Kapitel 9

Generisches Programmieren

9.1 Generische Funktionen

Eine der großen Errungenschaften der Mathematik ist die Fähigkeit, von konkreten Anwendungen zu abstrahieren und Probleme auf allgemeiner Ebene zu behandeln. Ein Satz für Vektorräume gilt für alle Vektorräume, egal ob es sich um $\mathbb{R}^n$, $\mathbb{F}_2^m$ oder Polynome handelt. Man muss daher nicht den selben Satz immer und immer wieder neu Beweisen.

Ebenso schön ist es, wenn man beim Programmieren nicht immer und immer wieder dieselben Funktionen schreiben muss. Betrachten wir beispielsweise die Funktion „Absolutbetrag“:

```
int abs(int i)
{ return i<0 ? -i : i; }

float abs(float r)
{ return r<0 ? -r : r; }

double abs(double d)
{ return d<0 ? -d : d; }
```

Bis auf den verwendeten Typ, sind diese Funktionen strukturell identisch. Man verwendet, dass die Typen `int`, `float` und `double` folgende Gemeinsamkeiten haben:

- `operator<` zum Vergleichen
- unärer `operator-`, damit das Negative zurückgegeben werden kann
- einen Konstruktor, der eine 0 als Parameter akzeptiert
- einen Kopierkonstruktor für die Rückgabe

Wenn ein weiterer Typ ebenfalls diese Voraussetzungen erfüllt, kann man die Funktion wieder komplett abschreiben. Es liegt daher nahe, den Computer diese Arbeit machen zu lassen. Wir schreiben daher nur noch eine Schablone (template) für die Funktion. Der Typ wird zum so genannten Template-Parameter:

```
template<typename T>
T abs(T x)
{ return x<0 ? -x : x; }
```

Anstelle des speziellen Typs ist nun der Platzhalter `T` getreten. Die Funktion kann (und wird) für alle Typen generiert („instanziiert“) werden, für die die nötigen Funktionen (<, -, 0, Kopie) vorhanden sind. Der Kompiler ist natürlich nicht schlau genug zu überprüfen, ob die Instanziierung Sinn macht. Falls der Operator < zum Beispiel für komplexe Zahlen definiert wäre, würde eine Funktion dafür generiert werden, die etwas anderes als den gewohnten Absolutbetrag zurückgibt.

Der Kompiler kann (derzeit) eine Funktion nur dann generieren, wenn er die Definition kennt. Insbesondere reicht also die Deklaration der Template-Funktion bzw. -Klasse nicht aus. Eine gängige Methode diese Schwäche zu umgehen ist, alle Template-Funktionen und -Klassen `inline` und damit `static` in der Header-Datei zu definieren.[1]

9.2 Generische Klassen

Was für Funktionen recht ist, ist für Klassen billigt. Gerade Containerklassen wie Listen, Bäume oder Hashtabellen schreien danach, dass man sie einmal schreibt und für verschiedenen Inhalte verwendet. Unter Java ist man dafür leider dazu gezwungen, den Typ zu casten. Dann wird aber erst zur Laufzeit (z.B. wenn das Programm bereits an den Kunden ausgeliefert ist) der Fehler erkannt. Dank Templates können wir auf Casts verzichten und erweitern die Klasse `Vektor` aus Kapitel 8.3 für allgemeine Elemente:

`Templateklasse - Deklaration`

```
template<typename T> class Vektor {
private:
  T *data;
  unsigned int groesse;
public:
  Vektor(unsigned int Groesse);
  Vektor(Vektor<T> const& v);
  ~Vektor();
  Vektor<T>& operator=(Vektor<T> const& v);

  Vektor<T> operator+(Vektor<T> const& v) const ;
  T const& operator[](unsigned int const index) const ;
  T & operator[](unsigned int const index) ;
  unsigned int Groesse() const
  { return groesse; }
};
```

Der Klassenname erhält nun als Anhängsel den Typ, für den er betrachtet wird. Gekennzeichnet wird er durch spitze Klammern. Ohne diese Kennzeichnung hätte der Kompiler keine

[1] In Zukunft muss dann der Linker den Kompiler anstoßen, um den Code für eine verlangte Funktion zu generieren.

Chance zu erraten, welchen Typ man denn meint. Bei der Definition der Klassenfunktionen steht der Typ noch nicht fest. Daher werden sie als Templatefunktionen definiert:

Templateklasse - Definitionen

```
template<typename T>
Vektor<T>::Vektor(unsigned int Groesse)
{
  groesse = Groesse;
  data = new T[Groesse];
}

template<typename T>
Vektor<T> Vektor<T>::operator+(Vektor<T> const& v) const
{
  if (groesse!=v.groesse)
    { throw "falsche Laenge"; }
  Vektor<T> Ergebnis(groesse);
  for (unsigned int position=0;
      position<groesse; ++position)
    {
      Ergebnis[position] = data[position]+v[position];
    }
  return Ergebnis;
}

template<typename T>
T const& Vektor<T>::operator[](unsigned int const index)
const
{ return data[index]; }
template<typename T>
T & Vektor<T>::operator[](unsigned int const index)
{ return data[index]; }
```

Beachten Sie, dass auch die Klasse beim Konstruktor eine Kennzeichnung mit `<T>` hat, der Name des Konstruktors aber nicht. Verwendet werden kann diese Klasse dann wie zu vermuten mit

```
    ⋮
  Vektor<double> MeinVektor(7);
  MeinVektor[4] = 3.14;
    ⋮
```

9.3 Templates – Für und Wider

Es gibt einige Gründe, die für die Verwendung von Templates sprechen:

- Durch die generische Programmierung definiert man *eine* Klasse oder Funktion für *viele* Typen.
- Dabei hat man Typsicherheit, das heißt dass *zur Kompilierzeit* sichergestellt wird, dass die verwendeten Typen die notwendigen Funktionen bereitstellen und der Typ einer Variable der richtige ist. (Es sind keine Casts notwendig.)
- Die Verwendung der Klassen oder Funktionen ist auch auch möglich, wenn eine (sinnvolle) gemeinsame Basisklasse fehlt. Einzig die verwendeten Funktionen und Variablen müssen vorhanden sein.

Es darf aber auch nicht verschwiegen werden, dass die Verwendung von Templates einige Nachteile hat:

- Die Kompilierzeiten steigen deutlich an.
- Die Funktionsnamen werden unhandlich lang.
- Man bekommt kryptische Fehlermeldungen (die selten in eine Zeile passen).
- Das Programm wird aufgebläht (wenn auch nicht der Quelltext).

9.4 Standard Template Library

Mit C++ kommen eine Menge Template-Klassen mit, die sich als sehr praktisch erweisen. Gemeint sind vor allem Containerklassen für Datenstrukturen wie doppelt verkettete Listen (`list`), dynamische Felder (`vector`) und balancierte Suchbäume (`map`). Aus diesen Datenstrukturen lassen sich dann sehr leicht kompliziertere zusammenstecken. So erhält man schnell eine Klasse für Graphen:

```
#include <vector>
#include <list>

using std::vector;
using std::list;

class Graph {
  struct Node {
    typedef Node* NeighbourType;
    typedef list<NeighbourType> NeighbourContainer;
    NeighbourContainer neighbours;
  };

  vector<Node> nodes;
};
```

Natürlich muss man sich noch um Funktionen kümmern, die etwa Knoten oder Kanten einfügen. Die wesentliche Struktur steht aber bereits. So stellen `list` und `vector` beide die

Funktion `push_back` zum hinten Anfügen zur Verfügung. Für `vector` ist der Operator `[]` definiert.

Eine Besonderheit bei diesen Containerklassen ist eine Konstruktion, die *Iterator* genannt wird. Unter einem Iterator kann man sich so etwas ähnliches wie einen Zeiger vorstellen:

- Der Operator `operator*` liefert den Zugriff auf das Element, auf den der Iterator zeigt.
- Mit dem Operator `operator++` kann man zum nächsten Element im Container weiterschalten.

Zur Verwendung von Iteratoren haben die Containerklassen zwei wichtige Funktionen:

`begin()`: Iterator, der auf das erste Element zeigt

`end()`: Iterator, der auf das erste Element *nach* dem Ende zeigt

Die seltsame Definition für `end()` führt dazu, dass man Iteratoren wie folgt verwendet:

```
int Graph::degree(Node & node){
  int number=0;
  for (Node::NeighbourContainer::iterator
            It=node.neighbours.begin();
       It != node.neighbours.end(); ++It)
    { ++number; }
  return number;
}
```

`list<Node*>::iterator` ist dabei der Typ des Iterators, der zur Klasse `list<Node*>` gehört. In der Schleife wird ein Iterator initialisiert, so dass er auf das erste Element des Containers zeigt. Dann wird die Schleife solange durchlaufen und der Iterator weitergeschaltet, bis er auf das Element nach dem letzten zeigt. Wir zählen dabei im konkreten Fall mit, wieviele Elemente wir besuchen.[2]

Das Unschöne am letzten Beispiel ist, dass der Knoten nicht verändert wird, wir aber einen nicht-konstanten Knoten benötigen, um den Iterator zu benutzen. Für diesen Fall gibt es eine zweite Art von Iterator, der für konstante Container funktioniert. (Das ist etwas anderes als ein konstanter Iterator!) Der Iterator heißt `const_iterator`:

```
int Graph::degree(Node const& node){
  int number=0;
  for (Node::NeighbourContainer::const_iterator
            It=node.neighbours.begin();
       It != node.neighbours.end(); ++It)
    { ++number; }
  return number;
}
```

[2]Natürlich wäre es in diesem Fall einfacher und effizienter, die Funktion `neighbours.size()` zu verwenden.

Nun wollen wir aber noch die Zeiger in unserem Graphen loswerden. In Wahrheit benötigt man nämlich nur einen Iterator, der weniger kann und daher weniger gefährlich ist (d.h. man tut nicht aus Versehen etwas damit, was man eigentlich nicht wollte). Mit einem Iterator sind unser Graph nun so aus:

```
class Graph {
  struct Node {
    typedef list<Node>::iterator NeighbourType;
    typedef list<NeighbourType> NeighbourContainer;
    NeighbourContainer neighbours;
  };

  vector<Node> nodes;

public:
  int degree(Node const& node);
};
```

Treiben wir das Ganze noch auf die Spitze. Wir wollen uns nicht festlegen, welchen Container wir für die Nachbarn verwenden, `list`, `vector` oder sonst irgendeinen. Daher übergeben wir der Graphklasse ein *template template Argument*.

```
template<template<typename T> class Container>
class Graph {
  struct Node {
    typedef typename Container<Node>::iterator NeighbourType;
    typedef Container<NeighbourType>  NeighbourContainer;
    NeighbourContainer neighbours;
  };

  vector<Node> nodes;

public:
  int degree(Node const& node);
};
```

Das Templateargument `Container` ist selbst eine Templateklasse. Wir können nun z.B. `Graph<list>` oder `Graph<vector>` instanziieren.

Dabei stolpern wir über eine weitere Besonderheit: Wir wollen einen Typ verwenden, der in einer Klasse definiert ist, die ein Template-Argument ist – nämlich `Container<Node>::iterator`. In diesem Fall muss dies dadurch kenntlich gemacht werden, dass davor `typename` steht.

9.5 Weitere Möglichkeiten

Die Anzahl der Template-Parameter ist erwartungsgemäß nicht auf einen beschränkt. Bei *mehreren Template-Parametern* werden diese wie gehabt durch Kommata getrennt.

Ähnlich wie bei Parametern von Funktionen, kann man auch bei Templates einen *Standardparameter* festlegen. Die Syntax hierfür ist dieselbe: Man bestimmt durch ein Gleichheitszeichen gefolgt von einem Typ.

Templateklasse mit Standardparameter

```
template<typename T = double>
class Vektor {
private:
  T *data;
    ⋮
```

Man kann somit `Vektor` ohne `<double>` verwenden und bekommt wieder den alten Vektor, d.h. den Spezialfall für `double`.

Letztlich soll noch erwähnt werden, dass Template-Parameter nicht nur Typen sondern auch *ganzzahlige Werte* sein können (z.B. 1, 2, 3 vom Typ `int`). Man kann damit z.B. Vektoren fester Länge definieren:

Templateklasse mit Integer

```
template<int L, typename T = double>
class Vektor {
private:
  T data[L];
    ⋮
```

Die Variable `Vektor<3,int> v` ist dann ein dreidimensionaler Vektor, der drei `int`s enthält. Der Unterschied zu `int v[3]` ist der, dass der Vektor in einer Klasse gekapselt ist, die

- sich um Kopieren und Zuweisung kümmern kann,
- weitere Funktionen bereitstellen kann und
- nicht aus Versehen mit `Vektor<4,int>` verwurschtelt werden kann.

In der Standard Template Library ist `bitset` sein prominentes Beispiel für eine Menge von Bits fester Größe (die Teilmengen einer Grundmenge modellieren).

9.6 Spezialitäten

Es kann den Fall geben, dass eine Template-Klasse oder -Funktion für spezielle Typen nicht instanziiert werden kann, weil eine Voraussetzung nicht erfüllt ist (i.e. eine notwendige Funktion nicht existiert). Noch unangenehmer ist der Fall, dass die Klasse oder Funktion zwar instanziiert werden kann, sie aber nicht so effizient oder gar falsch ist. In diesen Fällen hat man die Möglichkeit, den Spezialfall für diese Funktion separat zu schreiben. Der Kompiler nimmt immer die speziellste Variante, die er für die Template-Parameter finden kann. Vielleicht wollen wir ja den Fall eines zweidimensionalen Vektors gesondert behandeln:

Templateklasse Spezialisierung

```
template<typename T = double>
class Vektor<2,L> {
private:
  T x,y;
    ⋮
```

Es gibt dabei sogar die Möglichkeit, Template-Klassen rekursiv zu deklarieren. Durch eine Spezialisierung stellt man sicher, dass die Rekursion terminiert. Ein Beispiel dafür wäre ein mehrdimensionales Array:

Rekursives Array

```
template<int DIM, int LENGTH, class TYPE>
class Array{
private:
  Array<DIM-1, LENGTH, TYPE> data[LENGTH];
public:
  Array<DIM-1, LENGTH, TYPE>& operator[](int pos)
    {return data[pos];} };

template<int LENGTH, class TYPE>
class Array<1, LENGTH, TYPE>{
private:
  TYPE data[LENGTH];
public:
  TYPE& operator[](int pos)
    {return data[pos];}
};
```

Man kann sich aber auch eine Template-Klasse `IF` definieren, die einen Typ bereitstellt, der von einer Bedingung abhängt:

IF - Definition

```
template<bool Cond, typename A, typename B>
struct IF {
  typedef A RET;
};

template<typename A, typename B>
struct IF<false,A,B> {
  typedef B RET;
};
```

Diese Template-Klasse `IF` kann verwendet werden um *zur Kompilierzeit* zu entscheiden, ob in Abhängigkeit von der Bedingung `Cond` der Typ `A` oder der Typ `B` verwendet wird. Der "Rückgabewert" dieser Metafunktion `IF` ist der Typ `RET`. Betrachten Sie beispielsweise die Definition

IF - Verwendung

```
IF<sizeof(int)<4, long, int>::RET a
```

Wenn die Größe von `int` kleiner als 4 ist, wird die allgemeine Definition von `IF` benutzt. Daher ist `a` vom Typ `long`. Wenn `int` größer oder gleich 4 ist, wird die Spezialisierung von `IF` benutzt und `a` ist vom Typ `int`.

Durch die Kombination von Rekursion und Spezialisierung kann man sich beliebig komplizierte Konstruktionen ausdenken. Beispiele wären:

- Eine Kontainerklasse, die intern Pointer verwendet, wenn die Elemente eine bestimmte Größe übersteigen.
- Eine Funktion `ZZ<m>` $\times$ `ZZ<n>` $\rightarrow$ `ZZ<m*n>`, die nur dann definiert ist, wenn n und m teilerfremd sind.
- Generische Implementation von Designpattern.

In der Tat ist es so, dass man durch Templates eine eigene Programmiersprache hat, die aber zur Kompilierzeit ausgeführt wird. Genauer gesagt ist der Templatemechanismus *Turing-vollständig*. Da einmal definierte Templates nicht wieder neu definiert werden können, ist diese Programmiersprache nicht imperativ sondern funktional.

9.7 Aufgaben

1. Schreiben Sie ihre Maximumsfunktion als Template-Funktion um. Probieren Sie sie mit 4 verschiedenen Typen aus. Überlegen Sie sich, warum Sie den Template-Parameter dabei nicht angeben müssen.
2. Ändern Sie ihr dynamisches Array derart ab, dass der Typ der Elemente ein Template-Parameter ist. Probieren Sie die Funktionen mit zwei unterschiedlichen Typen aus.
3. Übernehmen Sie die generische Funktion für den Absolutbetrag. Verifizieren Sie, dass sie für ihre Klasse der komplexen Zahlen nicht funktioniert. Schreiben Sie eine Spezialisierung für komplexe Zahlen.
4. Schreiben Sie eine Template-Funktion, die das größte Element in einem Container sucht. Probieren Sie Ihre Funktion für eine Liste und einen Vector aus.
5. Lesen Sie die letzten vier Literaturangaben aus Abschnitt 1.5.[3]

[3] ;-)