TU Berlin, Institut für Mathematik
Thomas Slawig

Funktionen in C++

Funktionen sind ein wesentliches Merkmal des strukturierten Programmierens. Sie sind in sich geschlossene Einheiten, die Anweisungen zusammenfassen. Sie haben wie eine mathematische Funktion Argumente (hier: Parameter) und einen Rückgabewert. Algorithmen, die mehrfach benutzt werden, werden sinnvollerweise in einer Funktion realisiert und im Programm mehrfach aufgerufen. Funktionen dienen der Strukturierung eines Programmes, sie machen es übersichtlich und besser lesbar, sie ersparen mehrfaches Codieren der selben Dinge und stellen sicher, dass Änderungen nur einmal implementiert werden müssen, aber überall Auswirkungen haben. Die Details eines Algorithmus (d.h. die Anweisungen einer Funktion) sind an der Stelle wo sie benutzt wird verborgen. Jedes größere und dennoch lesbare Programm arbeitet mit Funktionen.

- Eine Funktion in C++ hat einen **Namen**, der den Konventionen für Variablen folgt (s. unter Grundstruktur eines C++ Programms).

- Eine Funktion kann mehrere Eingabe- und *maximal einen* Rückgabewert haben. Die Eingabewerte nennt man auch die **Parameter** der Funktion.

- In der **Deklaration** einer Funktion werden Name, Parameter- und Rückgabetypen festgelegt, z.B.

  ```
  double min(double x, double y);
  ```

  Dies bezeichnet man oft auch als Prototype oder Interface (Schnittstelle). Die Parameter in der Deklaration der Funktion nennt man **Formalparameter**. Die Deklaration einer Funktion sollte *außerhalb aller anderen Funktionen*, also auch außerhalb der Hauptfunktion `main` erfolgen. Die Deklaration muss in jeder Datei, in der die Funktion benutzt wird, stehen, z.B. für das Hauptprogramm:

  ```
  double min(double x, double y);
  int main()
  {
     ...
     z=min(x,y);
  }
  ```

  Der Compiler kennt dadurch beim Übersetzen der Quelldatei den Namen sowie Ein- und Rückgabewerte der Funktion. Das reicht, um diese Datei auf richtige Syntax zu prüfen.

- Oft schreibt man die Deklaration in eine eigene Headerdatei, z.B. `min.h`, die man mit einer Präprozessoranweisung einbindet:

  ```
  #include <iostream>
  #include "min.h"
  int main()
  {
     ...
  }
  ```

- Der eigentliche Quellcode ist in der **Definition** einer Funktion zu finden. Diese beginnt mit dem Funktionskopf oder Signatur (das ist die Deklaration ohne Semikolon). Danach folgt der Block mit den eigentlichen Anweisungen in geschweiften Klammern. Ist ein Rückgabewert vorhanden, so wird dieser durch die `return`-Anweisung zurückgegeben und damit die Funktion beendet, egal wo die `return`-Anweisung steht. Ein Beispiel:

  ```
  double min(double x, double y)
  {
     if (x<=y) return x;
     else      return y;
  }
  ```

In diesem Sinne ist ein Hauptprogramm nichts anderes als die Definition einer Funktion `main`. Die Formalparameter werden *innerhalb der Funktion* wie Variablen benutzt, ohne noch einmal deklariert zu werden. Der Übersicht wegen sollte man Funktionsdefinitionen in eigenen Quelldateien (z.B. `min.cc`) speichern. So kann man sie getrennt compilieren. Erst beim Linken wird dann der eigentliche Code der Funktion `min` mit dem von `main` verbunden. Stimmen jetzt Deklaration und Funktionskopf in der Definition nicht überein oder fehlt der Quellcode einer benutzten Funktion, dann gibt es eine Fehlermeldung.

- Gibt es **keine Parameter**, so bleiben die runden Klammern leer oder der Typ `void` wird benutzt:

  `double f()` oder `double f(void)`

- Gibt es **keinen Rückgabewert**, so wird der Typ `void` gesetzt. Die `return`-Anweisung kann entfallen oder ohne Argument benutzt werden:

```
void f(double x)
{
   ...
   return;
}
```

- Der **Aufruf** einer Funktion erfolgt durch den Namen und die **Übergabe** der Eingabewerte, z.B.

  `min(a,b);` bzw. `f();` bei einer Funktion mit bzw. ohne Parameter.

  Hat die Funktion einen Rückgabewert, so kann dieser einer Variablen zugewiesen werden, z.B.

  `c=min(a,b);` bzw. `z=f();`

  Die Parameter, die der Funktion beim Aufruf übergeben werden, heissen **aktuelle** oder **Aktualparameter**. Sie müssen nicht die Namen, sollten aber den jeweiligen Typ der Formalparameter in der Deklaration der Funktion haben. Wichtig ist die *Reihenfolge der Parameter* in der Parameterliste. Im obigen Beispiel der Funktion `min` entsprechen also die Aktualparameter `a` und `b` den Formalparametern `x` und `y` in der Deklaration.

- Parameter können mit voreingestellten Werten (*defaults*) gesetzt werden:

```
double f(double x, double y=0.0)
```

  Solche Parameter heissen **optional**. Wird die Funktion mit zwei Parametern aufgerufen, dann wird der Vorgabewert überschrieben. Die Funktion kann aber auch mit nur einem Paramter aufgerufen werden, dann wird für den anderen (hier `y` der Vorgabewert benutzt. Logischerweise müssen optionale Parameter *am Ende der Parameterliste* stehen.

- Funktionen ohne Rückgabewert nennt man **Prozeduren**. Sie benutzt man oft, um ein Programm besser zu strukturieren, d.h. Teile davon zusammenzufassen und mit einem prägnanten Namen zu versehen, z.B.

```
int main()
{
   mach_was();
   jetzt_mach_was_anderes();
}
```

- Beim Aufruf einer Funktion werden die *Werte* der Aktualparameter auf die Formalparameter kopiert und in der Funktion als Variablen benutzt. So sind z.B. `x` und `y` nach dem Aufruf von `min` im obigen Beispiel unverändert. Analog ist nach dem Aufruf von `f` in

```
void f(double x)
{
   x=x+1.0;
}
int main()
{
```

```
    double x=1.0;
    f(x);
}
```

der Wert des Aktualparameter `x` in `main` *nach dem Aufruf* derselbe wie vorher (`x` = 1). Man bezeichnet dies als ***call by value***, da nur die *Werte* der Variablen übergeben werden.

Gerade bei Prozeduren will man aber Eingabeparameter verändern, da die Prozedur ja sonst keine Auswirkungen auf das Hauptprogramm hätte. Dazu übergibt man als Parameter nicht den Wert der Variablen, sondern ihre **Referenz**: Hinter die Typenbezeichnung des Parameters oder vor den Variablennamen wird der *Referenzoperator* `&` geschrieben:

`void f(double& x)` oder (äquivalent) `void f(double &x)`

Der Rest der Funktionsdeklaration und ihr Aufruf in `main` bleiben unverändert. Im obigen Beispiel hat dann `x` nach dem Aufruf von `f` in `main` den Wert 2. Diese Deklaration und den entsprechenden Aufruf bezeichnet man als ***call by reference***, da hier *Referenzen* auf Variablen übergeben werden. Die Werte der Aktualparameter (also der übergebenen Variablen aus der aufrufenden Funktion, z.B. `main`), werden *nicht* kopiert.

- Die *call by value*-Variante wird also benutzt, wenn die Funktion die Eingabeparameter *nicht* verändern soll. Die Werte der Aktualparameter werden auf die Formalparameter der Funktion kopiert. Dies kann bei großen Datenmengen sowohl zeit- als auch speicheraufwändig sein. Praktischer wäre es, die *call by reference*-Methode zu verwenden. Dort werden die Werte der Parameter ja nicht kopiert (s.o). Dann muss man aber sicherstellen, dass die Werte der Parameter innerhalb der Funktion *nicht verändert werden können*. Dies geschieht durch ein vorangestelltes `const`:

  `void f(const double& x)` bzw. `void f(const double &x)`

  Damit wird der Compiler jeden Versuch, den Wert des Parameters `x` innerhalb der Funktion zu ändern, als Fehler melden.

- Da eine Funktion nur einen Rückgabewert haben kann, braucht man also *call by reference*, um in einer Funktion mehrere Variablen der übergeordneten Funktion zu verändern. Es ist sinnvoll, dann *keinen* Wert zurückzugeben und alles über die Parameter zu realisieren. Umgekehrt sollte eine Funktion mit Rückgabewert die Parameter unverändert lassen, also nur mit *call by value* arbeiten.

- Übergabe von Funktionen an Funktionen:

  In manchen Fällen ist es sinnvoll, Funktionen an andere Funktionen als Parameter zu übergeben, z.B. wenn eine Funktion die Ableitung (oder das Integral) einer beliebigen mathematischen Funktion berechnen soll. Dazu gibt man in der Parameterliste der Funktion, die die Ableitung berechnet, die zu übergebende Funktion (z.B. `f`) als Parameter an:

  ```
  double ableitung(double f(double x));
  ```

- Wenn man in einer Funktion oder einem Programm mehrere Funktionen benutzt und die Deklarationen in Header-Dateien geschrieben hat, so kann es passieren, dass die Deklaration einer Funktion mehrfach auftaucht, weil sie von mehreren Funktionen benutzt wird. Der Compiler gibt einen Fehler aus. Um das zu vermeiden, kann man die **bedingte Compilierung** und den Präprozessor nutzen, z.B. in `min.h`:

  ```
  #IFNDEF MIN_H
  #DEFINE MIN_H
  double min(double x, double y);
  #ENDIF
  ```

  Der Text zwischen `#IFNDEF` (= *if not defined*) und `#ENDIF` wird nur eingebunden, wenn die Präprozessor-Variable `MIN_H` noch *nicht* gesetzt ist. Beim ersten Einbinden von `min.h` wird also die Deklaration der Funktion eingebunden, bei einem zweiten Einbinden in einer anderen Header-Datei nicht mehr. Analog funktioniert bedingte Compilierung mit `#IFDEF`.

- Funktionen können sich selbst aufrufen, man spricht dann von **Rekursion**.