Funktions-Templates

Zum Vertauschen zweier Ganzzahlen definierten wir die Funktion

```
void swap(int& i, int& j)
     { int h = i; i = j; j = h; return; }
```

Mit einem *Funktions-Template* (einer "Funktions-Schablone") kann man jetzt vom Typ **int** abstrahieren:

```
template <typename Irgendwas>
void swap(Irgendwas& i, Irgendwas& j)
     { Irgendwas h = i; i = j; j = h;
                                return; }
```


Zu beachten:

- Statt dem Schlüsselwort **typename** kann das veraltete (und verwirrende!) **class** verwendet werden.

Mögliche Aufrufe sind nun:

```
int   i=1, j=2;  swap(i, j);
float a=1, b=2;  swap(a, b);
Punkt A(3,4), B; swap(A, B);
```

Bemerkung: die Standardbibliothek enthält bereits ein solches Funktions-Template **swap**.
(Man unterscheide also **std::swap** und **::swap**).

Klassen-Templates

Statt einer einzelnen Funktion kann auch eine ganze Klasse mit einem Datentyp parametrisiert werden. Dazu wird ein *Klassen-Template* benutzt.

Eine mathematische Anwendung wäre ein Klassen-Template **Polynom** für Polynome mit Koeffizienten aus beliebigen Datentypen.

Die Idee ist, dass hier **float** oder **int** ebenso verwendet werden können wie **Bruch** oder wie eine Klasse für komplexe Zahlen aus der Standardbibliothek.

Vorausgesetzt wird nur, dass die verwendete Klasse bestimmte Operationen besitzt, etwa **+ - * /** für das Beispiel **Polynom**.

Als einfaches Beispiel soll nun die Klasse **Bruch** zu einem Klassen-Template erweitert werden, das z.B. **short** oder **unsigned int** für Zähler und Nenner zulässt.

Die *Deklaration* geschieht wie beim Funktions-Template:

```
template <typename T>
class Bruch {
     T zaehler, nenner;
   public:
     Bruch(T z=0, T n=1);
     double wert();
     //…
};
```


Ein syntaktischer Unterschied ergibt sich aber beim *Gebrauch* der beiden Arten von Templates:

- Beim Funktions-Template wird der Typ auf Grund der aktuellen Parameter automatisch erkannt.
- Beim Klassen-Template geht dies nicht. Hier muss der Typ explizit angegeben werden.
  Beispiel:

```
Bruch<short> a, b(3), c(15,4);
Bruch<short> d = a;
```


Komplexe Zahlen

In ähnlicher Weise funktioniert ein Klassen-Template für komplexe Zahlen aus der Standardbibliothek.

Es unterstützt komplexe Zahlen über **float**, **double**, **long double** und **int**.

Beispiel:

```
#include <complex>
complex<float> i(0,1), n(-1);
cout << i*i     << endl;      // -1
cout << sqrt(n) << endl;      // +-i
```

Arithmetik-Operatoren: **+ - * /**

Vergleichs-Operatoren: **== !=**

Ein-/Ausgabeoperatoren: **>> <<**

Elementfunktionen:

```
real()  // Realteil
imag()  // Imaginaerteil
norm()  // Betragsquadrat
abs()   // Betrag
arg()   // Argument
polar() // Polarform
```

Unterstützte Standardfunktionen:

```
sin() cos() tan() sinh() cosh() tanh()
sqrt() exp() log() log10()
```

Konstruktor, Zuweisungs- und Eingabeoperator akzeptieren auch *reelle* Zahlen.

Anwendungsbeispiel: Mandelbrot-Menge

```
#include <complex>
#include <ifmwindow>
using namespace std;

int main()
{
  const int    nMax     = 150;
  const double maxNorm  = 1e2;
  const int    width    = wio.xmax() - wio.xmin();
  const int    height   = wio.ymax() - wio.ymin();
  const int    maxColor = wio.number_of_colors()-2;

  double x0 = -2., y0 = -2., h = 4./width;
```



```
while (true) {
  int col, row, n;
  for (row = 0; row <= height; row++) {
    for (col = 0; col <= width; col++) {
      complex<double> c(x0+col*h, y0+row*h),
                      z(0,0);

      for (n=0; n<nMax && norm(z)<maxNorm; n++) {
           z = z*z+c;
      }

      wio << color(n*maxColor / nMax)
          << Point(col,row);
    }
    wio << flush;
  }
```



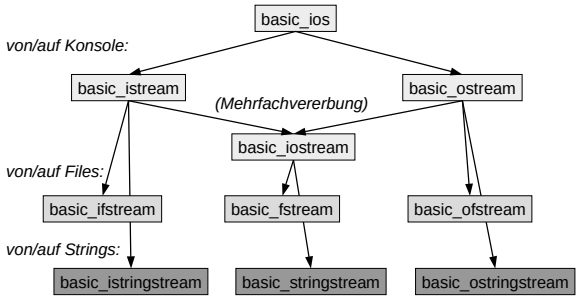
```
    while (!wio.check_key() &&
        !wio.check_mouse_click());// wait for event
    if (wio.check_key()) break;

    int button = wio.get_mouse_click(col, row);

    switch (button) {
          // L: zoom in, M: set center, R: zoom out
      case 1: h /= 2.;  x0 += col*h;
                        y0 += row*h; break;
      case 2: x0 -= (width /2.-col)*h;
              y0 -= (height/2.-row)*h; break;
      case 3: x0 -= col*h;
              y0 -= row*h;  h *= 2.; break;
    }
  }
  return 0;
}
```


Beispiel: Ein-/Ausgabeströme

Die Standardbibliothek definert die folgende Hierarchie von (Template-) Klassen für Text-Ein/Ausgabe:



Alle Template-Klassen **basic_X** können für die beiden Datentypen **char** und **wchar_t** benutzt werden.

Dazu wurden die Abkürzungen definiert:

```
typedef basic_X<char>    X;
typedef basic_X<wchar_t> wX;
```

Also z.B.:

```
typedef basic_istream<char>    istream;
typedef basic_istream<wchar_t> wistream;
```

Die bereits bekannten **cin, cout, cerr** sind nun Variablen mit den Definitionen:

```
istream cin;
ostream cout, cerr;
wistream wcin;
wostream wcout, wcerr;
```

Analog können wir jetzt z.B. einen Strom für Texteingabe von einem File definieren:

```
fstream fin;
```

Die Operatoren und Methoden **>>, getline** etc. stehen zur Verfügung nachdem man das File "geöffnet" hat:

```
fin.open("meinText.txt", ios::in);
```



```
// Zaehle Woerter in Textfile
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
  ifstream fin;
  fin.open("countWords.cpp", ios::in);
  if (!fin.good()) {
     cerr << "File not readable." << endl;
     return 1;
  }
  int count = 0;
  while (!fin.eof()) {
     string s;  fin >> s; count++;
  }
  count--; // abschliessendes 'newline' Zeichen
  cout << count << " words." << endl;
  return 0;
}
```


Numerische Parameter

Zur Parametrisierung einer Klasse kann auch ein *numerischer Wert* verwendet werden.

Beispiel: *Restklassen modulo n.*

```
template <int n>
class Restklasse {
  private:
    int wert;  // Wert der Restklasse
        // Normalisieren (interne Hilfsfkt.)
    Restklasse<n> normal() {
        if (wert %= n < 0) wert += n;
        return *this;
    }
```



```
  public:
        // Konstruktor:
    Restklasse<n> (int W=0) {
        wert=W; normal(); }
        // ueberladene Operatoren:
    Restklasse<n> operator+ <> (
        const Restklasse<n>& b) {
        return Restklasse<n>(wert+b.wert);
    }
        // etc.
        // Ausgabe-Operator:
    friend ostream& operator<< (
        ostream& s, Restklasse<n>&);
};
```


Testprogramm:

```
int main()
{
     Restklasse<10> a(5), b(8), c;
     c = a + b;  cout << c << endl;

     Restklasse<7> d(5), e(8), f;
     f = d + e;  cout << f << endl;

     // f = a + d; // Typen-Fehler!

     return 0;
}
```


Container-Klassen

Eine wichtige Anwendung haben die Klassen-Templates in den sog. *Container-Klassen.*

Container-Klassen sind "Sammelbehälter" für Daten vom gleichen Typ, der aber nicht zum vornherein spezifiziert wird.

Die *Standardbibliothek* bietet Container-Klassen für Mengen, Stacks, Warteschlangen, dynamische Arrays etc., die bis vor kurzem noch in einer Extra-Bibliothek STL *(standard template library)* untergebracht waren.

Einige wichtige Container-Klassen sind:

Klasse	Beschreibung
vector	Wie Array, aber mit dynamischem Vergrössern/Verkleinern. Einfügen/Löschen am Ende in O(1), sonst in O(n).
deque	"double ended queue": Wie **vector**, aber veränderbar an beiden Enden. Einfügen/Löschen am Anfang/Ende in O(1), sonst in O(n).
list	Doppelt verkettete Liste. Einfügen/Löschen an beliebiger Position in O(1).
set	Enthält Elemente immer sortiert und nur je einmal.
stack	Hat nur Methoden **empty()**, **push()**, **pop()**, **top()** und **size()**.
queue	Erlaubt nur Zugriff auf erstes und letztes Element. Löschen nur vorne, Einfügen nur hinten.
priority_queue	Wie **queue**, aber sortiert nach **<**, mit grösstem Element zuvorderst.

Vektoren

Wir betrachten nun die spezielle Container-Klasse **vector**, die als verbesserte Form des Arrays benutzt werden kann.

Einen Vektor für 5 Ganzzahlen erzeugt man z.B. so:

```
#include <vector>
vector<int> v(5);
```

Beispiele mit weiteren Konstruktoren:

```
vector<char> u;           // leerer Vektor
vector<float> v(5, 1.0); // Anzahl und Wert
int a[] = {1,4,9,16,25};
vector<int> w(a, a+5);   // Vektor aus Array
```


Auf die Elemente kann man mit dem überladenen Operator **[]** zugreifen. Beispiele:

```
v[2] = 2.0;
float f = v[j-1];
```

Zum Durchlaufen eines Vektors benutzt man aber besser die sog. *Iteratoren*.

Iteratoren sind Objekte verschiedener in der Header-Datei **#include <iterator>** definierter Klassen.

Dank den überladenen Operatoren * und ++ kann man Iteratoren praktisch wie *Zeiger* auf Vektor-Elemente verwenden.

Vorteil der Iteratoren: Zusammen mit den Methoden **begin()** und **end()** der jeweiligen Container-Klasse ermöglichen sie das sequentielle Durchlaufen des Containers mit einer **for**-Schleife.

Beispiel:

```
#include <vector>
#include <iterator>
vector<int> v = ...;
vector<int>::const_iterator i;
for (i = v.begin(); i != v.end(); i++) {
     cout << (*i) << " ";
}
```


Mit den Methoden **rbegin()** und **rend()** kann auch eine **for**-Schleife in umgekehrter Reihenfolge gebildet werden.

Durch Verwendung von Iteratoren vermeidet man das Überschreiten des Indexbereichs!

Iteratoren gibt es für *alle* Container-Klassen der Standardbibliothek.

Generisches Programmieren

Die Verwendung von Templates erlaubt es, von den Datentypen zu abstrahieren, was auch als *generisches Programmieren* bezeichnet wird.

Die Container-Klassen unterstützen diese Philosophie. Sie erlauben es, bereits implementierte Algorithmen auch für neue Klassen zu verwenden.

Weil die Container-Klassen zu einem grossen Teil identische Methoden und Funktionen haben, ist sogar der Container-Typ relativ leicht nachträglich austauschbar (z.B. gegen einen effizienteren oder einen flexibleren).

Die Container-Klassen **vector**, **deque**, **list** und **set** besitzen (unter anderem) alle die Methoden:

```
• empty()   // Abfrage ob Container leer
• size()    // aktuelle Grösse
• max_size()// maximal mögliche Grösse
• begin()   // liefert Iterator auf erstes El.
• end()     // … auf Position nach letztem El.
• rbegin()  // liefert Rückwärts-Iterator
• rend()    // liefert Rückwärts-Iterator
```


Die Klassen **vector**, **deque** und **list** haben zusätzlich:

```
• assign()     // bei Iterator Element ersetzen
• insert()     // ... Element einfügen
• erase()      // ... Element löschen
• clear()      // ganzen Container leeren
• push_back()  // Element hinten anhängen wobei
               // die Grösse des Containers
               // automatisch angepasst wird,
               // wenn nötig mit Re-Allokation
• pop_back()   // letztes Element löschen
```


Zusätzlich zu den Methoden gibt es auch viele Funktionen für Container-Klassen. Diese werden deklariert mit **#include <algorithm>**.

Eine kleine(!) Auswahl davon ist:

```
for_each()   // Funktionsaufruf mit jedem Element:
             // for_each(c.begin(),c.end(),drucke);
sort()       // sortiert Elemente mit Quicksort
stable_sort()// sortiert, unter Beibehaltung
             // der Reihenfolge bei Gleichheit
find()       // sequentielle Suche
binary_search() // binäre Suche, setzt Sortiertheit
                // voraus
merge()      // Mischen zweier sortierter Container
```


Weitere Bemerkungen:

Die Standardbibliothek bietet leider keine Container-Klasse für *binäre Bäume*.

Dafür existiert ein Container **multimap**, der Paare (Schlüssel, Wert) effizient verwalten kann. Er ist nicht als Baum, sondern als *Hash-Tabelle* implementiert.

Die *sortierten* Container funktionieren für beliebige Klassen, wo der (mit einer linearen Ordnungsrelation) überladene Operator **<** existiert.

Beispiel: Einen Vektor von Zahlen sortieren

```
vector<int> a;
while (…) { … ; a.push_back(…); }
a.sort(a.begin(), a.end());
```

Intern wird der Operator **<** verwendet.

Weil **int** keine Klasse ist, kann dieser nicht überladen werden. Man kann aber als drittes Argument von **sort** eine Vergleichsfunktion angeben:

```
bool myLess(const int& p, const int& q)
              { return p%10 < q%10; }
a.sort(a.begin(), a.end(), myLess);
```


Zum Vergleich: Einen Vektor von **Person**en sortieren

Man kann genau gleich vorgehen:

```
vector<Person> a;
while (…) { … ; a.push_back(…); }
a.sort(a.begin(), a.end());
```

Es wird wiederum der Operator **<** verwendet, ein solcher wird also für **Person** vorausgesetzt.

Dieser kann jetzt nach eigenem Bedarf programmiert werden. Beispiel:

```
bool operator<(const Person& p, const Person& q)
     { return p.telNr < q.telNr; }
```

Mehrdimensionale Arrays

Einen 2D-Array (z.B. eine Matrix) kann man als Vektor von Vektoren auffassen.

Damit haben wir jetzt (endlich) mehrdimensionale dynamische Arrays.

Beispiel:

```
int m = ..., n = ...;
vector<float> v(n);
vector<vector<float> > A(m, v);
for (int i=0; i < m; i++)
   for (int j=0; j < n; j++) A[i][j] = i*j;
```


Die Matrix kann nachträglich in der Grösse verändert werden:

Eine Zeile anhängen:

```
A.push_back(v);
```

Eine Spalte anhängen:

```
for (int i = 0; i < m; i++)
     A[i].push_back(0.);
```


Ausnahmebehandlung

Der *Exception*-Mechanismus dient zum Abfangen von Fehlern oder anderen Ausnahmebedingungen.
Er besteht aus drei Teilen:

- Mit **`throw`** "wirft" man eine Ausnahme. Praktisch heisst dies: Man verlässt solange alle Schleifen und Funktionen (ähnlich wie mit **`break`** resp. **`return`**) bis man
- einen **`try`**-Block antrifft. Falls nun einer der
- dazugehörigen **`catch`**-Ausdrücke auf die Ausnahme passt, wird der entsprechende Block ausgeführt. Einen **`catch`**-Block nennt man *Handler*.

Ein Handler kann die Ausnahme nochmals werfen, damit sie von einem Handler auf einer höheren Stufe (weiter-) behandelt werden kann.

Die Syntax der Ausnahmebehandlung ist:

```
try {
  …
  throw  Ausnahme ; // auch in Schleife, Fkt.
  …
}
catch ( Datentyp   Variable ) {
  // Ausnahmebehandlung
}
```

Der Ausdruck **`Ausnahme`** ist von beliebigem Datentyp. Falls dieser mit **`Datentyp`** übereinstimmt (oder davon abgeleitet ist!), wird der Handler ausgeführt.

Bemerkungen:

- Einem **`try`**-Block können mehrere **`catch`**-Blöcke (Handler) folgen.
- Ein Handler **`catch (...)`** fängt Ausnahmen von jedem Datentyp. Wichtig: Drei Punkte (keine Metasprache!).
- Ausnahmen für die kein passender Handler gefunden wird, lösen eine neue Ausname vom Typ **`bad_exception`** aus (die dann von einem Default-Handler aufgefangen wird).
- Ein geeigneter Datentyp sind *C-Strings*. Der dazu passende Handler beginnt mit: **`catch (const char* s)`**. Siehe folgendes Beispiel:


```
double hmean(double a, double b)
{
  if (a == -b) throw "hmean: a==-b nicht erlaubt";
  return  2. * a * b  / (a + b);
}

int main()
{
  double x, y, z;
  while (cin >> x >> y) {
    try {
      z = hmean(x, y);
    }
    catch(const char* s) {
      cout << s << "\nBitte 2 andere Zahlen!\n";
      continue;
    }
    cout << "Harmonisches Mittel: " << z
         << "\nBitte 2 neue Zahlen oder 'q'!\n";
  }
  return 0;
}
```

Neben C-Strings werden vor allem *Objekte* verwendet. Hierzu definiert man sogenannte *Fehlerklassen*.

Es ist sogar üblich mit *leeren* Fehlerklassen zu arbeiten. Die Information über die Art des Fehlers befindet sich dann nicht im Objekt, sondern in dessen Datentyp. Darum ist es erlaubt, nur **`catch ( Datentyp )`** statt **`catch ( Datentyp Variable )`** zu schreiben.

Hierarchische Fehlerklassen haben den Vorteil, dass man wahlweise einen allgemeinen oder mehrere spezielle Handler verwenden kann.

Das folgende Beispiel verdeutlicht dies:


```
// Berechnung von log(x) zur Basis y:
#include <iostream>
#include <math>
#include <climits>
using namespace std;

class Matherr{};                  // Basisklasse
class Singularity : public Matherr{}; // drei
class Base        : public Matherr{}; // abgel.
class Domain      : public Matherr{}; // Klassen

double logb(double x, double y) {
   if (x < 0)                 throw Domain();
   if (fabs(x) < eps)         throw Singularity();
   if (y<0||fabs(y)<eps||fabs(y-1)<eps)
                              throw Base();
   return log(x)/log(y);
}
```



```
int main()
{
  try {
    double x, y;
    cout << "Argument: " ; cin >> x;
    cout << "Basis: " ;    cin >> y;
    cout << "log = " << logb(x,y) << endl;
  }
  //catch(Matherr)  { cout << "Fehler in logb!\n";}
  catch(Domain)     { cout << "Nicht im "
                           "Definitionsbereich!\n"; }
  catch(Singularity){ cout << "Singularitaet!\n"; }
  catch(Base)       { cout << "Nicht erlaubte "
                              "Basis!\n"; }
  catch(...)        { cout << "Andere Ausnahme\n";}
  return 0;
}
```


Die Klasse **`exception`** ist in **`#include <exception>`** vordefiniert. Sie enthält eine virtuelle Methode **`what()`** welche eine Fehlerbeschreibung liefert.

Davon abgeleitet ist z.B. die recht nützliche Fehlerklasse **`bad_alloc`** (definiert in **`#include <new>`**).

Ausnahmen dieser Klasse werden vom **`new`**-Operator geworfen, falls die gewünschte Allokation nicht möglich ist. Um den Programmabbruch zu vermeiden, kann man diese Ausnahmen abfangen und evtl. die Methode **`what()`** aufrufen.

Auch hierzu nochmals ein Beispiel:


```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

int main(int argc, char** argv)
{
  if (argc != 2) {
    cerr << "Usage: a.exe nBytes\n"; exit(1);
  }
  int n = atoi(argv[1]);
  char* p;
  try { p = new char[n]; }
  catch(bad_alloc) {
    cout << "Soviel gibt's nicht\n"; }
  cout << "Adresse " << hex << (int)p << endl;
  return 0;
}
```


Danke für Ihr Interesse

und

viel Spass und Erfolg mit C++!