Kapitel 8

Operatoren

Was Mathematiker schon vor Jahrhunderten erfunden haben, gibt es jetzt endlich in ihrer Programmiersprache: Operatoren definieren. Es ist in C++ möglich, Operatoren wie $+$, $-$ oder $*$ für eigene Klassen zu definieren. Unser Beispiel eines Vektors krankt bei der Verwendung der Addition:

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

  Vektor addiere(Vektor v);
};

      ⋮
    VektorA = VektorB.addiere(VektorC);
      ⋮
```

Eigentlich will man (zumindest als Mathematiker)[1] stattdessen lieber schreiben:

```
      ⋮
    VektorA = VektorB + VektorC;
      ⋮
```

Wir werden in Kürze erfahren, wie das geht. Neben der Addition kennt man in C++ (unter anderem) die Verknüpfungen in Tabelle 8.1.

Sehr vorsichtig sollte man mit der Definition von „`()`" sein. Von „`,`" und „`->`" lässt man am besten ganz die Finger (wenn man sie sich nicht verbrennen möchte).

[1] Es existiert mindestens ein Mensch, der das nicht möchte.

+	Typ × Typ	→	Typ	++	Typ	→	Typ
-	Typ × Typ	→	Typ	--	Typ	→	Typ
*	Typ × Typ	→	Typ	+=	Typ × Typ	→	Typ
/	Typ × Typ	→	Typ	-=	Typ × Typ	→	Typ
==	Typ × Typ	→	bool	*=	Typ × Typ	→	Typ
!=	Typ × Typ	→	bool	/=	Typ × Typ	→	Typ
>	Typ × Typ	→	bool	=	Typ × Typ	→	Typ
-	Typ	→	bool	[]	TypA × TypB	→	TypC

Tabelle 8.1: Beispiele für Operatoren in C++

8.1 Syntax für binäre Operatoren

Die Syntax für Operatoren soll am Beispiel eines binären Operators erläutert werden, also eines Operators mit zwei Argumenten. Wir wählen die Addition:

Format:
```
Typ& operator+(Typ const& ZweitesArgument) const;
```

Für unseren Vektor sieht die Deklaration folgendermaßen aus:

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

  Vektor operator+(Vektor const& v) const;
};
```

Die Frage ist nur, wo das erste Argument steckt. Die Funktion `operator+` gehört zu einer Klasse. Für diese wird sie aufgerufen. Dabei ist die Instanz, für die `operator+` aufgerufen wird, das erste Argument wohingegen das zweite übergeben wird. Das wird vielleicht etwas klarer, wenn man sich die Definition von `Vektor::operator+` ansieht:

Definition:
```
      ⋮
Vektor Vektor::operator+(Vektor const& v) const
{
  return Vektor(myX+v.myX, myY+v.myY);
}
      ⋮
```

Unäre Operatoren werden dann naheliegenderweise entsprechend ohne den Parameter deklariert und definiert.

8.2 Die goldene Regel der drei

Brauchst du eine, brauchst du alle!

Gemeint sind die folgenden Funktionen:

1. Kopierkonstruktor:
   `Vektor::Vektor(Vektor const& v)`
2. Destruktor:
   `Vektor::~Vektor()`
3. Zuweisungsoperator:
   `Vektor& Vektor::operator=(Vektor const& v)`

Es hat sich gezeigt, dass man (fast?) immer alle drei Funktionen benötigt, wenn man eine davon implementieren muss. Falls man – aus welchem Grund auch immer – eine der drei Funktionen nicht definieren möchte, sollte man sie (private) deklarieren und nicht definieren. Dadurch erhält man eine Fehlermeldung, wenn man sie doch „aus Versehen“ benutzt.

8.3 Beispiel: Dynamischer Vektor

Um die viele Theorie mit etwas Leben zu füllen, wird in diesem Abschnitt ein Beispiel etwas ausführlicher vorgestellt. Es handelt sich um einen Vektor, bei dem die Dimension bei der Initialisierung festgelegt wird.

Deklaration

```
class Vektor {
private:
  double *data;
  unsigned int groesse;
public:
  Vektor(unsigned int Groesse);
  Vektor(Vektor const& v);
  ~Vektor();
  Vektor& operator=(Vektor const& v);
    :
};
```

Die Definition des Konstruktors sieht dabei folgendermaßen aus: Wir speichern die aktuelle Größe und legen ein Feld dieser Größe an.

```
Vektor::Vektor(unsigned int Groesse)
  {
    groesse = Groesse;
    data = new double[Groesse];
  }
```

Im Fall des Kopierkonstruktors muss man die Größe aus dem alten Vektor holen. Außerdem kopiert man den Inhalt des Feldes:

```
Vektor::Vektor(Vektor const& v)
  {
    groesse = v.groesse;
    data = new double[groesse];
    for (unsigned int position=0;
        position<groesse; ++position)
      { data[position] = v.data[position]; }
  }
```

Im Destruktor wird der Speicherbereich wieder freigegeben, den man belegt hat:

```
Vektor::~Vektor()
  {
    delete[] data;
  }
```

Beim Zuweisungsoperator muss man zuerst den Speicher freigeben, den der alte Vektor belegt hat. Anschließend reserviert man den Speicher für die Kopie und führt das Kopieren durch:

```
Vektor& Vektor::operator=(Vektor const& v)
  {
    delete[] data;
    groesse = v.groesse;
    data = new double[groesse];
    for (unsigned int position=0;
        position<groesse; ++position)
      { data[position] = v.data[position]; }
    return *this;
  }
```

So gesehen ist eine Zuweisung die Kombination von Destruktor und Kopierkonstruktor. In anderen Fällen ist es aber möglich, die Zuweisung effizienter zu machen – darum gibt es die Möglichkeit, sie selbst zu definieren. Wir geben eine Referenz von dem Objekt zurück, das wir verändert haben. Dadurch sind Zuweisungen wie

```
  a = b = c;
```

möglich. Diese Zuweisungskette wird von rechts nach links abgearbeitet. (Warum wohl?) Damit ist das Ergebnis von `b = c` der neue Inhalt von `b`. Dieser kann dann `a` zugewiesen werden. Es ist sinnvoll, eine Referenz zurückzugeben, damit man sich eine (unnötige) Kopie spart. Beachten Sie, dass in C++ die Variable `this` ein Zeiger auf das aktuelle Objekt ist. Man muss daher `*this` zurückgeben.

Die Addition soll uns als Beispiel für einen binären Operator herhalten. Wir lassen nur die Addition von gleichlangen Vektoren zu.

```
Vektor Vektor::operator+(Vektor const& v) const
  {
    if (groesse!=v.groesse)
      { throw "falsche Laenge"; }

    Vektor Ergebnis(groesse);
    for (unsigned int position=0;
        position<groesse; ++position)
      {
      Ergebnis.data[position] =
        data[position]+v.data[position];
      }
    return Ergebnis;
  }
```

Beachten Sie, dass bei der Rückgabe der Gültigkeitsbereich der Variable `Ergebnis` verlassen wird. Die Variable existiert danach nicht mehr! Bei der Rückgabe wird daher mit dem Kopierkonstruktor eine Kopie erstellt. Diese Kopie ist zwar eigentlich unnötig und daher unerwünscht, ohne ausgefuchste Konstruktionen ist dies aber der Preis, den man für die schöne Syntax zahlen muss.

Zu guter Letzt ein Beispiel dafür, dass zwischen konstanten und nicht-konstanten Funktionen unterschieden wird. Der Operator „`[]`“ wird in beiden Varianten definiert. Die erste ist im Gegensatz zur zweiten nicht konstant.

```
double&
Vektor::operator[](unsigned int const index)
  { return data[index]; }

double const&
Vektor::operator[](unsigned int const index) const
  { return data[index]; }
```

Die Verwendung des Operators „`[]`“ ist diejenige, die man von Feldern kennt. Man kann also mit `A[5]` das fünfte Feld unseres Vektors ansprechen. Falls der Vektor dabei keine Konstante ist, wird die erste Variante verwendet. Die Funktion gibt eine Referenz des entsprechenden Feldes zurück. Man kann daher den Inhalt des Feldes ändern: `A[5] = 17`.

Falls jedoch der Vektor konstant ist, so wird die zweite Variante benutzt. Man erhält eine Konstante, die man nicht ändern kann. So etwas wie `A[5] = 17` ist dann nicht möglich, was ja auch sinnvoll ist, wenn der Vektor konstant ist.

8.4 Warnung

Operatoren sollten nur dann definiert werden, wenn ihre Semantik klar ist!

Selbst wenn es einem im Augenblick praktisch erscheint, sich irgendwelche Operatoren zu definieren, so kann man damit später böse Überraschungen erleben.

Schlechtes Beispiel:

```
bool
Komplex::operator<(Komplex const& c) const
  {
    return realteil<c.realteil;
  }
```

Da sich die komplexen Zahlen nicht anordnen lassen, ist es nicht sehr intuitiv dafür das Kleinerzeichen zu definieren.

8.5 Binäreren Operator außerhalb der Klasse

Abgesehen von der obigen Möglichkeit gibt es noch eine zweite Methode, binäre Operatoren neu zu definieren. Der Operator wird nicht als Funktion der Klasse definiert, sondern als „globale“ Funktion mit zwei Argumenten:

```
Vektor operator-(Vektor const& v1, Vektor const& v2)
  {
    if (v1.Groesse()!=v2.Groesse())
      { throw "falsche Laenge"; }

    Vektor Ergebnis(v1.Groesse());
    for (unsigned int position=0;
        position<v1.Groesse(); ++position)
      {
        Ergebnis[position] = v1[position]-v2[position];
      }
    return Ergebnis;
  }
```

Da nun die Funktion nicht mehr zur Klasse gehört, müssen alle Variablen und Funktionen, die benützt werden, `public` sein.[2]

8.6 Ein- und Ausgabe

Eine Anwendung der alternativen Definition eines binäreren Operators ist die (Ein- und) Ausgabe. Konkret kann man den Operator `<<` so definieren, dass wir ihn für die Ausgabe verwenden können, wie man es von anderen Typen gewöhnt ist:

```
#include <iostream>
#include "vektor.h"

int main() {
  int drei = 3; // int drei(3);
  double pi = 3.14;
  std::cout << drei << " + " << pi << std::endl;

  Vektor v(7);
  for (int w=0; w<7; ++w) { v[w] = w; }
  std::cout << v << " ist der Inhalt von v " << std::endl;

  return 0;
}
```

Dazu muss man wissen, dass `std::cout` vom Typ `std::ostream` ist. Wenn wir `std::cout << v` schreiben, dann soll der Operator `operator<<(std::ostream&, Vektor const&)` angewendet werden:

[2] Eine andere Möglichkeit ist, die Funktion als `friend` zu deklarieren, was aber kontrovers diskutiert wird.

```
std::ostream&
operator<<(std::ostream& ausgabe, Vektor const& v)
  {
    ausgabe << '(';
    for (unsigned int position=0;
        position<v.Groesse(); ++position)
      {
        ausgabe << v[position] << ' ';
      }
    ausgabe << ')';
    return ausgabe;
  }
```

Das erste Argument `ausgabe` ist in unserem Fall `std::cout`, das zweite unser Vektor `v`. Die Routine schreibt nun den Inhalt des Vektors nach `ausgabe`. Das clevere an der Konstruktion ist nun, dass wir eine Referenz der `ausgabe` zurückgeben. Das Ergebnis der Funktion ist also vom Typ `std::ostream&`. Daher kann man den Operator `<<` auf das Ergebnis anwenden und z.B. `std::cout << v << " ist der Inhalt von v " << std::endl;` schreiben.

Das schöne an unserem Operator ist, dass wir ihn auch gleich verwenden können, um in eine Datei zu schreiben:

```
#include <fstream>
#include "vektor.h"

int main() {
  int drei = 3; // int drei(3);
  double pi = 3.14;

  Vektor v(7);
  for (int w=0; w<7; ++w) { v[w] = w; }

  std::ofstream Ausgabe("MeineDatei.txt");
  Ausgabe << drei << " + " << pi << std::endl;
  Ausgabe << " v ist " << v << std::endl;

  return 0;
}
```

Die Klasse `std::ofstream` kapselt eine Datei, in die geschrieben werden kann. Da sie von `std::ostream` abgeleitet ist, funktioniert unser Ausgabeoperator dafür.

8.7 Aufgaben

1. Ändern Sie in Ihrer Klasse für komplexe Zahlen die Funktionen für Addition, Subtraktion, Multiplikation und Division in Operatoren.

2. Schreiben Sie die Ausgaberoutine in einer ihrer Klassen als Operator um. Probieren Sie, die Ausgabe mit anderen Typen zu kombinieren sowie die Ausgabe in eine Datei.

3. Wenden Sie die goldene Regel der Drei auf ihr dynamisches Array an.