Überladen von Funktionen in C++

Überladen von Funktionen = **Verwendung des gleichen Funktionsnamens im gleichen Scope für unterschiedliche Funktionen.** Das gibt es in Python nicht, in C ist es verboten, in C++ gilt:

1) Funktionen in verschiedenen Scopes dürfen immer denselben Namen haben.
2) Funktionen im gleichen Scope dürfen **denselben Namen** haben, wenn sie sich in **der Parameterliste** unterscheiden (entweder haben sie eine andere Anzahl von Parametern oder wenigstens ein Parameter hat einen unterschiedlichen Typ). **Ein unterschiedlicher Rückgabetyp reicht nicht aus**, **auch nicht ein unterschiedlicher Bezeichner eines Parameters:**

Erlaubt sind z.B. folgenden Paare von Funktionen:

```
void read(int a)          double read(double a)
      // jeweils 1 Argument, aber unterschiedlicher Typ
int f(int a)              int f(int a, int b)
      // 1 Argument vs. 2 Argumente
int f(int a)              int f(int& a)
      // jeweils 1 Argument, aber unterschiedlicher Typ!!
```

Falsch:

```
int f(int a)              double f(int b)
      // jeweils 1 Argument mit gleichem Typ!!
      // Rückgabetyp und Name der Parameter werden NICHT beachtet
```

Überladene Funktionen müssen keineswegs eine ähnliche Aufgabe haben (obwohl das sinnvoll und meist der Fall ist).

Overload Resolution

Darunter versteht man die Ermittlung der „am besten passenden“ Funktion für einen Aufruf durch den Compiler. Ich kenne keinen guten deutschen Ausdruck für diesen Vorgang. Bei jedem Funktionsaufruf muss C++

1) alle Objekte (Funktionen, Variable) mit diesem Namen finden, die zu diesem Zeitpunkt bekannt sind. Natürlich hat auch dieser Vorgang ein komplexes Regelwerk (ADL = Argument Dependent Lookup), auf welches ich hier nicht eingehe.
2) alle Objekte aus 1) ausscheiden, die zum Aufruf inkompatibel sind: wenn sie nicht aufrufbar sind, die Argumentzahlen nicht stimmen, der Argumenttyp nicht passt etc.
3) Alle verbliebenen Kandidaten bewerten: welche Argument-Konvertierungen sind für den Aufruf nötig?
4) eventuell noch Tiebreak – Regeln beachten

Am Ende der Overload-Resolution muss es für den Compiler **eindeutig** sein, welche der überladenen Funktionen aufgerufen werden soll, **sonst ist es ein Fehler!**

ALs Beispiel deklariere ich folgende 3 Funktionen, die das Overload-Set bilden sollen. Diese unterscheiden sich in wenigstens einem Parameter, sodass diese Deklarationen korrekt sind:

```
1) int f(int, double);
2) int f(double, int);
3) int f(double, double);
```

```
f(1, 1.0)       // Aufruf von 1), da exakte Übereinstimmung
f(1.0, 1)       // Aufruf  von 2), da exakte Übereinstimmung
f(1.0, 2.0)     // Aufruf von 3), da exakte Übereinstimmung
f(1, 2)         // Fehler: kein eindeutiges bestes Ziel f() ermittelbar
```

`f(1, 2)`: könnte 1) aufrufen und `2` in `2.0` umwandeln
könnte 2) aufrufen und `1` in `1.0` umwandeln
könnte 3) aufrufen und `1` in `1.0` und `2` in `2.0` umwandeln

Für jedes Argument ermittelt C++, wie „heftig“ die Umwandlung in den Zieltyp ist (C++ unterscheidet hier 4 Stufen). Eine Funktion 1 ist besser geeignet als eine Funktion 2, wenn **jede** Umwandlung der einzelnen Argumente bei Funktion 1 **nicht „heftiger“** als bei Funktion 2 ist und wenn **mindestens eine Umwandlung besser** ist. Bei Gleichstand gibt es noch Regeln für den Tie-Break. Sind danach immer noch mehrere Funktionen für den Aufruf zur Auswahl, ist das ein Fehler (ambiguous call …)! Im Fall `f(1, 2)` gibt es keinen eindeutigen Sieger: 1) gewinnt gegen 3), 2) gewinnt gegen 3) aber 1) gegen 2) ist unentschieden -> Fehler. Entfernt man z.B. Variante 2), so sind alle Aufrufe korrekt (bitte überlegen)!

Das Überladen von Funktionen in der Standardbibliothek

C++ macht von diesen Möglichkeiten auch in den Standardbibliotheken (z.B. `<cmath>`, `<iostream>`) regen Gebrauch. Alle Funktionen der Mathematikbibliothek werden in C++ für die Typen `float`, `double` und `long double` extra implementiert:

```
float x;
sin(x);         // ->  float sin(float)
sin(1.0);       // ->  double sin(double)
```

Auch die C++ Operatoren sind in Wirklichkeit Funktionen, die überladen wurden :

```
cout << 1;      // -> ostream& operator<<(ostream&, int)
```

```
cout << 1.0;   // -> ostream& operator<<(ostream&, double)
```

C muss für die unterschiedlichen Varianten unterschiedliche Namen definieren, z.B.:

```
int abs(int); long labs(long); double fabs(double);
```