C/C++

Das komplette Programmierwissen für Studium und Job

DIRK LOUIS

Markt+Technik

KOMPENDIUM

Einführung Arbeitsbuch Nachschlagewerk

3 Das Programmgerüst – Einführung für Programmieranfänger

Dieses Kapitel ist als Orientierungskapitel für Programmieranfänger gedacht, die noch keinerlei oder nur ganz wenig Erfahrungen im Umgang mit C oder C++ gemacht haben.

Zuerst wird der Frage nachgegangen, was ein Programm eigentlich ist.

Danach schauen wir uns ein einfaches Programm an, das einmal in C und einmal in C++ implementiert ist und anhand dessen die wichtigsten Programmelemente vorgestellt werden.

Im dritten Abschnitt geht es um die Programmerstellung. Für vier verschiedene Compiler und Entwicklungsumgebungen ist beschrieben, wie die Programme kompiliert, gelinkt und ausgeführt werden.

Zum Abschluss werden die wichtigsten Regeln und Begriffe noch einmal zusammengefasst.

Programme, für die im Listing der Dateiname angegeben ist, brauchen Sie nicht abzutippen. Die zugehörigen Quelldateien sind auf der Buch-CD enthalten. Hinweise zur Erstellung der Programme finden Sie im Anhang zur Buch-CD.

3.1 Was ist ein Programm?

Einfach ausgedrückt, ist ein Programm eine Abfolge von Befehlen, die vom Prozessor des Computers nacheinander ausgeführt werden.

In Kapitel 2.3 wurde bereits ausgeführt, dass es dabei ein Kommunikationsproblem zwischen Mensch und Rechner gibt, denn der Computer versteht nur die Befehle aus dem Befehlssatz seines Prozessors – und diese sind zudem noch binär codiert.

Aus diesem Grunde wurden die höheren Programmiersprachen entwickelt, zu denen auch C und C++ gehören. Statt sein Programm als eine Abfolge von binär codierten Maschinenbefehlen aufsetzen zu müssen, kann sich der Programmierer der leichter verständlichen Syntax und der Konstrukte von C und C++ bedienen und einem Compiler die Übersetzung dieser Programme in Maschinenbefehle überlassen.

Ein und dasselbe Programm liegt damit in zwei Erscheinungsformen vor:

- einmal als Quelltext, der in C/C++ formuliert ist,
- einmal als ausführbare Datei, die den Maschinencode aus der Übersetzung des Quelltextes enthält.

Wie wird ein C-Programm als Quelltext aufgebaut?

Am einfachsten wäre es sicherlich, wenn man die C-Anweisungen untereinander eintippen würde und dann die Programmausführung mit der obersten Anweisung beginnen ließe. (Ich werde im Zusammenhang mit C von Anweisungen und im Maschinenkontext von Befehlen sprechen, um beide deutlich zu unterscheiden und anzudeuten, dass eine C-Anweisung komplexer ist als ein Maschinenbefehl – sprich: für die Übersetzung einer C-Anweisung meist mehrere Maschinenbefehle erforderlich sind.)

Nach diesem Modell werden beispielsweise die Batch-Dateien unter DOS, die früheren Basic-Programme oder auch Assembler-Programme ausgeführt. Das Aufsetzen und Warten größerer Programme würde damit allerdings zur Qual. Aus diesem Grunde unterstützen fast alle höheren Programmiersprachen die Strukturierung und Modularisierung des Quellcodes.

Die Unterscheidung in modulare und nichtmodulare Programmierung entspricht im Wesentlichen dem Unterschied zwischen einem Roman und diesem Buch.

- Romane bestehen aus einem einzigen lang gezogenen Fließtext. Wunderbar zu lesen, aber schlecht zum Nachschlagen. Haben Sie einmal versucht, in Peter Høegs Krimi »Fräulein Smillas Gespür für Schnee« die Stelle zu finden, wo Fräulein Elsa Lübing gegenüber Fräulein Smilla ihre Vermutungen über die Expeditionen von 1966 äußert?
- Bücher, die auch zum Nachschlagen geeignet sein sollen, strukturieren ihren Text in betitelte Kapitel und Unterkapitel. An den Kapiteltiteln kann man dann ablesen, welche Informationen in dem Text des Kapitels zu finden sind (vorausgesetzt der Autor versucht nicht, die Kapiteltitel zur Demonstration seiner literarischen Bildung und seines Sprachwitzes zu missbrauchen – wie in den folgenden Kapiteln aus einem – ansonsten von mir sehr geschätzten – Buch zur Windows-Programmierung: »Von Menschen und Mäusen«, »Feuerzangenbowle«, »Eierwärmer«, »Weichzeichnen und Scharfmachen«). Des Weiteren erlaubt die Kapitelstruktur auch die Angabe von Verweisen, wenn für das tiefere Verständnis der in einem Kapitel dargelegten Sachverhalte die Informationen aus einem anderen Kapitel erforderlich sind.

In C entspricht die Kapitelstruktur der Modularisierung durch Funktionen.

Buch	C-Quelltext

Buch, das die Zubereitung eines chinesischen Gerichts beschreibt:
Kapitel 1: Vorbereitung
Kapitel 2: Zutaten schneiden
Kapitel 3: Soße anrühren
Kapitel 4: Fleisch anbraten
Kapitel 5: Soße hinzufügen
Kapitel 6: Servieren
	Programm, das die Zubereitung eines chinesischen Gerichts simuliert:
	`int vorbereiten();`
	`int schneiden(char * gemuese);`
	`int sosse_vorb(char * sosse);`
	`int fleisch_braten(int garzeit);`
	`int sosse_zugeben();`
	`int servieren();`
	`int main()  // Hauptprogramm`

Verweise	Aufrufe

aus Kapitel 5:
Nachdem das Fleisch 2 Minuten pfannengerührt wurde, können Sie die Soße (siehe Kapitel 3) hinzugießen.
	`int sosse_zugeben()`
	`  {`
	`  sosse_vorb(»SuessSauer«);`
	`  ...`
	`  }`

Modularisierung durch Funktionen

In C werden Anweisungen in Funktionen zusammengefasst. In einem C-Programm können beliebig viele Funktionen an beliebigen Stellen im Quelltext definiert werden, wobei in einem guten C-Programm jede Funktion so implementiert ist, dass sie eine spezielle Aufgabe erfüllt. So könnte man beispielsweise eine Funktion namens `printf()` aufsetzen, die eine Zeichenkette auf den Bildschirm ausgibt. Diese Funktion kann dann über ihren Namen aus anderen Funktionen heraus aufgerufen werden. Die Programmausführung springt dann zu den Anweisungen der Funktion `printf()` und kehrt nach deren Abarbeitung in die aufrufende Funktion zurück.

Daraus ergeben sich drei wichtige Vorteile:

- Der Quelltext wird durch die Aufteilung in Funktionen besser lesbar.
- Wann immer eine Ausgabe auf den Bildschirm benötigt wird, braucht man nur noch die Funktion über ihren Namen aufzurufen, anstatt die ganzen Anweisungen, die in der Funktion definiert sind, erneut einzutippen.
- Hat sich ein Fehler in den Anweisungen zur Ausgabe der Zeichenkette eingeschlichen, braucht dieser nur einmal in der Definition der Funktion `printf()` korrigiert zu werden.

C/C++ kennt keine untergeordneten Funktionen

Um noch einmal auf den Vergleich mit den Büchern zurückzukommen: In einem Buch kann man den Text in Kapitel, diese wiederum in Unterkapitel und diese wieder in einzelne Abschnitte mit eigenen Überschriften unterteilen. Kann man nun gleichermaßen in einer Funktion eine weitere untergeordnete Funktion definieren, Funktionen also quasi verschachteln? Man

kann, allerdings nicht in C. In C sind alle Funktionen gleichberechtigt. Eine Funktion kann eine andere Funktion aufrufen, aber sie kann keine untergeordnete Funktion beinhalten. In anderen Sprachen wie z. B. Pascal ist es aber durchaus erlaubt, Funktionen innerhalb von Funktionen zu definieren.

Modularisierung durch Klassen und Quellmodule

C++ bietet durch sein Klassenkonzept viel weitreichendere Möglichkeiten zur Strukturierung von Quelltexten. In Klassen kann man mehrere Funktionen zusammenfassen, man kann Funktionen und Variablen zusammenfassen, man kann Klassen verschachteln (was mit C-Funktionen wie gesagt nicht erlaubt ist). Das soll nun aber nicht heißen, dass man Klassen vornehmlich zur Strukturierung von Quellcode verwendet. Sie sollten Klassen als ein Mittel zur Repräsentation von Objekten (siehe Kapitel 15) und zur Definition eigener Datentypen (siehe Kapitel 8.5) sehen. Die Strukturierung des Quelltextes ist dabei ein sich zwangsläufig einstellender, positiver Nebeneffekt.

C wie C++ erlauben Ihnen auch die Aufteilung Ihres Quelltextes auf mehrere Dateien. Der Inhalt verschiedener Dateien kann mithilfe der `#include`-Direktive zu einer Übersetzungseinheit (siehe Kapitel 25) zusammengefasst werden. Mehrere Übersetzungseinheiten können vom Linker zu einem Programm zusammengebunden werden. Die Aufteilung des Quelltextes in mehrere Dateien und Übersetzungseinheiten dient ebenfalls der Strukturierung des Codes und erleichtert insbesondere die Arbeit an größeren Programmen (gerade auch, wenn mehrere Programmierer an einem Projekt zusammenarbeiten). Der Modularisierung durch Übersetzungseinheiten ist Kapitel 25 gewidmet. Im Moment brauchen Sie sich damit nicht zu belasten; die meisten Beispiele dieses Buches bestehen aus einer einzigen Quelltextdatei.

Programmstart mit main()

Wenn in einem C-Programm alle Anweisungen in Funktionen gekapselt werden und diese Funktionen an beliebiger Stelle im Quelltext definiert werden können, mit welcher Anweisung oder mit welcher Funktion beginnt dann das Programm?

Zu diesem Zweck definiert der C-Standard eine besondere Funktion, die so genannte Eintrittsfunktion. Der Name dieser Funktion ist vorgegeben: Sie heißt `main()`. Jedes funktionsfähige C-Programm muss eine `main()`-Funktion definieren. Die Programmausführung beginnt mit dem Aufruf dieser Funktion und der Ausführung der Anweisungen in der Definition von `main()`. Innerhalb der Funktion `main()` kann man dann andere im Programm definierte Funktionen aufrufen.

Bis dato besteht unser C-Programm lediglich aus einer Ansammlung von Funktionsdefinitionen. Innerhalb der Funktionsdefinitionen stehen die Anweisungen, die die eigentlichen Befehle darstellen.

Diese Anweisungen dienen zum Teil dazu, den Programmfluss zu steuern. Die meisten Anweisungen dienen aber der Verarbeitung von Daten.

Daten

In C-Programmen werden Daten in Form von Variablen oder Konstanten repräsentiert. Bevor man mit Variablen arbeiten, d.h. ihnen Werte zuweisen oder ihre Werte abfragen kann, muss man sie definieren.

Ein C-Programm ist letztlich also eine Ansammlung von Variablen- und Funktionsdefinitionen.

3.2 Ein einführendes Beispiel

Seit dem Erscheinen des Klassikers »Programmieren in C« von Kernighan und Ritchie beginnen die meisten C-Lehrbücher mit einem einfachen Beispielprogramm, das nichts anderes macht, als die Grußworte »Hello, world« auszugeben.

3.2.1 Hello, world in C und C++

In C sieht das »Hello, world«-Programm wie folgt aus:

```
/* HalloWelt.cpp - Ein erstes Programm */
#include <stdio.h>

int main()
  {
  printf("Hello, world\n");
  return 0;
  }
```

Dieses Programm definiert lediglich die Eintrittsfunktion `main()`, in der nur eine einzige Anweisung steht: der Aufruf der Funktion `printf()`. Diese Funktion wurde bereits für uns definiert. Sie ist Teil der Standardbibliothek von C. Um sie benutzen zu können, muss ihre Deklaration in den Quelltext aufgenommen werden. Dies geschieht mithilfe der `#include`-Direktive.

Beachten Sie beim Abtippen, dass C/C++ zwischen Groß- und Kleinschreibung unterscheidet.

Da C++ die Sprache C fast vollständig und ohne große Anpassungen einschließt, ist das obige Programm auch ein absolut korrektes C++-Programm. Was C++ aber auszeichnet, ist die objektorientierte Programmierung, d.h. die Programmierung mit Klassen und Klasseninstanzen. Dies spiegelt sich auch in den mit C++ ausgelieferten Bibliotheken wider. Während in den C-Bibliotheken die Ein- und Ausgabeoperationen in Form von Funktionen implementiert sind, sieht C++ zum gleichen Zwecke spe-

zielle Klasseninstanzen und eigens überladene Operatoren vor. Unter Ausnutzung dieser Konzepte würde das Programm dann wie folgt aussehen:

```
#include <iostream>
using namespace std;

int main()
  {
  cout << "Hello, World\n";
  return 0;
  }
```

E/A in C und C++

Ob man nun aber `printf()` oder `cout` verwendet, macht keinen allzu großen Unterschied. Beide haben ihre Vor- und Nachteile, und es steht Ihnen frei, in einem Programm beide Konzepte durcheinander zu verwenden.

Ich werde es in diesem Buch so halten, dass ich in den ersten Kapiteln, die sich vornehmlich mit den Konzepten beschäftigen, die schon in C vorhanden waren, nur die C-Funktionen verwenden werde. Beispielprogramme, die spezielle Konzepte von C++ erläutern, verwenden dagegen die Klasseninstanzen `cout` und `cin`.

In den Kapiteln 23 und 24 werden die von C und C++ zur Verfügung gestellten Möglichkeiten zur Ein- und Ausgabe bzw. zur String-Behandlung eingehender beschrieben und gegenübergestellt.

3.2.2 Einbindung der Header-Dateien

Um Funktionen (oder auch andere Elemente wie Konstanten oder Klassen) der Laufzeitbibliothek nutzen zu können, müssen die passenden Header-Dateien mithilfe von `#include`-Direktiven eingebunden werden.

Abgesehen davon, dass man wissen muss, welche Header-Datei für welche Funktion eingebunden werden muss, spielt auch die korrekte Syntax der `#include`-Direktiven eine große Rolle.

C versus C++

Die genaue Syntax hängt dabei ganz wesentlich von Ihrem Compiler ab.

C-Compiler

In C endet der Name der Header-Datei immer auf .h – beispielsweise

```
include <stdio.h>
```

C++-Compiler

In C++ haben die Header-Dateien keine Extension mehr:

```
include <iostream>
```

Die alten C-Header-Dateien werden durch ein vorangehendes »c« gekennzeichnet:

```
include <cstdio>
```

Alle Elemente dieser Header-Dateien sind zudem in den Namensbereich `std` eingefasst. Man muss daher nach Auflistung der Header-Dateien diesen Namensbereich einführen:

```
using namespace std;
```

Unabhängig davon können C++-Compiler die C-Header-Dateien auch im alten Stil einbinden (die Funktionen sind dann nicht in den `std`-Namensbereich eingefasst).

Übergangs-Compiler

Alte C++-Compiler verwenden für die speziellen C++-Header-Dateien die gleiche Nomenklatur wie für die C-Header-Dateien:

```
include <iostream.h>
```

Diese Header-Dateien kennen ebenfalls keinen Namensbereich `std`. (Auch die Namen dieser Dateien sind nicht einheitlich, manche Compiler bezeichnen die Header-Datei mit den Stream-Klassen beispielsweise als `stream.h` statt als `iostream.h`.)

Es gibt auch die Möglichkeit, dass ein Compiler die neuen Namen für die Header-Dateien unterstützt, aber nicht den Namensbereich `std`.

Eine Übersicht der Header-Dateien zu den Laufzeitbibliotheken und der Funktionen und Klassen, die in den einzelnen Header-Dateien enthalten sind, finden Sie im Anhang.

3.2.3 Ein umfangreicheres Beispiel

Zur Übung erstellen wir nun ein etwas umfangreicheres Programm, das entscheiden soll, ob (irischer) Whiskey das Gleiche ist wie (schottischer) Whisky.

```
/* Whisky.cpp - Ein erstes C-Programm */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()  {
  char lebenswasser1[20];
  char lebenswasser2[20];

  strcpy(lebenswasser1, "Whiskey");
  strcpy(lebenswasser2, "Whisky");
  if (strcmp(lebenswasser1,lebenswasser2) == 0)
     printf("%s und %s sind das gleiche\n",
                                lebenswasser1,lebenswasser2);
```

```
    else
    printf("%s ist nicht gleich %s \n",
                                lebenswasser1,lebenswasser2);
return 0;
}
```

Ein solcher Programmtext oder Quellcode kann mit jedem Texteditor erstellt werden, der den Text ohne Formatierungszeichen als ASCII-Text abspeichern kann. Bequemer ist es natürlich, den Quelltexteditor einer Entwicklungsumgebung zu benutzen (beispielsweise Turbo C++ oder Visual C++). Nachdem der Quelltext im Editor eingegeben ist und sicherheitshalber schon als Datei abgespeichert wurde, wird der Compiler aufgerufen, um den C++-Quellcode in Binärformat umzuschreiben. Das Ergebnis ist eine Objektdatei mit der Extension .obj. Obwohl das Programm in dieser Datei in maschinenlesbarer Form vorliegt, ist es so noch nicht ausführbar. Erst muss noch zusätzlicher Code in das Programm eingebunden werden. Diese Aufgabe übernimmt der Linker, der automatisch aufgerufen wird, wenn der Compiler das Programm korrekt übersetzen konnte. Das Ergebnis ist eine ausführbare .exe-Datei. Diese kann direkt aufgerufen werden.

Doch nicht immer geht bei der Programmierung alles so glatt.

Während der Compiler den Quelltext in Objektcode übersetzt, prüft er, ob der Quelltext auch den in C++ geltenden Syntaxregeln entspricht. Ist dies nicht der Fall, weiß der Compiler nicht, wie er den Quelltext übersetzen soll, und gibt eine Fehlermeldung aus.

Konnte der Compiler dagegen seine Arbeit erfolgreich beenden, heißt dies, dass ein syntaktisch korrektes C++-Programm erstellt wurde. Ob dieses Programm lauffähig ist, entscheidet allerdings erst der Linker. Beispielsweise ruft obiges Programm eine Funktion mit Namen `printf()` auf. Diese Funktion hat die Aufgabe, die Antwort auf unsere Frage auf den Bildschirm auszugeben. Wo aber ist definiert, wie diese Funktion ihre Aufgabe erfüllt?

Die Funktion `printf()` (wie auch `strcpy()` und `strcmp()`) ist Teil des Standardsprachumfangs von C/C++ und liegt als bereits kompilierter Objectcode in einer der mitgelieferten Bibliotheken vor. Der Linker sucht die entsprechende Bibliothek, in der die Funktion definiert ist, und bindet sie mit der .obj-Datei des Programms zusammen. Das Ergebnis ist ein lauffähiges Programm. Doch ob dieses Programm auch seine Aufgabe erfüllt, ist immer noch nicht sichergestellt. Verhält sich das Programm anders als geplant oder stürzt es gar ab, liegen logische Fehler vor. Solche Fehler sind oft nur schwer zu finden. Um sie zu erkennen und zu lokalisieren, bedient man sich eines Debuggers.

Schritt	Ergebnis
1. Quellcode in Editor eingeben	KEINNAME.cpp
2. Quellcode speichern	DATEINAME.cpp
3. Programm kompilieren	DATEINAME.obj
4. Programm linken	DATEINAME.exe
5. Programm ausführen	

Tabelle 3.1: Ablauf der Programmerstellung

Im folgenden Abschnitt werden wir obiges Programm genauer auf seine einzelnen Elemente und seine Syntax hin untersuchen.

3.3 Ein einführender Programmierkurs

Die wichtigsten Elemente, aus denen praktisch jedes C-Programm zusammengesetzt ist, sind

- Kommentare,
- Datenelemente,
- Operationen auf Daten,
- Konstrukte zur Steuerung des Programmablaufs,
- Funktionen,
- Präprozessor-Direktiven.

3.3.1 Kommentare

Umfangreichere Kommentare beginnen stets mit /* und enden mit */. Alles dazwischen wird vom Compiler ignoriert.

Das hat zur Folge, dass sich die Kommentar-Marker nicht verschachteln lassen. Ein Nachteil, der besonders beim Debuggen deutlich wird.

C versus C++

C++ stellt darüber hinaus eine weitere, sehr bequeme Art der Kommentierung zur Verfügung. Mittels // können Sie dem Compiler mitteilen, dass der Rest der aktuellen Zeile nur noch Kommentar beinhaltet.

```
#include <stdio.h>

/* Eintrittsfunktion */
int main()
  {
  printf("Hello, world\n");   // Ausgabe
  return 0;
  }
```

Diese Form der Auskommentierung ist so praktisch und übersichtlich, dass sie auch nachträglich in den ANSI-C-Standard aufgenommen wurde.

Wenn Sie mit einem älteren C-Compiler arbeiten, kann es sein, dass dieser die //-Kommentare in den Beispiellistings nicht akzeptiert. Schreiben Sie die Kommentare dann in die / */-Syntax um oder löschen Sie die Kommentare.*

3.3.2 Datenelemente

Datenelemente dienen dazu, Ihre Daten (Zahlen, Wörter etc.) aufzunehmen und zu repräsentieren. Man unterscheidet Konstanten und Variablen. Während eine Konstante durch einen festen (konstanten) Wert gegeben ist, kann sich der Wert einer Variablen im Laufe des Programms ändern (ist also variabel). Dies stellt aber für das Programm (beziehungsweise den Rechner, der das Programm ausführt) ein organisatorisches Problem dar.

Die M8-50-Schraube

Nehmen wir einmal an, Sie verfügen wie ich über leidliche handwerkliche Fähigkeiten (gepaart mit großen handwerklichen Ambitionen). Des Weiteren nehmen wir an, Sie haben ein paar Schrauben mit M8-Gewinde und 50 mm lang, die Sie in einer kleinen Schublade verwahren. Wenn neue M8-50-Schrauben hinzukommen, legen Sie diese dazu. Wenn Sie Schrauben zum Aufbau eines Regals benötigen, entnehmen Sie sie der Schublade. Wenn Sie einfach nur wissen wollen, wie viele M8-50-Schrauben Sie noch haben, schauen Sie in die Schublade rein und zählen nach.

Der Inhalt der Schublade ist also variabel – in dem Sinne, dass sich die Anzahl der Schrauben ändern kann.

Wir können dies auf unser Programm übertragen. Statt einer Schublade verwendet der Rechner Speicherzellen. Statt Schrauben legt er in der Speicherzelle Zahlen ab (allgemeiner ausgedrückt, Daten eines bestimmten Datentyps). Der Wert dieser Zahl entspricht der Anzahl der Schrauben und kann sich im Laufe des Programms ändern.

M8-50 und M16-120

Mittlerweile haben Sie sich auch einen Vorrat an M16-Schrauben zugelegt, die 120 mm lang sind. Sie verwahren diese in einer eigenen Schublade, die aber natürlich größer sein muss als die Schublade für die 50-mm-Schrauben.

Übertragen auf unser Programm bedeutet dies, dass der Speicherbereich, der für eine Variable bereitgestellt wird, sich danach richtet, welche Art von Objekten darin abgespeichert wird (ganze Zahlen, Gleitkommazahlen, Zeichen – in C spricht man in diesem Zusammenhang von Datentypen).

Schraubenmagazine

Irgendwann haben Sie eine Sammlung von 20 bis 30 verschiedenen Schraubenarten und für jede Schraubenart ihre eigene Schublade. Um nicht die Übersicht zu verlieren, haben Sie die einzelnen Schubladen natürlich beschriftet, damit Sie die passende Schraube immer schnell finden.

Aus dem gleichen Grund geben Sie jeder Variablen einen Namen.

Schrauben-Beispiel	Variable
Schublade	Speicherzelle
Schraubenzahl	Wert der Variablen
Schraubenart und -größe	Datentyp
Beschriftung	Variablenname

Tabelle 3.2: Analogie Schrauben – Variablen

Wenn Sie also eine neue Variable brauchen, um in dieser einen Wert zu speichern, ist das, als würden Sie eine neue Schublade einrichten – nur dass Sie viel Arbeit an den Compiler und den Rechner delegieren.

Sie selbst brauchen die Variable nur zu deklarieren:

```
int n;
```

Dabei geben Sie der Variablen einen eindeutigen Namen (entspricht der Beschriftung einer Schublade) und teilen dem Compiler mit, welche Art von Objekten in der Variablen verwahrt wird und wie groß deren Speicherbedarf ist.

In obigem Fall sollen in der Variablen n natürliche Zahlen abgespeichert werden, wir haben daher den Datentyp »`int`« gewählt. Da ganze Zahlen (Integer-Werte) in 32-Bit-Umgebungen durch eine 32-stellige binäre Zahl kodiert werden, also aus 4 Byte bestehen, reserviert der Compiler für eine `int`-Variable auch genau 4 Byte im Arbeitsspeicher.

Bei der Deklaration einer Variablen geschieht also erst einmal zweierlei:

- In Abhängigkeit davon, welche Werte die Variable aufnehmen soll (festgelegt durch den Datentyp; in obigem Beispiel also »`int`«), wird ein Speicherbereich für die Variable reserviert.
- Die Adresse des Speicherbereichs wird mit dem Variablennamen verbunden. Während wir bei der Programmierung unter einer Variablen meist ihren zugeordneten Wert (Speicherinhalt) verstehen, ist die Variable für den Computer eine Adresse im Speicherraum.

Der nächste Schritt besteht nun darin, der Variablen `n` einen Wert zuzuweisen, d. h. eine natürliche Zahl in der Speicherzelle abzulegen:

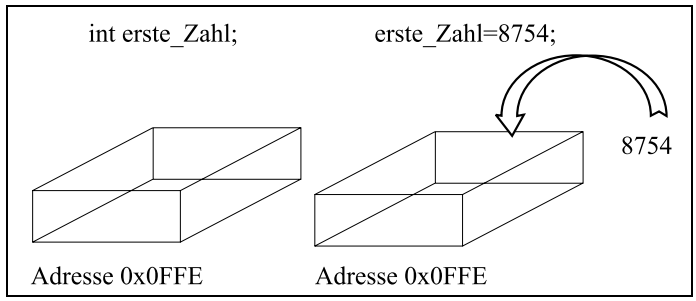
```
n = 10;
```

Eine andere Möglichkeit wäre, einer Variablen den Wert einer anderen Variablen zuzuordnen:

```
int n1;
int n2;

n1 = 10;
n2 = n1;
```

Abbildung 3.1: Variablendefinition



Danach würde in beiden Variablen der Wert 10 stehen. Da es in den meisten Fällen nicht notwendig ist, zwischen der Variablen als einer reservierten Speicherzelle und dem Inhalt dieser Speicherzelle zu unterscheiden, kann man auch sagen: Die Variablen `n1` und `n2` sind beide gleich `10`.

3.3.3 Operationen auf Daten

Neben der Zuweisung von Werten gibt es noch andere Operationen auf Daten. Dabei legt der Datentyp (siehe Kapitel 6.1) bereits fest, welche Operationen auf die Daten angewendet werden dürfen. Integer-Werte lassen sich zum Beispiel addieren, subtrahieren, dividieren und multiplizieren:

```
1:  /* mult.cpp Multiplikation */
2:  #include <stdio.h>

3:  int main()
4:    {
5:    int n1, n2;         // Variablendeklaration

6:    n1 = 10;            // Initialisierung von n1
7:    n2 = n1;            // Initialisierung von n2
8:    n1 = n1 * n2;       // Multiplikation

9:    printf("Ergebnis der Multiplikation = %d\n",n1);
10:   return 0;
11: }
```

Die Nummerierung der Zeilen dient nur der besseren Orientierung im Listing; sie ist nicht Bestandteil des Programmcodes.

Ausgabe:

```
Ergebnis der Multiplikation = 100
```

Analyse Die Variablendeklaration wurde hier etwas vereinfacht. Für Variablen gleichen Datentyps – hier `int` – genügt es, den Datentyp an den Anfang zu stellen und danach die Variablen durch Kommata getrennt aufzulisten. Abgeschlossen wird die Deklaration wiederum mit einem Semikolon.

In der Zeile 6 wird der Variablen `n1` der Wert `10` zugewiesen. In der darauf folgenden Zeile wird der Variablen `n2` der Wert der Variablen `n1` zugewiesen. Beide Variablen sind jetzt also gleich `10`.

In Zeile 8 steht die Variable `n1` sowohl links wie auch rechts vom Gleichheitszeichen! Der Compiler löst dies so auf, dass er zuerst die rechte Seite berechnet. Nach der Berechnung wird das Ergebnis der Variablen `n1` zugewiesen (jetzt erst wird der Speicherinhalt von `n1` überschrieben). In der 8. Zeile werden also nacheinander zwei Operationen ausgeführt: Multiplikation (Operator *) und Zuweisung (Operator =).

Schließlich wird die Funktion `printf()` aufgerufen, um das Ergebnis anzuzeigen.

Damit haben Sie bereits die wirklich grundlegenden Konzepte von C++ – Daten und Operationen auf Daten – kennen gelernt. Viel mehr bedarf es nicht, um eine so erfolgreiche und mächtige Sprache wie C oder C++ zu schaffen bzw. zu beherrschen. Allerdings wäre es recht mühsam, wenn man jedes Programm Anweisung für Anweisung niederschreiben müsste. Obiges Programm berechnet das Quadrat von 10. Wie aber, wenn wir ein Programm schreiben möchten, das uns die Quadratzahlen von 1 bis 10 ausgeben soll?

```
int main()
  {
  int n1;

  n1 = 1;
  n1 = n1 * n1;
  printf("Das Quadrat von 1 ist = %d\n",n1);

  n1 = 2;
  n1 = n1 * n1;
  printf("Das Quadrat von 2 ist = %d\n",n1);

  n1 = 3;
  n1 = n1 * n1;
  printf("Das Quadrat von 3 ist = %d\n",n1);

  n1 = 4;
  n1 = n1 * n1;
  printf("Das Quadrat von 4 ist = %d\n",n1);

  ...

  return 0;
  }
```

Analyse

Das Programm wurde zwar noch etwas vereinfacht, indem die zweite Variable `n2` weggelassen wurde, aber es wird deutlich, dass es keinen Spaß macht, auf diese Weise ein Programm zu schreiben, das die Quadratzahlen von 1 bis 1000 berechnet. Die Lösung liegt darin, den Programmablauf in einer Schleife auszuführen.

Die Ellipse »...« ist natürlich nicht Teil des Programms, sondern soll Ihnen nur anzeigen, dass hier noch die entsprechenden Anweisungen für die Quadratzahlen von 5 bis 10 folgen sollen.

3.3.4 Programmablauf

Verschiedene Konstrukte dienen dazu, den Programmablauf zu regeln. Neben dem linearen Ablauf gibt es Verzweigungen und Schleifen. Die `for`-Schleife wird häufig verwendet, wenn der Programmierer bereits weiß, wie oft die Schleife durchlaufen werden soll:

```
1:  /* forschleife.cpp   for-Schleife  */
2:  #include <stdio.h>

3:  int main()
4:   {
5:   int loop;
6:   int n1;

7:   for(loop = 1; loop <= 10; loop++)
8:     {
9:     n1 = loop * loop;
10:    printf("Das Quadrat von %d ist = %d\n",loop, n1);
11:    }
12:  return 0;
13:  }
```

Ausgabe:

```
Das Quadrat von  1 ist = 1
Das Quadrat von  2 ist = 4
Das Quadrat von  3 ist = 9
Das Quadrat von  4 ist = 16
Das Quadrat von  5 ist = 25
Das Quadrat von  6 ist = 36
Das Quadrat von  7 ist = 49
Das Quadrat von  8 ist = 64
Das Quadrat von  9 ist = 81
Das Quadrat von 10 ist = 100
```

Analyse In Zeile 7 beginnt die Schleife, eingeleitet durch das Schlüsselwort `for`. In der nachfolgenden Klammer wird festgelegt, wie oft die Schleife durchlaufen wird. Dazu bedarf es einer Schleifenvariablen, die ich »`loop`« genannt habe. Bei Eintritt in die Schleife muss der Schleifenvariablen ein Anfangswert zugewiesen werden. Dies geschieht in der Anweisung:

```
loop = 1;
```

Dahinter ist die Abbruchbedingung formuliert, die festlegt, wann die Schleife verlassen wird:

```
loop <= 10;
```

»<=« ist ein Vergleichsoperator, der überprüft, ob ein Wert (hier der Wert der Schleifenvariablen `loop`) kleiner oder gleich einem zweiten Wert (hier 10) ist.

Solange die Bedingung erfüllt ist, d.h. solange die Schleifenvariable `loop` kleiner oder gleich 10 ist, wird die Schleife ausgeführt. Die dritte Anweisung wird nach Abarbeitung des Schleifenkörpers ausgeführt. Der Schleifenkörper ist durch die geschweiften Klammern in den Zeilen 8 und 11 markiert. Beim ersten Schleifendurchgang hat die Schleifenvariable `loop` noch den Anfangswert 1. Folglich bekommt `n1` den Wert `1 * 1 = 1` zugewiesen. Das Ergebnis der Berechnung wird in Zeile 10 ausgegeben. Am Ende der Schleife wird die dritte Anweisung

```
loop++
```

ausgeführt.

Bitte beachten Sie, dass die letzte Anweisung im Schleifenkopf nicht mit einem Semikolon abgeschlossen wird.

Die Anweisung `loop++` ist identisch zu `loop = loop + 1`. Sie ist also nur eine kürzere Schreibweise für die Addition von 1, auch als Inkrement bezeichnet. Danach springt das Programm zurück an den Schleifenanfang. Die Zuweisung

```
loop = 1;
```

wird nun nicht mehr ausgeführt, wohl aber die Überprüfung der Abbruchbedingung. Da nach jedem Schleifendurchgang die Variable `loop` um 1 erhöht wird, wird `loop` nach dem zehnten Durchgang den Wert `11` haben. Die Überprüfung der Abbruchbedingung ergibt, dass `loop` nun größer als 10 ist, und die Schleife wird verlassen. Das Programm wird in Zeile 12 fortgesetzt.

Wichtig ist, dass die Schleife auch wirklich wieder verlassen wird. Betrachten Sie folgende Schleife:

```
for(loop=1; loop <= 10; loop--)
  {
  ...
  }
```

Hier wird die Schleifenvariable `loop` nach jedem Schleifendurchgang dekrementiert, das heißt um 1 vermindert. Das heißt, `loop` wird nie größer als 10 werden und die Schleife wird nie verlassen! Das Gleiche kann einem passieren, wenn man die Schleifenvariable in der Schleife auf ungünstige Art verändert:

```
for(loop=1; loop <= 10; loop++)
  {
  ...
  loop--;
  }
```

Hier wird die Variable `loop` zwar nach jedem Durchgang ordnungsgemäß um 1 erhöht; da `loop` aber zuvor in der Schleife um 1 erniedrigt wurde, heben sich die beiden Anweisungen gegeneinander auf. Die Variable `loop` hat bei Überprüfung der Abbruchbedingung immer den Wert 1, weshalb die Schleife nicht verlassen wird.

Solche Endlosschleifen sind ein häufiges Beispiel für logische Fehler innerhalb eines syntaktisch korrekten Programms.

Wir haben nun gesehen, wie sich der Programmieraufwand mittels einer `for`-Schleife wesentlich verringern kann. Es gibt aber noch weitreichendere Möglichkeiten, ein Programm zu strukturieren. Dazu gehört beispielsweise die Kapselung von Teilproblemen in Funktionen.

3.3.5 Funktionen

Funktionen bestehen aus einem Funktionskopf und einem Funktionskörper. Der Funktionskörper kapselt die zur Funktion gehörenden Anweisungen in geschweiften Klammern. Der Funktionskopf deklariert die Funktion und legt die Schnittstelle zur Außenwelt fest.

Eine Funktion zur Berechnung des Quadrats könnte wie folgt aussehen:

```
int quadrat(int n)    // Funktionskopf
   {                  // Beginn des Funktionskörpers
   int ergebnis;

   ergebnis = n * n;
   return (ergebnis);
   }                  // Ende des Funktionskörpers
```

Es ist sinnvoll, den Funktionsnamen so zu wählen, dass daraus der Gebrauch der Funktion direkt ersichtlich ist. Hinter dem Funktionsnamen folgt in Klammern die Liste der Parameter, die der Funktion übergeben werden. Im obigen Fall wird der Funktion nur ein Parameter übergeben. Um den Parameter aufzunehmen, wird für die Funktion eine Integer-Variable `n` eingerichtet. Eine weitere Variable `ergebnis` wird eingerichtet, um das berechnete Quadrat aufzunehmen. Mittels der Anweisung

```
return (ergebnis);
```

wird das Ergebnis der Berechnung als Wert der Variablen `ergebnis` zurückgeliefert. Das Schlüsselwort `int` vor dem Funktionsnamen gibt an, dass es sich bei dem von der Funktion zurückgelieferten Wert um einen Integer-Typ handelt.

Den Informationsfluss in die Funktion hinein und aus ihr heraus wollen wir uns noch einmal am vollständigen Programm gründlicher anschauen:

```
1:  /* FUNC_A.CPP Funktionen */
2:  #include <stdio.h>

3:  int quadrat(int n)    // Funktionsdeklaration
```

```
4:   {
5:   int ergebnis;

6:   ergebnis = n * n;
7:   return (ergebnis);   // Funktion liefert Ergebnis zurueck
8:   }

9:  int main()
10:  {
11:  int n_main;

12:  n_main = 10;
13:  n_main = quadrat(n_main);           // 10 * 10   = 100
14:  printf("1. Quadrat = %d\n",n_main);

15:  n_main = quadrat(n_main);           // 100 * 100 = 10000
16:  printf("2. Quadrat = %d\n",n_main);
17:  return 0;
18:  }
```

Ausgabe:

```
1. Quadrat = 100
2. Quadrat = 10000
```

Analyse

Um keine Verwirrung zu stiften, wurden die Variablennamen umbenannt.

In Zeile 12 wird der Variable `n_main` der Wert 10 zugewiesen.

In Zeile 13 erfolgt der erste Aufruf der Funktion `quadrat()`. Dabei wird der Wert der Variablen `n_main` an die Variable `n` der Funktion `quadrat()` übergeben. Physikalisch gesehen bedeutet dies, dass der Inhalt der Speicherzelle `n_main` in die Speicherzelle der Funktionsvariablen `n` kopiert wird. Das Programm springt jetzt in Zeile 4 und wird dort fortgesetzt. Wenn die Funktion beendet wird, springt das Programm zurück in Zeile 13, von wo aus die Funktion aufgerufen wurde. Der Wert der Variablen `ergebnis` wird der Variablen `n_main` zugewiesen.

Wenn Sie detaillierter nachvollziehen wollen, was in dem Programm geschieht, können Sie es in einem Debugger schrittweise ausführen und sich die Inhalte der Variablen dabei anzeigen lassen oder das folgende Programm ausführen.

```
/* FUNC_B.CPP Funktionen */
#include <stdio.h>

int quadrat(int n)    // Funktionsdeklaration
  {
  int ergebnis;

  printf("\t Dies ist die Funktion quadrat()\n");
  printf("\t n = %d\n",n);
  ergebnis = n * n;
  printf("\t ergebnis = %d\n",ergebnis);
  return (ergebnis);
```

```
  }

int main()
  {
  int n_main;

  printf("Dies ist die Funktion main()\n");
  n_main = 10;
  printf("n_main = %d\n",n_main);
  n_main = quadrat(n_main);                // 10 * 10   = 100
  printf("Dies ist die Funktion main()\n");
  printf("n_main = %d\n",n_main);
  n_main = quadrat(n_main);                // 100 * 100 = 10000
  printf("Dies ist die Funktion main()\n");
  printf("n_main = %d\n",n_main);
  getchar();
  return 0;
  }
```

Ausgabe:

```
Dies ist die Funktion main()
n_main = 10
    Dies ist die Funktion quadrat()
    n = 10
    ergebnis = 100
Dies ist die Funktion main()
n_main = 100
    Dies ist die Funktion quadrat()
    n = 100
    ergebnis = 10000
Dies ist die Funktion main()
n_main = 10000
```

Es ist angebracht, an dieser Stelle noch zwei Begriffe klarzustellen:

Definition

Parameter — Als *Parameter* einer Funktion werden einfach diejenigen Variablen der Funktion bezeichnet, die innerhalb der Klammern im Funktionskopf deklariert werden (im Gegensatz zu den Variablen, die zu Beginn des Funktionskörpers deklariert werden).

Argumente — Die Werte, die beim Funktionsaufruf an die Parameter übergeben werden, nennt man auch *Argumente*.

Analyse In den Zeilen 13 und 15 des Programms *FUNC_A.CPP* wird also der Wert der Variablen `n_main` als Argument an die Parametervariable `n` der Funktion `quadrat()` übergeben.

Die Eintrittsfunktion

Wie Sie sicherlich bemerkt haben, haben wir schon die ganze Zeit von Funktionen Gebrauch gemacht, denn

```
int main()
  {
  ...
  }
```

ist ebenfalls eine Funktion. Die Funktion `main()` ist insofern jedoch ausgezeichnet, als sie die Haupt- und Eintrittsfunktion jedes Programms darstellt. Das heißt, jedes C-Programm beginnt mit der `main()`-Funktion (und nicht etwa in der ersten Zeile des Quelltextes!). Daraus folgt, dass jedes ausführbare Programm eine Funktion mit Namen `main` haben muss!

Obige Feststellung gilt nur für Textbildschirm-Anwendungen. Windows-Anwendungen beginnen mit der Eintrittsfunktion `WinMain()`.

Funktionen sind in C/C++ sehr gebräuchlich. Zum einem kann man Funktionen im Programm einmalig definieren und braucht sie dann bei Bedarf nur noch aufzurufen (wie oben gesehen), zum anderen bietet es sich an, Sammlungen von Funktionen in spezielle Dateien auszulagern und anderen Programmen zur Verfügung zu stellen. Dabei ist zwischen Include-Dateien und Bibliotheken zu unterscheiden. Include-Dateien beinhalten normalen Quelltext und werden vom Compiler in das aufrufende Programm einkopiert, Bibliotheken beinhalten bereits übersetzten Objektcode und werden vom Linker eingebunden.

Das Programm *FUNC_C.CPP* benutzt beispielsweise die Funktion `quadrat()` aus der Include-Datei *QUADR.BIB*:

```
/*  Quadr.bib  */
int quadrat(int n)
 {
 int ergebnis;

 ergebnis = n * n;
 return (ergebnis);
 }
```

Das dazugehörige Hauptprogramm (und nur dieses brauchen Sie aufzurufen und auszuführen):

```
/* FUNC_C.CPP Funktionen */
#include <stdio.h>
#include "quadr.bib"

int main(int argc, char **argv)
  {
  int n_main;

  n_main = 10;
  n_main = quadrat(n_main);           // 10 * 10   = 100
  printf("1. Quadrat = %d\n",n_main);

  n_main = quadrat(n_main);           // 100 * 100 = 10000
```

```
printf("2. Quadrat = %d\n",n_main);
getchar();
return 0;
}
```

Die Einbindung von Include-Dateien führt uns zu den Präprozessor-Direktiven.

3.3.6 Präprozessor-Direktiven

Betrachten wir die 3. Zeile des Beispielprogramms *FUNC_C.CPP*:

```
#include "quadr.bib"
```

`#include` ist eine Präprozessor-Direktive (Präprozessor-Direktiven sind an dem vorangestellten Doppelkreuz # erkennbar), die den Compiler anweist, die in Hochkommata oder eckigen Klammern angegebene Datei an die entsprechende Stelle in den Quelltext zu kopieren. Häufig benutzte Funktionen lassen sich so Ihren Programmen bequem einverleiben. Steht der Dateiname in Hochkommata, sucht der Compiler nur im aktuellen Verzeichnis nach der Datei, während er im Fall von eckigen Klammern den Include-Pfad durchsucht (Einstellung des Compilers).

Nach der Kopieraktion wird das Programm normal übersetzt.

Bibliotheken

Sind die in der Include-Datei stehenden Funktionen sehr umfangreich, führt dies zu einer beträchtlichen Aufblähung Ihres Quelltextes und verlängert die Kompilierung. Dies ist besonders ärgerlich, wenn in der Include-Datei viele Funktionen stehen, die Ihr Programm gar nicht benutzt. Einen Ausweg daraus bieten die Bibliotheken. In Bibliotheken sind die Funktionen in kompilierter Form abgespeichert. Solche Dateien können selbstverständlich nicht mehr als Include-Dateien in den Quelltext hineinkopiert werden, da sie binär codiert sind und vom Compiler nicht verstanden würden. Bibliotheken werden stattdessen vom Linker in das Programm eingebunden. Dabei ergibt sich jedoch ein Problem. Wenn wir die Funktion `quadrat()` als OBJ-Datei vom Linker einbinden lassen, statt sie als Include-Datei einzukopieren, stolpert der Compiler bei der Bearbeitung des Hauptprogramms darüber, dass er auf einen Namen (nämlich »`quadrat`«) stößt, der ihm gänzlich unbekannt ist. Was der Compiler braucht, ist eine Deklaration des Namens, der ihm zum einen angibt, dass es sich um eine Funktion handelt, und zum anderen, wie viele Parameter der Funktion übergeben werden und welchen Typ diese Parameter sowie der Rückgabewert haben. Im Prinzip also den vollständigen Funktionskopf:

```
int quadrat(int);
```

Diese Deklaration können Sie entweder an den Anfang Ihres Programmes schreiben oder in eine Header-Datei (*Headername.h*), die dann mit einem Include-Befehl in das Programm kopiert wird. Der Vorteil ist, dass nur noch Funktionsdeklarationen kopiert und kompiliert werden, während der umfangreichere Funktionskörper erst in das endgültige Programm eingebunden wird.

Damit wäre unser Streifzug durch die verschiedenen Elemente eines C-Programms schon fast beendet. Wahrscheinlich werden Sie einige Konzepte als ungewohnt oder schwierig empfinden. Ihre Kenntnisse werden sich jedoch bald festigen und Sie werden schnell an Sicherheit gewinnen. Mit wachsender Erfahrung werden Ihre Programme umfangreicher und leider auch trotz guten Programmierstils immer schwerer verständlich. C bietet daher die Möglichkeit, Programme mit Kommentaren zu versehen.

Bevor wir nun in den nachfolgenden Kapiteln tiefer in die C-Programmierung einsteigen, kehren wir noch einmal zu unserem anfänglichen Programm zurück, das wir nun schon wesentlich besser verstehen:


```
1:  /* Whisky.cpp - Ein erstes C-Programm */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:  #include <string.h>

5:  int main()
6:    {
7:    char lebenswasser1[20];
8:    char lebenswasser2[20];

9:   strcpy(lebenswasser1, "Whiskey");
10:  strcpy(lebenswasser2, "Whisky");
11:  if (strcmp(lebenswasser1,lebenswasser2) == 0)
12:     printf("%s und %s sind das gleiche\n",
                                lebenswasser1,lebenswasser2);
13:     else
14:     printf("%s ist nicht gleich %s \n",
                                lebenswasser1,lebenswasser2);
15:  return 0;
16:  }
```

Analyse

In Zeile 1 steht ein kurzer Kommentar zu Dateiname und Programm.

In Zeile 2 bis 5 werden verschiedene Header-Dateien aufgerufen. Im Prinzip brauchen wir von diesen Header-Dateien nur `<string.h>` und `<stdio.h>`, die die Deklarationen zu den im Programm verwendeten Funktionen `strcpy()`, `strcmp()` und `printf()` beinhalten. Die Funktion `strcpy()` dient dazu, Zeichenketten (Strings) zu kopieren, `strcmp()` vergleicht zwei Zeichenketten und liefert als Ergebnis `0` zurück, wenn beide Zeichenketten identisch sind, und `printf()` schließlich wird dazu verwendet, Strings auf den Bildschirm auszugeben.

In Zeile 5 wird die Hauptfunktion `main()` aufgerufen.

In den Zeilen 7 und 8 werden die vom Programm benötigten Variablen definiert. Die Variablen `lebenswasser1` und `lebenswasser2` können jeweils 20 Elemente vom Typ `char` aufnehmen (die einzelnen Typen werden in den nachfolgenden Kapiteln genauer besprochen).

In den Zeilen 9 und 10 werden den beiden String-Variablen Textstrings zugewiesen (da wir dem Computer leider keine zwei Gläser mit irischem

Whiskey und schottischem Whisky einschenken können, legen wir einfach zwei String-Variablen an und weisen diesen die beiden unterschiedlichen Bezeichnungen zu).

In Zeile 11 muss sich der Computer dann damit begnügen, die beiden Bezeichnungen statt der beiden Whisky-Sorten zu vergleichen. Glücklicherweise kommt er dennoch zu dem richtigen Ergebnis, denn die Funktion `strcmp()` stellt natürlich fest, dass die beiden Strings nicht identisch sind und liefert einen Wert ungleich Null zurück. Aus diesem Grunde springt das Programm direkt in Zeile 14 (in Kapitel 11.1 wird die `if`-Verzweigung ausführlicher beschrieben).

In Zeile 14 wird die Funktion `printf()` aufgerufen. Diese kopiert den Inhalt der beiden Variablen `lebenswasser1` und `lebenswasser2` in den übergebenen Text; und zwar anstelle der Platzhalter `%s`.

Danach wird das Programm beendet.

3.3.7 Die Funktion printf()

Die Funktion `printf()` dient zur Textausgabe unter C/C++. Sie ist fester Bestandteil der Laufzeitbibliothek von C (ihre Deklaration steht in der Header-Datei *stdio.h*).

Anmerkung Da C++ die Laufzeitbibliothek von C vollständig übernommen hat, können Sie `printf()` auch in C++-Programmen zur Ausgabe von Text auf den Bildschirm verwenden (Gleiches gilt für die mit `printf()` verwandten Funktionen zum Einlesen von der Tastatur (`scanf`) und zur Datei-Ein-/Ausgabe [`fprintf`, `fputs`, `fscanf`, `fgets`]). Darüber hinaus verfügt C++ aber auch über eigene Möglichkeiten der Ein- und Ausgabe. Da diese Formen der Ein- und Ausgabe an spezielle C++-Klassen gebunden sind und folglich auch nicht unter C verwendet werden können, werden wir uns mit ihnen erst später beschäftigen (siehe Kapitel 24).

Zurück zur Funktion `printf()`. Ihr Aufruf sieht folgendermaßen aus:

```
printf(FORMATSTRING, ARGUMENT1, ARGUMENT2, ...);
```

`FORMATSTRING` stellt dabei eine Zeichenkette in Hochkommata dar, die eine beliebige Anzahl von Platzhaltern für Argumente von Variablen enthalten kann. Für jeden Platzhalter muss ein Argument nachfolgen, zum Beispiel also eine Variable, deren Wert dann anstelle des Platzhalters ausgegeben wird. Zu beachten ist dabei, dass die Typspezifikation des Platzhalters und der Typ der Variablen ineinander konvertierbar sind (siehe zum Beispiel Programm *Interpret.cpp* aus 6.1.3). Die Platzhalter haben folgendes Aussehen:

```
% [FLAG] [BREITE] [.PRÄZISION] [GRÖSSE] TYP
```

Dabei sind die Spezifikationen in Klammern optional. Welche Werte Sie angeben können, entnehmen Sie den nachfolgenden Tabellen.

Flag	Ausgabe
-	Ordnet die Ausgabe linksbündig im spezifizierten Feld an.
+	Zeigt das Vorzeichen eines Werts an.
Leerstelle	Zeigt eine Leerstelle vor der Ausgabe an, wenn der Wert positiv ist; ansonsten wird ein Minuszeichen angezeigt.
#	Konvertierung des Arguments in 'alternativer Form'.

Tabelle 3.3: FLAG

BREITE: Gibt die minimale auszugebende Zeichenzahl an.

.PRÄZISION: Gibt die gewünschte Genauigkeit in den Nachkommastellen an (für `float`- und `double`-Werte).

Size	Standardgröße des Arguments
N	near-Zeiger
F	far-Zeiger
h	short int
l	long
L	long double

Tabelle 3.4: GRÖSSE

Kategorie	Typzeichen	Ausgabe
Zeichen	c	Einzelne Zeichen
	d	Vorzeichenbehafteter dezimaler Integer
	i	Vorzeichenbehafteter dezimaler Integer
	o	Vorzeichenloser oktaler Integer
	u	Vorzeichenloser dezimaler Integer
	x	Vorzeichenloser hexadezimaler Integer
	X	Vorzeichenloser hexadezimaler Integer (verwendet Großbuchstaben)
Zeiger	p	Gibt Zeiger in hexadezimaler Form aus
Zeiger auf int	n	Speichert an der Position, auf die das Eingabeargument zeigt, die Anzahl der bis jetzt ausgegebenen Zeichen
Real	f	Vorzeichenbehaftete Werte im Format [-]dddd.dddd
	e	Vorzeichenbehaftete Werte in wissenschaftlicher Schreibweise
	E	Vorzeichenbehaftete Werte in wissenschaftlicher Schreibweise

Tabelle 3.5: TYP

Tabelle 3.5: TYP (Forts.)

Kategorie	Typzeichen	Ausgabe
	g	Vorzeichenbehaftete Werte im Format f oder e, je nach Wert und Genauigkeit
	G	Vorzeichenbehaftete Werte im Format f oder e, je nach Wert und Genauigkeit
Zeiger auf Zeichenketten	s	Zeigt die Zeichen an, bis der Endbegrenzer erreicht ist

Schließlich gibt es noch eine Reihe von Escape-Sequenzen, mit denen bestimmte Sonderzeichen für Zeilenumbrüche und andere spezielle Zeichen ausgegeben werden können. Nachfolgende Tabelle gibt diese Sequenzen an:

Tabelle 3.6: Escape-Zeichen

Sequenz	Dezimalwert	Hex-Wert	Aufgabe
\a	7	0x07	Ton
\b	8	0x08	Backspace
\f	12	0x0C	Seitenvorschub
\n	10	0x0A	Neue Zeile
\r	13	0x0D	Wagenrücklauf
\t	9	0x09	Horizontaler Tabulator
\v	11	0x0B	Vertikaler Tabulator
\\	92	0x5C	Backslash
\'	44	0x2C	Einfaches Anführungszeichen
\"	348	0x08	Doppeltes Anführungszeichen
\?	63	0x3F	Fragezeichen
\ooo			1 bis 3 Stellen für oktale Zahlen
\Xhhh		oxhhh	Hexadezimaler Wert
\xhhh		oxhhh	Hexadezimaler Wert

Im nachfolgenden Listing werden die wichtigsten Einsatzmöglichkeiten der Formatierungsoptionen für die `printf`-Funktion vorgestellt:

```
/* printf.cpp   Daten ausgeben */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
```

```
{
int     i1 = 1000;
int     i2 = -1;
float   f1 = 3.1416;
char    ch = 'A';
char    str[2] = "A";

printf(" %d  \t %d  \t %f    \t %c \t %s\n"
                                        ,i1,i2,f1,ch,str);
printf(" %3d \t %3d \t %3.1f \t %c \t %s\n"
                                        ,i1,i2,f1,ch,str);
printf(" %03d\t %03d\t %03.1f \t %c \t %s\n"
                                        ,i1,i2,f1,ch,str);
return 0;
}
```

Ausgabe:

```
1000   -1   3.141600     A  A
1000   -1  3.1  A  A
1000 -01 3.1  A  A
```

3.3.8 scanf() und wie man es vermeiden kann

Die Funktion `scanf()` ist das Pendant zu `printf()`. Mittels `scanf` können Eingaben von der Tastatur aus eingelesen und an Variablen übergeben werden. Die Syntax entspricht weitgehend der Syntax von `printf()`. Allerdings erwartet `scanf()` als Argumente Adressen. Um die Adresse einer Variablen zu bekommen, benutzt man den Adressoperator `&`, der direkt vor den Variablennamen gestellt wird. Der Compiler weiß dann, dass nicht der Wert der Variablen, sondern die Adresse ihrer Speicherzelle gemeint ist.

Die Funktion `scanf()` kann bei unsachgemäßem Aufruf jedoch allerlei Probleme machen (siehe Kapitel 24.3.1). Es ist daher empfehlenswert, das Einlesen aufzugliedern:

1. Definition eines Eingabepuffers
   char zeile[100];
2. Einlesen der Tastatureingabe in Puffer
   fgets (zeile, sizeof(zeile), stdin);
3. Formatiertes Einlesen von Puffer in Variablen. (Der Puffer ist üblicherweise ein String, daher das zusätzliche s in `sscanf`.)
   `sscanf (zeile, FORMATSTRING, &var1, &var2,..);`

Das folgende Programm verdeutlicht, wie auf beide Weisen von der Tastatur aus eingelesen werden kann.

```
/* scanf.cpp   Daten einlesen */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main()
  {
  char zeile[100];
  char vorname[40]= "";
  int alter=0;

  printf("Geben Sie Ihren Vornamen und ihr Alter an: \n\n\t");
  fgets(zeile, sizeof(zeile), stdin);
  sscanf(zeile,"%s %d\n", vorname, &alter);
  printf("Sie heissen %s und sind %d Jahre alt\n\n",vorname,alter);

  printf("Geben Sie Ihren Vornamen und ihr Alter an: \n\n\t");
  scanf("%s %d",vorname, &alter);
  printf("Sie heissen %s und sind %d Jahre alt\n\n",vorname,alter);
  fflush(stdin);
  return 0;
  }
```

Ausgabe:

```
Geben Sie Ihren Vornamen und ihr Alter an

   Dirk 33
   Sie heissen Dirk und sind 33 Jahre alt

Geben Sie Ihren Vornamen und ihr Alter an:

   Ken 26
   Sie heissen Ken und sind 26 Jahre alt
```

Analyse Das Argument `stdin` im Aufruf von `fgets()` gibt an, dass von der Standardeingabe gelesen werden soll – also üblicherweise von der Tastatur. Die Funktion `sscanf()` wird im Prinzip genauso aufgerufen wie `scanf()`, nur dass nicht direkt von `stdin`, sondern von einem String, der als erster Parameter übergeben wird, gelesen wird. Zeichenketten wie `vorname[40]` brauchen keinen Adressoperator `&` im Aufruf von `scanf()` oder `sscanf()`, da sie von C/C++ sowieso als Adressen behandelt werden (mehr dazu später im Abschnitt 8.2.4 zu den Arrays).

3.4 Stil

Zum Abschluss noch ein Wort über guten und schlechten Stil.

C/C++ besitzt eine sehr kryptische Syntax, die daraus resultiert, dass es viele bedeutungstragende Syntax-Elemente gibt, die durch einzelne Zeichen dargestellt werden ({, (, ++, %, ., -> etc.), und dass sich die einzelnen Syntax-Elemente zu den merkwürdigsten Konstrukten verbinden lassen (`for(;;)` oder `while(strcmp(getc(),'c'))`).

Zwar kann man den sich daraus ergebenden Quelltexten eine gewisse asketische Eleganz kaum absprechen, doch trägt dies weder zur Lesbarkeit noch zur Wartbarkeit der Programme bei – was umso schwerer wiegt, als ein

Tippfehler in nur einem Zeichen in C schnell zu einem katastrophalen Fehler führen kann.

Tun Sie also zumindest Ihr Übriges, um den Quelltext übersichtlich und gut lesbar zu gestalten.

Verbessern Sie die Lesbarkeit Ihrer Programme durch Einrücken und Kommentare

Wenn Sie beispielsweise Ihre Funktionen wie folgt formulieren:

```
int main(){printf("Hello, World\n"); return 0;}
```

um die Codegröße gegenüber der Version

```
/* Hauptfunktion */
int main()
  {
  printf("Hello, World\n");
  return 0;
  }
```

zu reduzieren, liegen Sie gleich in zweierlei Hinsicht falsch:

1. Man sollte die Codegröße zwar nicht unnötig aufblähen, doch sollte man alles in der richtigen Relation sehen. Geringe Einsparungen fallen bei den heutzutage zur Verfügung stehenden RAM-Kapazitäten kaum ins Gewicht und lohnen die Mühe nicht.
2. Für die Code-Reduzierung ist nur die Größe der resultierenden EXE-Datei maßgeblich und nicht die Größe des Quelltextes. Obige Reduzierung wurde beispielsweise nur dadurch erreicht, dass alle so genannten Whitespace-Zeichen (Leerzeichen, Tabulatoren, Zeilenumbruch) und Kommentare zwischen den bedeutungstragenden Elementen gelöscht wurden. Dies macht der Compiler aber vor der Übersetzung automatisch. Es ist also ganz unnötig, dass Sie dies selbst tun.

Verbessern Sie also die Lesbarkeit Ihrer Programme, indem Sie

- in eine Zeile nur eine Anweisung schreiben

  ```
  printf("Hello, World\n");
      return 0;
  ```

  statt

  ```
  printf("Hello, World\n"); return 0;
  ```

- Blöcke durch Leerzeichen oder Tabulatoren einrücken

  ```
  int main()
        {
        int i, n;
        for (i=1; i < 10; i++)
          {
           n = i * i;
           printf("%d\n",i);
          }
        }
  ```

  oder

  ```
  int main()
  {
    int i, n;
    for (i=1; i < 10; i++)
    {
       n = i * i;
       printf("%d\n",i);
    }
  }
  ```

  statt

```
int main()
    {
    int i, n;
    for (i=1; i < 10; i++)
    {
    n = i * i;
    printf("%d\n",i);
    }
    }
```

- bedeutungstragende Elemente durch Whitespace-Zeichen trennen, wenn dies die Lesbarkeit fördert

```
while( (i<12) &&
          ((n>100) || (n<0))  )
```

statt

```
while((i<12)&&((n>100)||(n<0)))
```

aber nicht etwa

```
i = n  ++  +  m;
```

statt

```
i = n++  +  m;
```

- Ihre Programme kommentieren, wobei Sie weder zu ausgiebig noch zu sparsam kommentieren sollten. Schön wäre es, wenn Sie
  - einzelne Funktionen mit einem vorangehenden Kommentar versehen, der die Aufgabe der Funktion beschreibt und die Parameter und den Rückgabewert erklärt,
  - wichtige Variablen durch sinnvolle Namen und einen kurzen //-Kommentar beschreiben,
  - interessante Code-Blöcke, die eine in sich abgeschlossene Aufgabe erfüllen, ebenfalls kurz kommentieren.


TIPP

Sie werden beim Durchsehen der Beispielprogramme aus diesem Buch wahrscheinlich feststellen, dass ich mich leider selbst nicht immer an obige Regeln halte. Mein schlechtes Beispiel soll für Sie allerdings nicht maßgeblich sein. Versuchen Sie, es mit der Kommentierung nicht zu übertreiben (nicht dass es Ihnen dann irgendwann lästig fällt und Sie überhaupt nicht mehr kommentieren), aber versuchen Sie auch, die Kommentierung als essenziellen Bestandteil der Programmentwicklung zu sehen.

```
/* FUNC_A.CPP Funktionen */
    #include <stdio.h>

       /* Funktion zur Berechnung des Quadrats        */
      /* Parameter n: Wert, für den das              */
     /*              Quadrat berechnet werden soll */
    /* Rueckgabe: Das berechnete Quadrat           */
```

```
int quadrat(int n)
  {
  int ergebnis;

  ergebnis = n * n;
  return (ergebnis);
  }

int main(int argc, char **argv)
  {
  int n_main;                    // Argument fuer Fkt-Aufruf

  n_main = 10;

  // Quadrat berechnen
  n_main = quadrat(n_main);

  // Ergebnis ausgeben
  printf("Quadrat = %d\n",n_main);

  return 0;
  }
```


Wo Whitespace-Zeichen nicht eingefügt werden dürfen:

Oben habe ich angemerkt, dass der Compiler Whitespace-Zeichen zwischen den bedeutungstragenden Elementen löscht, weswegen man sie zur übersichtlicheren Gestaltung des Quelltextes nutzen sollte. Es gibt allerdings einige Ausnahmefälle, in denen man Whitespace-Zeichen nicht so freizügig setzen darf:

- *String-Literale*
  In String-Literalen (»Dies ist ein String-Literal«) werden Whitespace-Zeichen natürlich nicht gelöscht – sie sind ja Teil des Strings.

  In Escape-Sequenzen (\n, \r) dürfen zwischen dem Backslash und dem Sonderzeichen keine Whitespace-Zeichen stehen.

- *Makros und Bezeichner*
  In Bezeichnern dürfen keine Whitespace-Zeichen vorkommen. Dies gilt insbesondere für die Bezeichner von Makros:

  ```
  #define inc(a) ++(a)   // korrekt
  #define inc  (a) ++(a)  // falsch: ersetzt inc durch a ++a
  ```

- *Template-Parameter (nur C++)*
  Wenn bei der Deklaration von Templates Template-Instanzen als Argumente eingesetzt werden, ist darauf zu achten, dass die schließenden eckigen Klammern durch Leerzeichen getrennt sind: `template<class T = class X<int> >`.

Umgekehrt nutzt der Compiler die Whitespace-Zeichen zur lexikalischen Analyse, denn Whitespace-Zeichen dienen zur Trennung der syntaktischen Elemente (Schlüsselwörter, Operatoren, Bezeichner, etc.). Fehlen Whitespace-Zeichen zwischen diesen Elementen, versucht der Compiler die Grenzen zwischen den Elementen selbst zu ermitteln, indem er versucht, von links nach rechts lesend so viele Zeichen wie möglich zu einem Element zusammenzunehmen. Der folgende Ausdruck:

```
j = n+++m;
```

würde also als

```
j = n++ +m;
```

und nicht als

```
j = n+ ++m;
```

interpretiert.

3.5 Allgemeine Hinweise

Zum Abschluss dieser Einführung noch ein paar Tipps für Ein- und Umsteiger.

3.5.1 Goldene Regeln für Anfänger

Niemals

1. `short`-Variablen verwenden.
2. `unsigned-int`-Variablen verwenden.
3. `goto`-Sprünge verwenden.
4. Makros nur für symbolische Konstanten und nicht etwa als eine Art Autokorrektur oder Ersetzen-Befehl für abgekürzte Code-Fragmente nutzen.
5. einen Zeiger dereferenzieren, der keine gültige Adresse beinhaltet.
6. Funktionsaufrufe als Operanden einsetzen.

Immer daran denken:

7. Die Array-Indizierung beginnt bei 0, und der größte erlaubte Index ist 1 kleiner als die Anzahl der Elemente im Array!
8. Jeden deklarierten Zeiger sofort initialisieren (zumindest mit `NULL`).
9. Bevor man beginnt, einen Zeiger zu dereferenzieren, prüfen, ob der Zeiger ungleich `NULL` ist.
10. Jeden Zeiger, dessen Objekt freigegeben wurde (mit `free` oder `delete`), auf `NULL` setzen.

3.5.2 Von A wie Anweisung bis W wie Whitespace

Anweisung — Einzelner Befehl eines Programms. In C/C++ werden Anweisungen mit Semikolon abgeschlossen.

```
int var1;              // Deklaration
var1 = (var2 + 3) * 3;  // Zuweisung
```

Argument — Wert, der einem Funktionsparameter beim Aufruf der Funktion übergeben wird (meist eine Konstante, eine Variable oder der Rückgabewert einer anderen Funktion).

```
func1(var1, 3);
```

Ausdruck — Einheit aus Operatoren und deren Operanden.

```
(i < 9)
(var2 + 3) * 3
```

Block — In geschweifte Klammern ({ }) eingeschlossene Folge von Anweisungen und Deklarationen. Blöcke können ineinander verschachtelt sein.

```
{
var1 = (var2 +3);
if (var1 < 10)
  {
  printf("Hurra!\n");
  }
}
```

Definition — Einführung eines Bezeichners inklusive Speicherreservierung für das Objekt, das der Bezeichner repräsentiert.

Deklaration — Einführung eines Bezeichners ohne Speicherreservierung.

Funktion — Ausgelagerter Anweisungsblock, der eine in sich abgeschlossene Aufgabe erfüllt. Der Anweisungsblock ist an einen Funktionsnamen gebunden, über den er aufgerufen und ausgeführt werden kann. Mittels Parameter und Rückgabewert kann eine Funktion Daten mit anderen Funktionen austauschen.

Parameter — Variablen, die Teil der Schnittstelle einer Funktion darstellen. Beim Aufruf der Funktion werden den Parametern Werte übergeben.

Variable — In ein Programm eingeführter Bezeichner, der mit einer Speicherzelle verbunden ist und je nach Kontext deren Adresse oder Inhalt repräsentiert.

Whitespace — Leerzeichen, Tabulatoren, Zeilenumbruch.

Ein ausführlicheres Glossar finden Sie im Anhang des Buches.

3.5.3 Hinweise für Umsteiger von Pascal/Delphi

Einige Sprachelemente in C führen bei Pascal-Programmierern besonders häufig zu Fehlern oder zu Verwirrung. Die wichtigsten finden Sie in unten stehender Tabelle.

Tabelle 3.7: Entsprechungen C – Pascal

C	Pascal	Bedeutung
`int`	`Integer`	Ganze Zahlen
`0, !0 ( C)` `bool (C++)`	`Boolean`	Boolescher Wahrheitswert
`float`	`Single`	Fließkommazahlen
`double`	`Double`	Fließkommazahlen
`char *, char str[]`	`String`	Strings, Zeichenketten
`=`	`:=`	Zuweisung
`==`	`=`	Vergleich auf Gleichheit
`/`	`/ oder div`	Division
`%`	`mod`	Rest der Division
`&&`	`and`	logisches UND
`||`	`or`	logisches ODER
`!`	`not`	Negation
`switch`	`case`	bedingte Entscheidung
`do-while`	`repeat-until`	Schleife
`float feld[5][3];`	`feld : array[1..5][1..3] of real;`	Array-Deklaration Vorsicht: Indizierung beginnt in C/C++ immer mit 0
`bioskey(1), kbhit()` `(kein ANSI)`	`keypressed`	Tastaturabfrage
`printf`	`write`	Ausgabe
`scanf`	`read`	Eingabe

Vergessen Sie auch nicht, dass C/C++ im Unterschied zu Pascal zwischen Groß- und Kleinschreibung unterscheidet.

3.5.4 Hinweise für Umsteiger von Java oder C++

Im Unterschied zu Java bzw. C++

- müssen alle in einer Quelltextdatei verwendeten Elemente vorab deklariert werden. C++-Bibliotheken erleichtern die Verwendung ihrer Elemente, indem sie Header-Dateien zur Verfügung stellen, in denen die Deklarationen der Bibliothekselemente gesammelt sind. Der Programmierer muss dann nur noch mittels der `#include`-Direktive die Header-Dateien in seinen Quelltext einkopieren. Die Compiler von Java und C# suchen dagegen selbstständig nach den Definitionen der verwendeten Elemente und zieht aus den Definitionen die benötigten Informationen heraus.
- erlaubt C++ die Programmierung mit Zeigern.
- werden String-Literale mit WideCharacters mit einem `L` gekennzeichnet: `L""`.

3.6 Abschlussquiz: Wie ein Zeichen ein Programm verändert

Manchem Außenstehenden kommt C wie das reinste Chinesisch vor. Und obwohl C nicht für »C wie Chinesisch« steht, gibt es Parallelen. So kann man beispielsweise im Chinesischen den Sinn eines ganzen Schriftzeichens durch Auslassung oder Ersetzung eines einzelnen kleinen Striches verändern. In gleicher Weise gibt es eine Reihe von Situationen, in denen man den Sinn einer C-Anweisung durch den Austausch eines einzelnen kleinen Zeichens verändern kann.

Gemeint ist hier natürlich nicht, dass man die Bedeutung einer Addition

```
var = 3 + 4;
```

verändert, indem man einer Zahl ein Negationszeichen voranstellt oder den Plus-Operanden durch den Minus-Operanden ersetzt. Gemeint sind hier bestimmte tückische Ersetzungen und Tippfehler, die wegen ihrer Unauffälligkeit oft nur schwer zu finden sind.

Vielleicht versuchen Sie selbst einmal, die Fehler zu finden:

1.

```
while(i <= MAX)
  {
  if ( ((x <= 100) && (x >= 0)) || ((x = 1) && (y != 0)) )
    printf("x = %d\n",x);
    else
```

```
    continue;
  i++;
  }
```

2.

```
for(int i = 3; i <= MAX; i++);
  {
  x = y*x;
  if ( x < y  )
    {
    ...
```

3.

```
float var_x;
...
printf("12,3 mal var_x = %f\n",var_x*12,3);
```

Und hier kommen die Auflösungen.

Zu 1. Zuweisung statt Vergleich

Hier wird in einem Teilausdruck der `if`-Bedingung der Variablen `x` der Wert 1 zugewiesen:

```
...(x = 1)...
```

Gewollt war natürlich der Vergleich von `x` gegen 1: `(x == 1)`.

Der Compiler führt die Zuweisung durch und interpretiert dann den Inhalt von `x` als Wahrheitswert. Da `x` nun gleich 1 ist, ist der Wahrheitswert gleich `true`. In obiger Bedingung führt dies dazu, dass die gesamte Bedingung immer erfüllt ist.

Zu 2. Semikolon nach Bedingung oder Schleifenkopf

Hier wurde der Kopf der `for`-Schleife mit einem Semikolon abgeschlossen:

```
for(int i = 3; i <= MAX; i++);
```

Zu einem Schleifenkopf gehört aber kein abschließendes Semikolon (ebenso wenig wie zu einer `if`-Bedingung). Der Compiler interpretiert das Semikolon als leere Anweisung und führt diese so oft aus, wie es die `for`-Schleife verlangt. Danach wird der nachfolgende, in Klammern gefasste Anweisungsblock ausgeführt – genau einmal.

Zu 3. Überflüssiges Komma in printf()

Hier wurde die Fließkommazahl `12.3` in der `printf()`-Anweisung mit Komma geschrieben:

```
printf("12,3 mal var_x = %f\n",var_x*12,3);
```

Fließkommazahlen müssen aber mit Punkt statt Komma geschrieben werden. Der Compiler interpretiert dies so, dass `printf()` drei Argumente übergeben werden. Außer dem Formatstring die Ausdrücke:

```
var_x*12
```

und

```
3
```

Den ersten Ausdruck baut er in den Formatstring ein, den zweiten unterschlägt er einfach.