- – `float pow(float, float)`
  – `float pow(float, int)`
  – `long double pow(long double, long double)`
  – `long double pow(long double, long int)`
  – `complex<double> pow(const complex<double>&, int)`
  – `complex<double> pow(const complex<double>&, const double&)`
  – `complex<double> pow(const complex<double>&, const complex<double>&)`
  – `complex<double> pow(const double&, const complex<double>&)`
  – …

Benutzerdefinierte Konvertierungen

Es kann für Klassen definiert dass Werte anderer Typen in Objekte der Klasse implizit konvertiertbar sein sollen. Hierfür wird ein Konstruktor akzeptiert der genau einen Parameter des Typs, von dem konvertiert werden soll, als Argument akzeptiert.

Umgekehrt kann definiert werden, dass Objekte der Klasse impliziert konvertierbar sein sollen in Werte anderer Typen (auch eingebauter Typen wie z.B. `int`). Hierfür wird spezielle Syntax verwendet, ein Typumwandlungsoperator für die Klasse definiert.

Typumwandlungen mittels Konstruktoren

Wann immer ein Konstruktor für eine Klasse existiert der sich mit genau einem Argument aufrufen lässt, auch unter Berücksichtigung von Parametervoreinstellungen, wird dieser als benutzerdefinierte Konvertierung für implizierte Konvertierung in Betracht gezogen.

Beispiel. Wir deuten eine eigene Vektor-Klasse an. Wir vereinbaren einen Konstruktor, der eine Angabe der Länge und eine Vorlage für die Elemente eines neuen Vektors als Parameter akzeptiert. Dies führt, evtl. unintuitiv, dazu dass Längen von Vektoren (Werte vom Typ `int`) implizit konvertiert werden zu Vektoren.

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

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

Soll dies nicht geschehen kann der Konstruktor mit `explicit` dekoriert werden. Dies wird in der Standardbibliothek häufig verwendet, z.B. die Konstruktoren der Klasse `vector<T>`.

Typumwandlungen mittels Operatorfunktionen

Insb. Konvertierungen von Objekten einer Klasse in eingebaute Datentypen (z.B. `int`) lassen sich nicht durch Konstruktoren mit einem Argument ausdrücken. Es gibt daher, in Anlehnung an Operatorüberladung, spezielle Syntax um derartige Konvertierungen zu definieren. Für einen Typ T, in den Objekte der Klasse konvertierbar sein sollen, wird `operator` T`()` ohne Rückgabetyp (auch nicht `void`) und ohne Parameter vereinbart. Bei T darf es sich um einen pointer- oder Referenztyp handeln (`*` bzw. `&`).

Beispiel. Wir definieren (rudimentär) eine eigene Klasse `Vektor` und implementieren die Umwandlung von und zu `vector<double>` mittels Operatorfunktion bzw. Konstruktor mit einem Argument.

vector_conversion.cpp

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

using namespace std;

class Vektor {
  private:
```

```
    double* ap;
    int len;

  public:
    Vektor() : ap(0), len(0) {}
    Vektor(const vector<double>& v) : len(v.size()) {
      ap = new double[len];
      for (int i = 0; i < len; i++) ap[i] = v[i];
    }

    ~Vektor() {
      delete[] ap;
    }

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

    operator vector<double>() {
      vector<double> v(len);
      for (int i = 0; i < len; i++) v[i] = ap[i];
      return v;
    }

    void ausgeben(string n) {
      for (int i = 0; i < len; i++)
        cout << n << "[" << i << "]: " << ap[i] << " ";
      cout << endl;
    }
};

int main() {
  vector<double> b(4, 2.0);
  Vektor a = b;
  a.ausgeben("a");

  a[2] = 3;
  vector<double> c = a;
  for (unsigned int i = 0; i < c.size(); i++)
    cout << "c[" << i << "]: " << c[i] << " ";
  cout << endl;

  return 0;
}
```

```
a[0]: 2 a[1]: 2 a[2]: 2 a[3]: 2
c[0]: 2 c[1]: 2 c[2]: 3 c[3]: 2
```

In der Standardbibliothek ist für Ströme ein Typumwandlungsoperator nach `bool` definiert. Für einen Strom *s* liefert die Umwandlung in einen `bool` das selbe Ergebnis wie `!s.fail()`.

Überladen von Operatoren (Forts.)

Beispiel. Wir überladen einige arithmetische Operatoren einer eigenen Klasse zur Modellierung von Vektoren aus $\mathbb{R}^3$. Den Operator ^ verwenden wir zur Darstellung des Kreuzprodukts.

vector3d_arith.cpp

```cpp
#include <iostream>

using namespace std;

class Vektor3D {
  public:
    double x, y, z;
    Vektor3D(double x_ = 0, double y_ = 0, double z_ = 0) : x(x_), y(y_), z(z_) {}
};

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) {
  return a * lambda;
}

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) {
  return stream << "(" << a.x << "," << a.y << "," << a.z << ")";
}

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;
  cout << "a: "; cin >> a;
  cout << "b: "; cin >> b;

  Vektor3D c = a ^ b;
  cout << "c = " << c << endl;

  c += (a *= 2) ^ b;
  cout << "a = " << a << endl
       << "b = " << b << endl
       << "c = " << c << endl;

  return 0;
}
```

```
a: (1,2,3)
b: (3,2,1)
c = (-4,8,-4)
a = (2,4,6)
b = (3,2,1)
c = (-12,24,-12)
```

Überladen von Operatoren als Methoden

Ein Operator $\circ$ kann auch als Methode überladen werden. Binäre Ausdrücke der Form $x \circ y$ werden übersetzt in den Aufruf x`.operator`$\circ$`(`y`)` und unäre Ausdrücke der Form $\circ x$ in x`.operator`$\circ$`()`.

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

Wird zwischen nicht als Methoden überladenen Operatoren und als Methoden überladenen Operatoren ausgewählt (vgl. [auswahl-uxfcberladener-funktionen]) so werden die Implementierungen, bei denen der Operator als Methode überladen wurde, behandelt als hätten sie einen zusätzlichen ersten Parameter vom Typ der Klasse. Bei nicht-konstanten Methoden (kein `const`) wird für eine Klasse T agiert als gäbe es einen zusätzlichen ersten Parameter vom Typ T`&`; bei konstanten Methoden (mit `const`) stattdessen `const` T`&`.

Beispiel. Wir definieren und implementieren Überladungen für den `[]` Operator für eine eigene Klasse zur Modellierung von Vektoren aus $\mathbb{R}^3$, sodass der Zugriff auf die Komponenten eines Vektors x geschrieben werden kann als x`[1]`, x`[2]` bzw. x`[3]`.

Um sowohl schreibenden Zugriff auf die Komponenten von nicht-`const` Objekten, wie auch lesenden Zugriff auf `const` Objekte zu ermöglichen definieren wir sowohl eine nicht-`const`, wie auch eine `const` Variante von `operator[]`. Es muss zudem auf den Typ des Rückgabewerts der nicht-`const` Variante geachtet werden.

```
class Vektor3D {
  public:
    double x, y, z;

    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;
      }
    }
};
```

Beispiel. Wir definieren eine eigene `Vektor`-Klasse die zusätzlich unterstützt beim Aufruf des Konstruktors einen eigenen Indexbereich $[m, n) \subset \mathbb{N}$ anzugeben.

ivector.cpp

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

using namespace std;

class IVektor {
  private:
    int m, n;
    double* ap;

  public:
    IVektor(int n_ = 0, int m_ = 0, double x = 0.0) : m(m_), n(n_) {
      ap = new double[n - m];
      for (int i = m; i < n; i++)
        ap[i - m] = x;
    }
    IVektor(const IVektor& v) : m(v.m), n(v.n) {
      ap = new double[n - m];
      for (int i = m; i < n; i++)
        ap[i - m] = v.ap[i - m];
    }
    ~IVektor() {
      delete[] ap;
    }

    IVektor& operator=(const IVektor& v) {
```

```
      delete[] ap;

      m = v.m; n = v.n;
      ap = new double[n - m];
      for (int i = m; i < n; i++)
        ap[i - m] = v.ap[i - m];
      return *this;
    }

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

    void ausgeben(string name) {
      for (int i = m; i < n; i++)
        cout << name << "[" << i << "]=" << ap[i - m] << " ";
      cout << endl;
    }
};

int main() {
  IVektor a{3, 1}, b{4}, 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;
}
```

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

e[1]=5 e[2]=1 e[3]=5
```

Vergleichsoperatoren

Um die Vergleichsoperatorn (<, >, <=, >=, ==, !=) zu überladen bietet die Standardlibrary in <utility> und namespace rel_ops vordefinierte Implementierungen, sodass es bei Verwendung genügt nur < und == zu überladen.

Beispiel. Wir definieren auf einer (rudimentären) eigenen Vektor-Klasse die lexikographische Ordnung.

vector_ordering.cpp

```cpp
#include <iostream>
#include <utility>

using namespace std;
using namespace rel_ops;

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

  public:
    Vektor(int len_ = 0, double x = 0.0) : len(len_) {
      ap = new double[len];
      for (int i = 0; i < len; i++) ap[i] = x;
    }
    ~Vektor() {
      delete[] ap;
    }
    double& operator[](int i) {
      return ap[i];
    }
    double operator[](int i) const {
      return ap[i];
    }

    friend bool operator== (const Vektor& v1, const Vektor& v2) {
      if (v1.len != v2.len) return false;
      for (int i = 0; i < v1.len; i++)
        if (v1[i] != v2[i])
          return false;
      return true;
    }

    friend bool operator< (const Vektor& v1, const Vektor& v2) {
      for (int i = 0; i < v1.len; i++) {
        if (v1[i] > v2[i])
          return false;
        else if (v1[i] < v2[i])
          return true;
      }
      return v1.len < v2.len;
    }
};

int main() {
  Vektor v1{2, 1.0}, v2{3, 1.0};

  cout << boolalpha;

  cout << "v1 < v2:  " << (v1 < v2) << endl
       << "v1 > v2:  " << (v1 > v2) << endl
       << "v1 == v2: " << (v1 == v2) << endl
       << "v1 != v2: " << (v1 != v2) << endl;
  cout << endl;
```

```
  v1[1] = 2.0;
  cout << "v1 < v2:  " << (v1 < v2) << endl
       << "v1 > v2:  " << (v1 > v2) << endl
       << "v1 == v2: " << (v1 == v2) << endl
       << "v1 != v2: " << (v1 != v2) << endl;
  cout << endl;

  Vektor v3{3, 1.0};
  cout << "v2 <= v3: " << (v2 <= v3) << endl
       << "v2 >= v3: " << (v2 >= v3) << endl
       << "v2 == v3: " << (v2 == v3) << endl
       << "v2 != v3: " << (v2 != v3) << endl;

  return 0;
}
```

```
v1 < v2:  true
v1 > v2:  false
v1 == v2: false
v1 != v2: true

v1 < v2:  false
v1 > v2:  true
v1 == v2: false
v1 != v2: true

v2 <= v3: true
v2 >= v3: true
v2 == v3: true
v2 != v3: false
```

Inkrement- und Dekrementoperatoren

Auf die Inkrement- und Dekrementoperatoren `++` und `--` können, sowohl in ihrer Präfix- wie auch Postfix-Form, überladen werden. Um Präfix- von Postfix-Variante unterscheiden zu können erhalten die Postfix-Varianten der Operatoren künstlich einen zusätzlichen Parameter vom Typ `int`. Beim Aufruf erhält dieser stets den Wert 0.

Beispiel. Wir definieren Präfix und Postfix-Varianten der Inkrement- und Dekrementoperatoren für eine eigene Klasse `Complex`.

complex_increment.cpp

```
#include <iostream>

using namespace std;

class Complex {
  private:
    double re, im;
```

```
  public:
    Complex (double re_ = 0, double im_ = 0) : re(re_), im(im_) {}

    double real() const { return re; }
    double imag() const { return im; }

    friend Complex& operator++(Complex& z) {
      z.re++;
      return z;
    }
    friend Complex operator++(Complex& z, int ignored) {
      return Complex{z.re++, z.im};
    }

    friend Complex& operator--(Complex& z) {
      z.re--;
      return z;
    }
    friend Complex operator--(Complex& z, int ignored) {
      return Complex{z.re--, z.im};
    }
};

int main() {
  Complex z{2.0, 3.0};

  cout << (z++).real() << endl
       << z.real()     << endl
       << (++z).real() << endl
       << z.real()     << endl;
  cout << endl;

  cout << (z--).real() << endl
       << z.real()     << endl
       << (--z).real() << endl
       << z.real()     << endl;

  return 0;
}
```

```
2
3
4
4

4
3
2
2
```

Überladen des Funktionsaufrufs

Für ein Objekt x vom Typ T können durch Definition und Implementierung einer Methode `operator()`$(a_1, a_2, \ldots)$ in T Funktionsaufrufe der Form $x(a_1, a_2, \ldots)$ ermöglicht werden.

Beispiel. Wir überladen den Funktionsaufruf für eine Klasse `Polynom` zur Modellierung von Polynomen über $\mathbb{R}$, sodass für ein Objekt p der Klasse `Polynom` der Aufruf $p(x)$ den Wert des zugehörigen Polynoms an der Stelle x liefert.

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
```

Beispiel. Wir definieren eine eigene Klasse `Matrix` zur Modellierung von $m \times n$-Matrizen über $\mathbb{R}$. Die Elemente der Matrix werden innerhalb des Objekts in einer listenartigen Datenstrukter `valarray` gespeichert. Es werden hierfür die Zeilen der Matrix einfach konkateniert.

Wir überladen behelfsweise die Funktionsauswertung für Objekte vom Typ `Matrix` um Zugriff zu erhalten auf die Elemente der Matrix anhand ihrer Zeile und Spalte.

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
```

Funktionsobjekte

Objekte einer Klasse für die der Funktionsaufruf überladen wurde, werden auch als *Funktionsobjekte* bezeichnet. Die Bezeichnung wird vor Allem dann verwendet wenn der Hauptzweck der Klasse darin besteht dass die Objekte funktionsartige Aufrufe bereitstellen.

Besonders nützlich sind Funktionsobjekte zur Darstellung von Funktionen deren Wert von zusätzlichen Parametern abhängt, die nicht bei jeder Auswertung angegeben sollen. Insb. ist die Manipulation der Parameter der Funktionsobjekte (z.B. eine algebraische Struktur wie Addition/Multiplikation auf den Objekten anzubieten) oft Zweck der Modellierung als Funktionsobjekt.

Beispiel. Wir definieren eine Klasse `NormV` zur Darstellung von Normalverteilungen. Eine Normalverteilung ist durch ihren Mittelwert und ihre Standardabweichung bereits vollständig spezifiziert. Die Funktionsauswertung eines Objekts der Klasse `NormV` liefert den Wert der Wahrscheinlichkeitsdichtefunktion an der gegebenen Stelle.

Es wäre durchaus denkbar Additionsoperatoren für die Klasse zu implementieren. Der Effekt könnte in diesem Fall äquivalent dazu sein die Wahrscheinlichkeitsdichtefunktionen der gegebenen Objekte erst konventionell zu addieren und dann zu normieren, da die Familie von Normalverteilungen unter dieser Addition abgeschlossen ist.

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
```