4.3 Von C nach C++/Overloading von Funktionen

4.3.1 Beispielprogramm

In C++-Programmen findet man als C-Programmierer ab und zu ungewöhnliche Konstruktionen, so wie hier (`cppquadrat.cpp`):

```
1  #include <iostream>
2  using namespace std;
3  int quadrat(int n)
4  {
5     return n*n;
6  }
7  double quadrat(double x)
8  {
9     return x*x;
10 }
11 int main()
12 {
13    int n;
14    cout << "Bitte eine ganze Zahl eingeben:";
15    cin >> n;
16    cout << "Erstens: " << quadrat(n) << endl;
17    double x;
18    cout << "Bitte eine Zahl eingeben:";
19    cin >> x;
20    cout << "Zweitens: " << quadrat(x) << endl;
21    return 0;
22 }
```

4.3.2 Was möglich ist

Hier tragen also verschiedene Funktionen den gleichen Namen `quadrat()`. Das ist in C++ erlaubt, wenn die Funktionen voneinander abweichende Parameterlisten besitzen. Diesen Sachverhalt nennt man *das Überladen von Funktionsnamen* oder kurz *das Überladen von Funktionen* oder auch *Overloading*.

Man kann das Überladen von Funktionsnamen benutzen, um gleiche Rechenoperationen mit verschiedenen Datentypen zu erlauben. In C++ kann man sogar Operatoren wie `+, -, *, /` überladen und ihnen damit neue Bedeutungen verleihen. Auf diese Art sind die Bit-Schiebe-Operatoren << und >> in Zusammenhang mit den Objekten `cout` und `cin` zu neuen Möglichkeiten gekommen.

Andererseits kann es gefährlich sein, wenn Überladen und implizite Typumwandlungen zusammentreffen: Falls es `quadrat(int)` und `quadrat(unsigned long)` schon gibt, welche Funktion wird dann aufgerufen, wenn man ein `unsigned int` übergibt? Es ist zwar eindeutig geregelt, aber viele Programmierer kennen nicht alle Regeln auswendig. Und was ist, wenn mehrere Parameter nicht passend sind und implizit umgewandelt werden müssen?

4.3.3 Was nicht möglich ist

Beim Überladen von Funktionsnamen müssen die Parameterlisten der beteiligten Funktionen verschieden sein, sonst können die Funktionen nicht auseinandergehalten werden. Der folgende Quelltext ist also falsch:

```
1     int a;
2     unsigned int b;
3     int xhochvier(int x){ return x*x*x*x; }
```

```
4     unsigned int xhochvier(int z){ return z*z*z*z; }
5     a = xhochvier(6);
6     b = xhochvier(7);
```

4.3.4 Wie es intern abläuft

Wer sorgt eigentlich dafür, dass immer die richtige Funktion eingesetzt wird, wenn mehrere Funktionen den gleichen Namen haben? Es sind der Compiler und der Linker. Der Compiler hängt intern an die Funktionsnamen einen Code an, der die Datentypen in der Parameterliste angibt. Und der Linker setzt das Programm dann anhand der gewünschten Namen zusammen. In C war es noch so, dass der Linker nur die Funktionsnamen bekam. Hier ist ein C-Beispielprogramm analog zum obigen C++-Programm (`cquadrat.c`):

```
1  #include <stdio.h>
2  int quadratvomint(int n)
3  {
4     return n*n;
5  }
6  double quadratvomdbl(double x)
7  {
8     return x*x;
9  }
10 int main(void)
11 {
12    int n;
13    double x;
14    printf("Bitte eine ganze Zahl eingeben:");
15    scanf("%i", &n);
16    printf("Erstens: %i\n", quadratvomint(n));
17    printf("Bitte eine Zahl eingeben:");
18    scanf("%lf", &x);
19    printf("Zweitens: %f\n", quadratvomdbl(x));
20    return 0;
21 }
```

Wenn man das compiliert (ohne zu linken, daher mit Option `-c`), kann man sich anschließend mit dem Befehl `nm` die Namen der angebotenen Programmteile ansehen:

Terminal

```
schueler@debian964:~$ gcc -c cquadrat.c
schueler@debian964:~$ nm cquadrat.c
...
0000000c T quadratvomdbl
00000000 T quadratvomint
```

Man sieht, dass die Funktionen `quadratvomdbl` und `quadratvomdbl` zur Verfügung stehen. Bei C++ dagegen sieht es anders aus:

Terminal

```
schueler@debian964:~$ g++ -c cppquadrat.cpp
schueler@debian964:~$ nm cppquadrat.cpp
...
0000000c T _Z7quadratd
00000000 T _Z7quadrati
...
```

Hier ist (abgesehen vom Vorsatz `_Z7`) an jeden der beiden Funktionsnamen etwas Anderes angehängt: ein `i` an die Funktion mit der Parameterliste `(int)` und ein `d` an die Funktion mit der Parameterliste `(double)`.

4.3.5 Ein bisschen Polymorphie

Wenn verschiedene Vorgänge gleich behandelt werden können und dennoch nicht gleich ablaufen, spricht man von *Polymorphie*. Und das ist hier der Fall. Aber eigentlich sind dabei nur Compiler und Linker beteiligt. Man nennt das *frühe Bindung*. Zur Laufzeit des Programms werden ganze Zahlen und Kommazahlen ganz wie in C getrennt behandelt. Deshalb nennt man dies hier auch nur *Schwache Polymorphie*.

Der Gedanke der Objektorientierung geht darüber hinaus: Man möchte gern mit einheitlichen Aufrufen verschiedene Objekte (Variablen) gleich behandeln und trotzdem dafür sorgen, dass erst zur Laufzeit immer die richtige Funktion aufgerufen wird (*späte Bindung*). Das wird dann mit so genannten *Basisklassenzeigern* möglich sein.