Kapitel 5

Parameterübergabe

5.1 Werte- und Referenz-Parameter

In diesem Abschnitt geht es darum, was es bewirkt, wenn in einer Funktion eine Variable geändert wird. Das heißt wir haben die Situation, dass einer Funktion eine Variable übergeben wird und diese auf einen anderen Wert gesetzt wird. In C++ gibt es die Möglichkeit festzulegen, ob sich dadurch der Wert der Variablen außerhalb der Funktion ändern soll:

Wert kopieren	**Referenz**

`Funktion(double Wert)`
`{`
`  Wert = 0.0;`
`}`
	`Funktion(double & Wert)`
	`{`
	`  Wert = 0.0;`
	`}`

Änderung der Kopie außerhalb wirkungslos	Änderung wirkt sich außerhalb aus
Kopieren notwendig	(fast) keine Laufzeiteinbußen

Im ersten Fall wird der Wert kopiert, was natürlich bedeutet, dass innerhalb der Funktion diese Kopie verändert wird. Im zweiten Fall ist die Variable `Wert` eine Referenz der Variable, die beim Funktionsaufruf übergeben wird. Ein Änderung bewirkt daher eine Änderung außerhalb. In C++ gibt es hier keine Unterscheidung zwischen eingebauten Typen und selbstdefinierten Klassen. Man kann also sowohl eingebaute Typen (z.B. `int` oder `double`) wie auch selbstdefinierte Klassen (wie unsere Vektorklasse) als Kopie oder als Referenz übergeben.

5.2 Konstante Argumente

Die Übergabe einer Variable bietet nicht nur die Möglichkeit, dass man sie innerhalb der Funktion nach außen sichtbar verändern kann, es hat auch den Vorteil, dass man sich das Kopieren spart. Bei einer Variable vom Typ `double` ist das keine große Ersparnis. Wenn man aber ein Feld mit einer Million `double`s hat, dann wirkt sich das schon aus. Aber auch bei kleineren Datenstrukturen kann dies einen großen Unterschied in der Laufzeit machen, wenn diese sehr oft kopiert werden müssen. Der Autor hatte einen konkreten Fall, bei dem ein einziges Zeichen – ein „&“ – ein Programm um ein Drittel schneller gemacht hat.

Wenn man allerdings eine Variable als Referenz übergibt um sich das Kopieren zu sparen, birgt dies die Gefahr in sich, dass man die Variable aus Versehen ändert, obwohl man dies

garnicht möchte. C++ bietet eine Möglichkeit, wie sich der Programmierer dagegen schützen kann. Man kann nämlich angeben, dass sich eine Variable nicht ändern soll. Dies geschieht, indem man `const` davorschreibt:

```
Funktion(double const& Wert)
{
  Wert = 0.0; // nicht mehr möglich
}
```

Man beachte dabei die Reihenfolge. Sie ist von der Variable ausgehend von rechts nach links zu Lesen. Im vorliegenden Fall haben wir eine Referenz, die konstant ist. Umgekehrt, also `double &const Wert`, hätten wir eine Referenz von einer Konstanten, was etwas anderes ist.

Abgesehen von konstanten Argumenten, gibt es noch die Möglichkeit, anzugeben, dass eine Funktion die Klasse selbst nicht ändert. Das ist sinnvoll, da man sonst keine Funktionen von konstanten Variablen aufrufen könnte:

vektor.cc
```
#include <cmath>
#include "vektor.h"
double Vektor::norm() const
  {
    return sqrt(myX*myX+myY*myY);
  }
```

Die Norm eines Vektors kann somit berechnet werden, auch wenn der Vektor eine Konstante ist.

5.3 Kopierkonstruktor

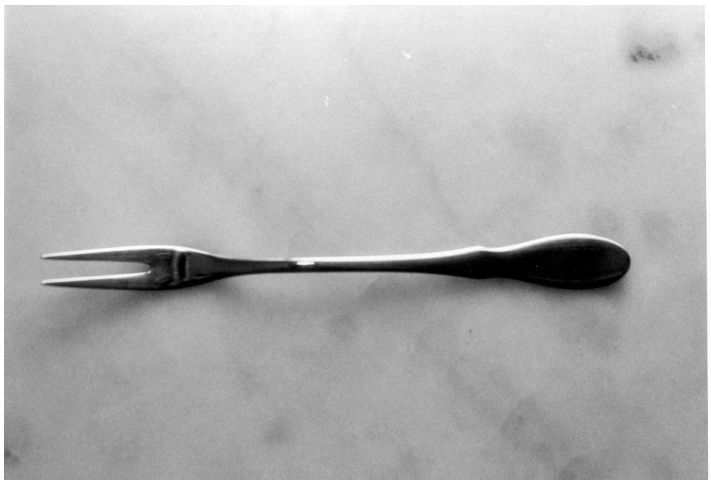
Bei selbstdefinierten Klassen mag es in einigen Fällen nicht sinnvoll sein, den Inhalt der Klasse einfach zu kopieren. Der Programmierer einer Klasse kann daher dem Kompiler mitteilen, welcher Code zum Kopieren verwendet werden soll. Das geschieht, indem man den so genannten *Kopierkonstruktor* definiert, d.h. einen Konstruktor, der als Argument eine konstante Referenz derselben Klasse erhält:

```
class Vektor {
private:
  double myX,myY;
public:
  Vektor(double x, double y)
    :myX(x),myY(y) {};

  Vektor(Vektor const& v)
    :myX(v.myX),myY(v.myY) {};

  Vektor addiere(Vektor v);
};
```

Wenn kein Kopierkonstruktur angegeben wird, generiert der Kompiler standardmäßig einen Kopierkonstruktor, der alle Elemente der Klasse kopiert.



Man kann (sollte aber nicht) beliebigen Blödsinn damit treiben:

```
Vektor::Vektor(Vektor const& v)
  :myX(v.myX+1),myY(v.myY) {};
```

In diesem Fall würde bei jeder Kopie, die x-Koordinate um eins erhöht werden. Das widerspricht vollkommen der Intuition, die man bei einem Kopierkonstruktor hat. Wir nehmen aber an, dass Sie "erwachsen" genug sind, so etwas nicht zu tun...

5.4 Kopieren und Referenzen bei Initialisierungen

Interessanterweise (oder konsequenterweise) ist es auch möglich, Referenzen außerhalb des Funktionskopfes zu deklarieren, genauso wie man Variablen deklariert. Die Referenz steht dann als Platzhalter für das Original:

Wert kopieren	**Referenz**

{
 ⋮
 `double Kontostand;`
 `Kontostand = 1000.0;`
 `double Einkommen(Kontostand);`
 `Einkommen = 0.0;`
 ⋮
 }
	{
	 ⋮
	 `double Kontostand;`
	 `Kontostand = 1000.0;`
	 `double & Einkommen(Kontostand);`
	 `Einkommen = 0.0;`
	 ⋮
	 }

Kontostand ist 1000	Kontostand ist 0
Änderung der Kopie	Änderung des Originals (und der Referenz)

Einer Referenz kann man nur bei der Initialisierung die Variable zuweisen, die sie referenzieren soll. Wenn man ihr später eine Variable „zuweist", wird deren Wert in die Variable geschrieben, die referenziert wird.

```
int main ()
{
  double a(100);
  double &b(a);
  double c(200);
  b = c; // a und b sind 200
  b = 500; // a und b sind 500
}
```

Wenn man daher eine Referenz als Mitglied einer Klasse hat, muss diese direkt im Konstruktor initialisiert werden, wie es in Abschnitt 3.1 vorgestellt wurde.

5.5 Aufgaben

1. Ändern Sie ihre Klasse für komplexe Zahlen derart ab, dass überall wo möglich `const&` bzw. `const` verwendet werden. Probieren Sie aus, was passiert, wenn Sie versuchen eine nicht-konstante Funktion auf eine konstante Variable anzuwenden.

2. Haben Sie an die Ausgabefunktion gedacht?

3. Schreiben Sie einen Kopierkonstruktor für ihre Klasse (auch wenn er dem Standardkopierkonstruktor entspricht).

4. Erklären Sie, warum der Kopierkonstruktor sinnvollerweise ein Argument mit `const&` und nicht `const` oder `&` erwartet.