Weitere Schleifen. Unterprogramme, Funktionen.

Jörn Loviscach

Versionsstand: 21. Oktober 2010, 21:03

Die nummerierten Felder sind absichtlich leer, zum Ausfüllen in der Vorlesung.
Videos dazu: http://www.youtube.com/joernloviscach

1 Schleifen: while, do ... while, for

Die `while`-Schleife in C prüft die in Klammern angegebene Bedingung. Ist die erfüllt, wird der Block `{...}` ausgeführt, abermals die Bedingung geprüft usw. Ergibt die Überprüfung ein `false`, geht das Programm mit den Befehlen nach dem `{...}` weiter. Das kann auch sofort passieren, so dass der Block `{...}` *nie* ausgeführt wird. In C beziehungsweise im Struktogramm sieht die `while`-Schleife so aus:

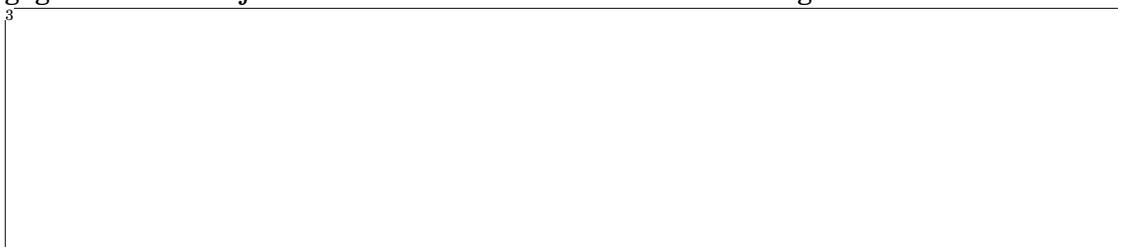
[1]

Die eher seltene `do...while`-Schleife in C führt den Block `{...}` aus und prüft *dann* die hinten in Klammern angegebene Bedingung. Ist die erfüllt, wird abermals der Block `{...}` ausgeführt usw. Ergibt die Überprüfung ein `false`, geht das Programm mit den Befehlen nach dem `;` weiter. Der Block `{...}` wird also immer mindestens einmal ausgeführt. In C und im Struktogramm sieht das so aus:

[2]

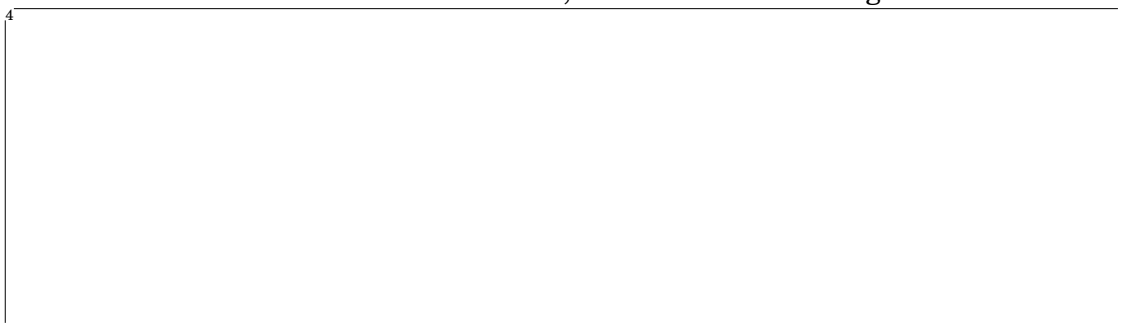
Der dritte und weitaus häufigste Schleifentyp in C ist die `for`-Schleife. Sie dient typischerweise dazu, eine Zählvariable – gerne `i` genannt – hochzuzählen, um eine gegebene Zahl Objekte zu durchlaufen. In C und im Struktogramm sieht das so aus:

[3]



Im `for` sind dabei drei Zutaten definiert, durch Semikolons abgetrennt:

[4]



Variablen, die in der Initialisierung im `for` eingeführt werden, sind außerhalb der `for`-Schleife unsichtbar. (Vorsicht mit antiken Systemen: `for(int i = ...` geht erst seit der Version C99. Früher musste man `i` außerhalb des `for` einführen. Und eine Anmerkung zu C-Nachfahren: JavaScript ist viel laxer mit den Sichtbarkeitsregeln.)

Alle drei Schleifen können mit `break;` jederzeit verlassen werden. In allen drei Schleifen bricht `continue;` den aktuellen Schleifendurchgang ab und springt damit ggf. zum nächsten Schleifendurchgang, je nachdem, ob die Schleifenbedingung noch erfüllt ist. Die Befehle `break;` und continue;[c1] sind aber verkappte `GOTO`s und können zu Spaghetticode führen.

[c1] jl: return;

2 Unterprogramme

Oft benötigt man dieselben Befehlsfolgen mehrfach im selben Programm, zum Beispiel aus der ersten Praktikumsaufgabe: einen Analogeingang einlesen. Ein wichtiges Prinzip der Softwareentwicklung ist, nichts mehrfach zu schreiben. Eine erste Art, das zu umgehen, sind Unterprogramme = Subroutinen = Prozeduren. In C heißen die Funktionen, in vielen objektorientierten Programmiersprachen Methoden. Man fasst eine Befehlsfolge zu einer Funktion zusammen und ruft diese Funktion an allen Stellen auf, an denen die Befehlsfolge nötig ist. Vorteile:

- weniger Tipparbeit,
- weniger Fehlersuche,
- mehr Übersichtlichkeit, insbesondere, wenn die Funktionen sinnvoll benannt sind.

Dies ist *prozedurale* Programmierung im engeren Sinne.

Im besten Fall sieht ein echtes Programm mit Hilfe von Funktionen aus wie Pseudocode:

```
while(isButtonPressed())
{
  int valueInMeters = readAnalogChannel(42);
  int valueInInches = 39*valueInMeters;
  writeOnDisplay(valueInInches);
}
```

Um flexibel zu sein, kann man Werte an Funktionen übergeben und von Funktionen Werte zurückerhalten – genau wie in der Mathematik. Anders als in der Mathematik können die Funktionen = Unterprogramme = Subroutinen = Prozeduren der imperativen Programmiersprachen aber Nebeneffekte [side effects] haben: ein Text wird ausgegeben, ein Servo verstellt, ein Flug gebucht. Diese Nebeneffekte sind meist das Wichtigste beim Programmieren.

„Top-Down Approach“: Ein bewährter Ansatz bei der Softwareentwicklung ist, das Hauptprogramm möglichst verständlich mit bequemen Funktionen hinzuschreiben und dann diese Funktionen auszuformulieren – gegebenenfalls wieder mit anderen Funktionen.

3 Wiederverwendung

Funktionen unterstützen außerdem eine Arbeitsteilung durch Wiederverwendung [recycling]. Das klassische Beispiel dafür sind Funktionssammlungen, die ein Hersteller und/oder ein Open-Source-Projekt zur Verwendung in von anderen Leuten entwickelten Anwendungsprogrammen bereitstellt.

Eine solche Funktionssammlung wird typischerweise nicht als C-Programm ausgeliefert, sondern als nackte Anwendungsprogrammierschnittstelle [API, Application Programming Interface] zu der Funktionssammlung (Details später). Die Anwendungsprogrammierer sehen nur, welche Funktionen enthalten sind, aber nicht, wie die intern arbeiten.

Klassische Beispiele:

- Die Windows API (früher Win32 genannt) ist die offizielle Programmierschnittstelle für Microsoft Windows. Sie umfasst mehr als 2000 Funktionen, vom Erzeugen eines Bildschirmfensters über das Abspielen eines Klangs bis hin zum Aufbau einer Netzverbindung.
- OpenGL ist eine plattformübergreifende Programmierschnittstelle, um Echtzeit-3D-Grafiken zu erzeugen. Die meisten der Funktionen stellen die Grafikkarte ein – oder laden Daten und spezielle Programme auf die Grafikkarte.

4 Funktionen in C

Funktionen in C können Werte entgegennehmen und Werte zurückliefern. Das sieht zum Beispiel so aus:

```
5
```

Sie können aber auch einfach nur Werte entgegennehmen:

```
6
```

Oder weder Werte entgegennehmen noch ausgeben:

```
7
```

Oder nur Werte ausgeben:

```
8
```

Die Hauptfunktion `int main(){...}` jedes C-Programms ist meist von der letzteren Art. Der Rückgabewert wird typischerweise als Fehlernummer verwendet (0 = kein Fehler). Für die heute aussterbenden Kommandozeilenprogramme verwendet man eine etwas andere Hauptfunktion: `int main(int argc, char *argv[]){...}`

Ohne Tricks können Funktionen in C immer nur einen Wert ausgegeben. Sie können aber mehrere Werte entgegennehmen:

[9]

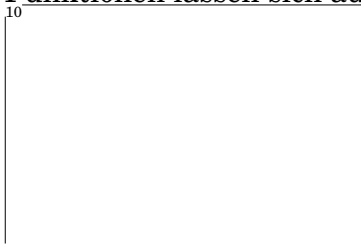
Demo: Debuggen in einen Funktionsaufruf hinein.

Funktionen, die einen Rückgabewert liefern, müssen in allen Fällen in ein `return` münden, in dem eine Größe des passenden Typs zurückgegeben wird. Funktionen, die keinen Rückgabewert liefern, brauchen kein `return`. Man kann aber auch bei diesen Funktionen ein leeres `return;` verwenden, um die Funktion mittendrin zu verlassen. Das ist aber wieder ein verstecktes `GOTO` und deshalb oft unschön.

5 Call by Value, statische Variablen

Funktionen lassen sich auch mit Ausdrücken aufrufen:

[10]



Was auch immer in den runden Klammern steht, wird beim Lauf des Programms ausgewertet; dieser Wert wird in die jeweils angegebene Variable geschrieben („Call by Value“). Was man innerhalb der Funktion mit dieser Variable anstellt, hat keinen Effekt auf den Rest des Programms. Auch alle anderen (Hilfs-)Variablen, die man innerhalb der Funktion einführt, bleiben außen unsichtbar: „lokale Variablen“. Ihre Werte gehen beim Verlassen der Funktion verloren.

In einigen wenigen, aber wichtigen Fällen ist es nötig, dass lokale Variablen ihren Wert bis zum nächsten Aufruf der Funktion behalten. Dazu kann man sie in C als `static` einführen:

[11]

6 Deklaration, Definition, Parameter, Argumente

Anders seine modernen Nachfahren wie Java und C# verlangt C, dass jede Programmdatei von Anfang bis Ende ohne Überraschungen zu lesen ist: Verwendet man vorne eine Funktion, die erst hinten eingeführt wird, ist das verboten (C in der Version C99) oder es passiert Blödsinn (C vor Version C99).

Am übersichtlichsten ist es, die Hauptfunktion `main` ganz vorne in der C-Datei zu haben, erst danach die Unterprogramme = Funktionen. Später werden wir der Ordnung halber sogar mehrere C-Dateien verwenden.

Das Problem dabei: Wenn man mit `main` anfängt, sind die darin verwendeten Funktionen noch unbekannt. Aber C hat einen Ausweg: Man kann die Funktionen zu Beginn knapp „deklarieren“, dann verwenden und erst später „definieren“.

Die Deklaration = der Prototyp[c1] sagt, wie eine Funktion von außen aussieht, wie sie verwendet wird:[12]

[c1] *text added by jl*

Statt der bisherigen Schweifklammern steht ein Semikolon! Die Deklaration gibt die „Signatur“ der Funktion an: Welche Typen von (formalen) Parametern gehen hinein, welchen Typ liefert die Funktion zurück?

Die Definition sagt dagegen, wie eine Funktion von innen aussieht, wie sie arbeitet. Das kennen wir schon:[13]

Die beim Programmlauf tatsächlich übergebenen Werte heißen Argumente der Funktion. Oft unterscheidet man aber nicht so pingelig zwischen den Begriffen „Parameter“ und „Argument“.

In C muss – anders als in neueren Sprachen der C-Familie – eine Funktion myFunction, die keinen Parameter hat und z. B. einen int zurückliefert, sicherheitshalber ausdrücklich als int myFunction(void); statt als int myFunction(); deklariert werden; entsprechend sollte (muss aber nicht) auch int myFunction(void){...} bei der Definition stehen. Beim Aufruf dieser Funktion stehen aber auch in C immer nur die leeren Klammern myFunction().[c2]

[c2] *text added by jl*