Programmiermethoden in der Mathematik WS 02/03
Funktionen in C++

Funktionen dienen zur Strukturierung eines Programmes und/oder fassen Anweisungen zusammen, die oft benutzt werden.

- 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 Rückgabewert mit ihren Typen festgelegt. 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;
}
```

  Die erste Zeile der Deklaration nennt man auch **Funktionskopf**. Die einzelnen Parameter in den runden Klammern (in der **Parameterliste**) werden durch Kommas getrennt. Die Parameter in der Deklaration der Funktion nennt man **Formalparameter**. Sie werden *innerhalb der Funktion* wie Variablen benutzt, ohne noch einmal deklariert zu werden.

- 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 **muss** nach dem ANSI-Standard (z.B. beim `g++`) der Typ `void` gesetzt werden. 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.

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

- An diesen Beispielen sieht man, dass das Hauptprogramm `main`, das in jedem C++ Programm vorhanden sein muss, nichts anderes als eine Funktion ist. Man nennt sie deshalb auch besser **Hauptfunktion**.

- Die Deklaration einer Funktion muss *außerhalb aller anderen Funktionen*, also auch außerhalb der Hauptfunktion `main` erfolgen. Der Übersicht wegen sollte man Funktionsdeklarationen in eigenen Quelldateien speichern. So kann man sie getrennt compilieren, z.B. mit

  `g++ -c min.cc` (erzeugt Objektdatei `min.o`).

  In der Quelldatei der Hauptfunktion muss außerhalb von `main` nur der **Funktionsprototyp** (**Signatur**) stehen. Darunter versteht man den Funktionskopf *ohne die Parameternamen*, z.B. für das obige Beispiel der Funktion `min`:

```
#include <iostream.h>
double min(double, double);

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

  Der Compiler kennt dadurch beim Übersetzen der Quelldatei, die `main` enthält, den Namen sowie Ein- und Rückgabewerte der Funktion. Das reicht, um die Hauptfunktion auf richtige Syntax zu prüfen und getrennt zu übersetzen, z.B. mit

  `g++ -c main.cc` (erzeugt Objektdatei `main.o`).

  Erst beim Linken wird dann der eigentliche Code der Funktion `min` mit dem von `main` verbunden:

  `g++ main.o min.o -o prog1` (erzeugt ausführbare Datei `prog1`).

  Stimmen jetzt Funktionsprototyp und Funktionskopf in `min.cc` nicht überein oder fehlt der Quellcode einer benutzten Funktion, dann gibt es eine Fehlermeldung.

  Mehrere Funktionsprototypen fasst man oft in eigenen Headerdateien zusammen, um das Hauptprogramm übersichtlicher zu gestalten:

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

main()
{
   ...
}
```

- 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;
}
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)` im Funktionskopf und `void f(double&);` in der Signatur

oder (äquivalent)

`void f(double &x)` im Funktionskopf und `void f(double &);` in der Signatur

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)` im Funktionskopf und `void f(const double&);` in der Signatur

  bzw.

  `void f(const double &x)` im Funktionskopf und `void f(const double &);` in der Signatur

  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.

- Wenn man in einer Funktion oder einem Programm mehrere Funktionen benutzt und die Signaturen in Header-Dateien geschrieben hat, so kann es passieren, dass die Signatur 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, double);
  #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 Signatur der Funktion eingebunden, bei einem zweiten Einbinden in einer anderen Header-Datei nicht mehr. Analog funktioniert bedingte Compilierung mit `#IFDEF`.