

Merkblatt für Funktionen und Kontrollstrukturen

Funktionen Die Syntax von Funktionen kann fast eins-zu-eins von Javaprogrammen übernommen werden. Ein wichtiger Unterschied zu Java ist die Unterscheidung zwischen *Funktionsdeklaration* und *Funktionsdefinition*. Die Deklaration lässt den Compiler lediglich wissen, dass es eine Funktion mit bestimmtem Prototyp (Signatur) gibt. Eine Funktionsdefinition haucht der Funktion sozusagen Leben ein, d.h. sie beschreibt, welchen Algorithmus die Funktion bei ihrem Aufruf ausführen soll.

Eine Funktionsdeklaration einer Funktion f mit 3 Parametern (x , y und z) könnte folgendermaßen aussehen:

```
int f( float x, float y, float z );  
//Hier werden die Parameternamen explizit genannt  
  
int f( float, float, float );  
//Hier verzichtet man auf Parameternamen (reine Signatur)
```

Eine Funktionsdefinition dagegen muss immer den Parametern Namen geben, falls sie verwendet werden, was meistens sinnvoll ist. Sie hängt einen Rumpf an die Deklaration an. Eine Funktionsdefinition für die Funktion f könnte folgendermaßen aussehen:

```
int f( float x, float y, float z )  
{  
    x = y = z;  
}
```

Wieso trennt man nun Deklaration und Definition? Dies hat mehrere Gründe. Einer davon ist, dass man dadurch eine Schnittstelle für Programmierer anbieten kann, ohne dass man die wirkliche Implementierung preis gibt. Ein anderer ist, dass bei jeder Definition Speicher vom Compiler allokiert wird, die obige Definition der Funktion f sagt dem Compiler, dass er f an dieser Stelle erzeugen soll, also Speicher dafür reservieren soll. Die Deklaration dagegen sagt dem Compiler nur, dass es solch eine Deklaration mit bestimmter Aufrufsignatur gibt, allerdings wird noch kein Speicher allokiert.

Aus diesen Gründen werden Funktionsdeklaration und -definition auch an verschiedenen Stellen im Code gestellt, meistens sogar in verschiedenen Dateien. C++ unterscheidet zwischen *header*-Dateien (mit Dateierdung *.h*) und *source*-Dateien (mit Dateierdung *.cpp*). In header-Dateien stehen u.a. die Funktionsdeklarationen, in source-Dateien die Funktionsdefinitionen.

Hierzu ist noch hinzuzufügen, dass diese Trennung von Deklaration und Definition nicht nur für Funktionen, sondern auch für Variablen gilt. Bei einer normalen Deklaration, z.B.

```
int i;
```

ist genügend Information vorhanden, um die Variable auch zu definieren, ihr also Speicherplatz zuzuweisen. Um dies zu vermeiden, muss ein gesondertes Schlüsselwort angegeben werden, das dem Compiler sagt, dass nur der Variablenname bekannt gemacht werden soll, nicht aber Speicher allokiert wird. Dieses Schlüsselwort ist `extern`:

```
extern int a; //Deklaration ohne Definition

int b;       //Deklaration & Definition
```

Kontrollstrukturen Ebenso wie in Java kann der Programmfluss eines C++-Programms durch Kontrollstrukturen gesteuert werden. Es gibt auch hier bedingte Anweisungen, Wiederholungsanweisungen usw. Die Syntax ist hier dieselbe wie in Java, bzw. in Java dieselbe wie in C(++), um den Erfindern dieser Sprache Genüge zu tun.

- *if-else*

```
if( CONDITION )
{
    //mache irgendwas
}
else
{
    //mache etwas anderes
}
```

Ein Beispiel dafür wäre:

```
int a = 1;
int b = a+1;
if ( a < b )
{
    b = b - 1;
}
```

`if`-Statements können auch abgekürzt werden, mit einer etwas kryptischeren, aber dafür prägnanteren Syntax:

```
( CONDITION ) ? /*mache irgendwas*/ : /*mache etwas anderes*/ ;
```

D.h. für obiges Beispiel könnte man ebenso schreiben:

```
b = ( a < b ) ? b-1 : b;
```

- *Schleifen*

```
for ( INIT; STOP; STEP )
{
    //mache etwas
}

while ( CONDITION )
{
    //mache etwas
}

do
{
    //mache etwas
} while ( CONDITION );
```

Ein Beispiel für einen Schleifendurchlauf wäre:

```
int i=1, c=10;
for (int j = c; j > 0; j--)
{
    i *= j; // bzw.: i = i*j;
}
```

- *switch-case* Manchmal will man verschiedene Fälle schnell abprüfen, die alle eine integer-Zahl als Repräsentanten haben¹. Dafür verwendet man die switch-case-Anweisung:

```
switch ( INTEGER )
{
case 0:
    //mache etwas
case 1:
    //mache etwas anderes
//...
case n:
    //mache etwas tolles
default:
    //mache etwas ganz normales.
}
```

Im folgenden Beispiel wird gleich eine Datenstruktur eingeführt, die bei solchen Anweisungen immer gerne verwendet wird: *Enumerations*. Enumerations sind Aufzählungen, die jedem Wert dieser Aufzählung einen Integerwert zuweisen. Z.B. ist enum Wochentag { Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag }; solch eine Aufzählung. Der Compiler kennt nun die Wochentage und weist ihnen auch die Integerwerte 0-6 zu. Eine switch-case-Anweisung kann sich dieser Aufzählungen direkt bedienen:

```
enum Wochentag {
    Montag,
    Dienstag,
```

¹Eigentlich muss es nur eine ganze Zahl sein, nicht unbedingt eine integer-Zahl, d.h. es können auch alle anderen ganzzahligen Datentypen verwendet werden, z.B. char oder unsigned short

```
Mittwoch,  
Donnerstag,  
Freitag,  
Samstag,  
Sonntag  
};  
//...  
switch(Wochentag)  
{  
case Montag:  
    cout << "Heute ist Montag." << endl;  
    break;  
case Dienstag:  
    //...  
}
```

In dem Beispiel wird auch gleich eine wichtige Fehlerquelle aufgezeigt: Will man die `switch-case`-Anweisung nach der Ausführung eines Falles verlassen, so muss man das Schlüsselwort `break` benutzen. Sonst werden die dahinterliegenden Fälle auch getestet, dies soll manchmal vermieden werden, insbesondere wenn man einen `default`-Fall hat.