Informatik Übungsstunde - Woche 10

Eric Ceglie

Überladen von Funktionen

In C++ ist es tatsächlich möglich, dass zwei Funktionen denselben Namen haben, solange der Compiler eine andere Möglichkeit hat, die Funktionen voneinander zu unterscheiden. Solche weiteren Möglichkeiten zur Unterscheideung von Funktionen sind *Anzahl* und *Typen* ihrer Argumente (Inputs).

Beispiele

Folgendes ist ein Beispiel zur Unterscheidung von Funktionen mittels unterschiedlicher Anzahl der Argumente:

```
int foo(const int a){ ... }

int foo(const int a, const int b){ ... }
```

Und ein Beispiel zur Unterscheidung mittels verschiedenen Typen der Argumente:

```
int foo(const int a){ ... }
int foo(const float a){ ... }
```

Beachte, dass bei den folgenden zwei Beispielen, der Compiler die Funktionen nicht unterscheiden kann (warum?), womit es zu einem Fehler kommt:

```
int foo(const int a){ ... }
int foo(const int b){ ... } // compiler error
```

```
int foo(const int a){ ... }
double foo(const int a){ ... } // compiler error
```

Anwendungsbeispiel

(Methematische Details sind natürlich nicht Prüfungsrelevant)
Klassisch ist die *Fakultät* einer Zahl $n \in \mathbb{N}$ definiert durch

$$n! := \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n.$$

Doch es gibt eine (auf gewisser Weise) eindeutige Funktion

$$\Gamma : \mathbb{C} \setminus \{-k \mid k \in \mathbb{N}_0\} \to \mathbb{C}$$

mit der Eigenschaft

$$\forall n \in \mathbb{N} : \Gamma(n) = (n-1)!$$

womit sich die Fakultät zu einer Funktion erweitern lässt, die *fast* auf der ganzen komplexen Zahlenebene definiert ist!
Für komplexe Zahlen $z \in \mathbb{C}$ mit $\mathrm{Re}(z) > 0$ ist Γ gegeben durch

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\, dz.$$

Die Standardbibliothek `cmath` erlaubt es uns den Wert der Gamma Funktion an einer beliebigen Stelle mit der Funktion `std::tgamma()` zu berechnen. Angenommen wir wollen nun eine Funktion für die Fakultät schreiben und dabei nur im absoluten Notfall auf die komplizierte Gamma Funktion zurückgreifen. Für alle natürlichen Zahlen soll unser Programm die klassische Methode zur Berechnung der Fakultät benutzen. Indem wir Funktionen Überladen lässt sich dies wie folgt implementieren:

```cpp
#include <iostream>
#include <cmath>

int factorial(int n){
  std::cout << "Using simple multiplication...\n";
  int res = 1;
  for(int i = 1; i <= n; i++){
    res *= i;
  }
  return res;
}
double factorial(double n){
  std::cout << "Using the gamma function...\n";
  return std::tgamma(n+1);
}


int main(){
  std::cout << factorial(5) << "\n";
  std::cout << factorial(5.5) << "\n";
  std::cout << factorial(-3.4) << "\n";

  return 0;
}
```

Output:

```
Using simple multiplication...
120
Using the gamma function...
287.885
```

```
Using the gamma function...
-1.10803
```

Klassen

Klassen sind ähnlich zu Structs, nur viel mächtiger.

Private vs. Public

Anders als bei Structs gibt es bei Klassen einen *public* und einen *private* Bereich. Auf alles was im private Bereich definiert wird, kann nur innerhalb der Klasse zugegriffen werden.

Memberfunktionen

Memberfunktionen sind Funktionen, die innerhalb einer bestimmten Klasse definiert werden. Diese können dadurch auch auf private Variablen der Klasse zugreifen und diese auch verändern. Damit kann man eine Brücke von aussen zum privaten Bereich einer Klasse bauen. Eine Memberfunktion kann zudem als `const` deklariert werden, damit sichergestellt wird, dass sie keine Variablen der Klasse verändert.

Konstruktor

Der Konstruktor einer Klasse ist eine spezielle Memberfunktion, der den Namen der Klasse selbst trägt. Der Konstruktor wird dafür benutzt, um beim Initialisieren einer Instanz der Klasse die Variablen mit Inputs zu belegen. Der *Default-Konstruktor* (kein Argument) wird automatisch erzeugt, falls eine Klasse keinen Konstruktor definiert. Wie bei üblichen Funktionen kann der Konstruktor auch überladen werden, um dem User verschiedene Möglichkeiten zu bieten, eine Klasse zu initialisieren.

Beispiel

```
#include <iostream>
#include <string>

class BankAccount{
  private:
    double balance;

  public:
    std::string owner;

    // Konstruktoren:
    BankAccount(std::string name, double value): owner(name), balance(value)
{
      std::cout << "Bank account for " << owner << " was created, ";
      std::cout << "starting with a balance of "<< balance << "CHF.\n";
    }
```

```
    BankAccount(double value): owner("unknown_user"), balance(value){
      std::cout << "Bank account for " << owner << " was created, ";
      std::cout << "starting with a balance of "<< balance << "CHF.\n";
    }

    // Member Funktionen:

    // PRE:  nan
    // POST: returns current balance of account
    double get_balance() const{
      return balance;
    }

    // PRE:  nan
    // POST: prints current balance of user in console
    void print_balance() const{
      std::cout << owner << " currently owns " << balance << "CHF.\n";
    }

    // PRE:  nan
    // POST: changes current balance by value
    void change_balance(double value){
      balance += value;
    }
};

int main(){
  BankAccount acc1("Manuel", 145);
  BankAccount acc2(529);
  acc1.change_balance(16);
  std::cout << acc1.owner << " now owns " << acc1.get_balance() << "CHF.\n";
  acc1.print_balance();
  acc2.print_balance();

  return 0;
}
```

Output:

```
Bank account for Manuel was created, starting with a balance of 145CHF.
Bank account for unknown_user was created, starting with a balance of
529CHF.
Manuel now owns 161CHF.
Manuel currently owns 161CHF.
unknown_user currently owns 529CHF.
```