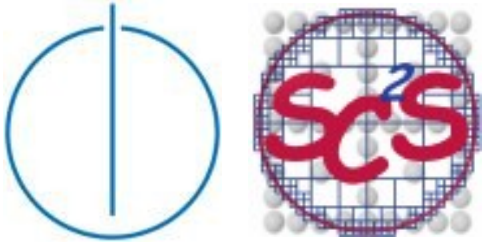
Einführung in die Programmierung II

3. Kontrollstrukturen

Thomas Huckle, Stefan Zimmer

2.5.2007



Anweisungen: was bisher geschah

- Bisher kennen wir zwei Formen von Anweisungen
  - Ein Ausdruck mit Semikolon ist eine Anweisung (Anders als in Maple umfasst das in C auch die Zuweisungen, die hier Ausdrücke sind)
  - Eine Variablendeklaration ist eine Anweisung (Streng genommen gilt das nur in C++; in C ist eine Variablendeklaration keine Anweisung, das hat für uns aber nur die Konsequenz, dass dort Deklarationen Anweisungen vorangehen müssen, während wir in C++ beliebig mischen dürfen)
- Heute kommen – analog zur gleichnamigen Maple-Lektion – die anderen Standardformen von Anweisungen dran: Blöcke (gab's in Maple nicht) Schleifen und Fallunterscheidungen

Blöcke

- Ein Block besteht aus keiner oder mehr Anweisungen, eingeschlossen in geschweiften Klammern `{}`, etwa so:

```
{   int x;
    x = 7;
    int y = x + 3;
    std::cout << y;
}
```

- Ein Block zählt als eine Anweisung
- Wenn er ausgeführt wird, werden die Anweisungen, aus denen er besteht, der Reihe nach ausgeführt (Ausnahmen später)
- Variablen, die innerhalb eines Blocks deklariert werden, gelten nur für diesen Block, nicht außerhalb der `{}` - dazu ist noch einiges zu sagen, aber nicht heute ☺

while-Schleife

- Ganz ähnlich wie in Maple, nur ohne `do`/`end do`, dafür mit runden Klammern um die Bedingung:
  `while` **`(`***Bedingung*`)`** ***Anweisung***
- Die Anweisung wird ausgeführt, solange die Bedingung (ein Ausdruck) `true` bzw. einen Wert ungleich 0 ergibt
- Die Anweisung kann natürlich auch ein Block sein
- Beispiel (gibt „`i: 14 s: 105`“ aus):

```
int i=0, s=0;
while (s<100) {
   i = i+1;
   s = s + i;
}
std::cout << "i: " << i << " s: " << s;
```

do-Schleife

- Die `do`-Schleife stellen wir uns als Variante der `while`-Schleife vor, sie sieht so aus:
  `do` *Anweisung* **`while`** `(`*Bedingung*`)` `;`
- Wieder wird die Anweisung wiederholt, solange die Bedingung `true` (bzw. einen Wert ungleich 0) ergibt
- Der Unterschied ist, dass die Bedingung das erste Mal *nach* dem ersten Schleifendurchlauf ausgewertet wird
- Bei der `do`-Schleife wird die Anweisung also mindestens einmal ausgeführt
- Wenn bei der `while`-Schleife die Bedingung von Anfang an nicht erfüllt ist, wird dort die Anweisung gar nicht ausgeführt
- Um das zu klären, ist die `do`-Schleife nützlich – ansonsten darf man sie gleich wieder vergessen ☺

for-Schleife (1)

- Hatten wir schon als einfache Zählschleife
  `for (i=1; i<=10; i++)` *Anweisung*
  Die `for`-Schleife in C kann aber viel mehr als das...
- Die Syntax ist allgemein
  `for` **`(`***Anfang*`;` *Bedingung*`;` *Weiter*`)` *Anweisung*
  mit beliebigen Ausdrücken *Anfang*, *Bedingung* und *Weiter*, die allesamt optional sind (d.h., entfallen können)
- Mit wenigen Ausnahmen entspricht das folgendem:

```
Anfang;
while (Bedingung) {
  Anweisung
  Weiter;
}
```

for-Schleife (2)

- Es werden also ausgewertet:
  - *Anfang* einmal, ganz zu Beginn
  - *Bedingung* einmal mehr als wir Schleifendurchläufe haben (weil die letzte Auswertung `false` ergibt)
  - *Weiter* so oft, wie die Anweisung wiederholt wird, und zwar nach jeder Ausführung einmal
- Wenn *Anfang* oder *Weiter* entfallen, entfällt auch der entsprechende Teil (einschließlich des Semikolons) in der `while`-Ersatzkonstruktion.
  Wenn *Bedingung* entfällt, wird `true` eingesetzt (die Schleife läuft ohne Hoffnung auf ein Ende immer weiter):

```
for (;;)
  std::cout << "Wie lange noch?\n";
```

for-Schleife (3)

- Wenn wir jede `for`-Schleife durch eine `while`-Schleife ausdrücken können: dürfen wir die `for`-Schleife auch gleich wieder vergessen?
- Nein! Das ist der Standardfall einer Schleife: ich muss etwas am Anfang machen (z.B. einen Zähler auf 0 setzen) , dann wieder zwischen zwei Schleifendurchläufen (z.B. den Zähler erhöhen) und habe ein Bedingung, wie lange das passieren soll
- Und es ist zweckmäßig, diese Information gesammelt an den Anfang der Schleife zu stellen, wo sie der Leser des Programms gleich findet
- Die `for`-Schleife ist also ein Hilfsmittel zur guten Strukturierung unseres Programms, auch wenn man in C eine Menge Unsinn damit anstellen *kann*

Geschachtelte Schleifen

- Der *Schleifenrumpf* (die Anweisung, die wiederholt wird) kann selber natürlich auch wieder eine Schleife (oder eine Fallunterscheidung) sein oder (wenn er z.B. ein Block ist) enthalten
- In diesem Fall entsteht eine geschachtelte Struktur wie diese hier

```
int i, j;
for (i=1; i<=10; i++)
  for (j=1; j<=i; j++)
    std::cout << "i*j = " << i*j << '\n';
```

- Man spricht in diesem Fall von einer *äußeren* Schleife (mit Schleifenzähler `i`) und einer *inneren* (mit Schleifenzähler `j`), bei tiefer geschachtelten Konstruktionen analog von „weiter außen/innen“ etc.

break und continue

- Gelegentlich kommt es vor, dass man eine Schleife vorzeitig abbrechen will – dazu gibt es die Anweisung
  `break;`
  die die innerste umgebende Schleife abbricht:

```
for (i=1; i<=10; i++) {
   for (j=0; ; j++)
     if (j==i) break;
   std::cout << j << " gefunden!\n";
}
```

- Desweiteren gibt es die Anweisung
  `continue;`
  die bewirkt, dass (bei der innersten umgebenden Schleife) sofort der nächste Schleifendurchlauf beginnt
- Beides sparsam verwenden, da es meist (aber eben nicht immer) die Programme unübersichtlicher macht!

Fallunterscheidung: if

- Die einfache Fallunterscheidung (`if`) gibt es in zwei Formen: mit und ohne else-Zweig:
  `if` **`(`***Bedingung***`)`** *Anweisung1* **`else`** *Anweisung2*
  `if` **`(`***Bedingung***`)`** *Anweisung*
- Die Bedingung ist wieder ein Ausdruck, wobei wieder die Regel „alles von Null verschiedene gilt als Wahr“ greift
- Die Anweisungen sind wieder beliebig, dürfen also insbesondere Blöcke `{...}` oder selber Fallunterscheidungen sein:

```
if      (i<0)   std::cout << "Negativ\n";
else if (i==0)  std::cout << "Null\n";
else            std::cout << "Positiv\n";
```

- Ein `elif` wie in Maple brauchen wir hier gar nicht!

Fallunterscheidung: dangling else

- Das Fehlen des end if elminiert den Wunsch nach einem elif – es hat aber einen schweren Nachteil, nämlich die folgende Fallgrube das *dangling* („baumelnden“) *else*
- Wie ist folgender Programmcode zu interpretieren?

```
if (B1) if (B2) A1; else A2;
```

  - `if (B1) { if (B2) A1; else A2; }`
  - `if (B1) { if (B2) A1 } else A2;`

- Dazu müssen wir noch eine neue Regel lernen: ein else gehört zum letzten if, das noch keins hat.
  Im Beispiel ist also die erste Interpretation (blaues else) richtig.
- Das spricht dafür, sicherheitshalber geschweifte Klammern zu verwenden, auch wenn man nur eine einzelne Anweisung damit klammert!

Fallunterscheidung: switch

- Statt einer komplizierten else if-Konstruktion ist manchmal eine switch-Anweisung praktischer, bei dem in Abhängigkeit vom Wert eines (ganzzahligen) Ausdrucks die Abarbeitung an einer bestimmten Stelle weitergeht:

```
switch (Ausdruck) {
  case Konstante1: Anweisungen1
  case Konstante2: Anweisungen2
  ...
  case KonstanteN: AnweisungenN
  default: Anweisungen                    // Optional
}
```

- Vorsicht: ab der Stelle mit der passenden Konstanten werden alle weiteren Anweisungen ausgeführt – außer mit einem `break;` wird die gesamte Konstruktion verlassen. Details brauchen uns hier nicht zu kümmern.

Sprunganweisung: goto

- Es gibt auch noch die Möglichkeit, Anweisungen mit einer Markierung zu versehen und mittels einer **`goto`**-Anweisung dorthin zu springen
- Da die Fälle, wo so etwas ein Programm verbessert, noch viel seltener sind als bei `break` oder `continue`, anderseits die Fälle, in denen die Verständlichkeit völlig ramponiert wird, sehr häufig sind, erkläre ich die Details gar nicht erst!

Anordnung des Programmcodes

- Dem Übersetzer ist es völlig egal, wie der Programmcode auf Zeilen verteilt ist
- Daher nutzen wir diese Freiheit, um das Programm schön lesbar anzuordnen, insbesondere, um geschachtelte Strukturen durch Einrücken zu verdeutlichen, etwa wie in dem Beispiel, das wir schon kennen:

```
for (i=1; i<=10; i++) {
   for (j=0; ; j++)
     if (j==i) break;
   std::cout << j << " gefunden!\n";
}
```