1.8.5 Wiederholungen

Häufig muss die gleiche Teilaufgabe oft wiederholt werden. Denken Sie nur an die Summation von Tabellenspalten in der Buchführung oder an das Suchen einer bestimmten Textstelle in einem Buch. In C++ gibt es zur Wiederholung von Anweisungen drei verschiedene Arten von Schleifen. In einer Schleife wird nach Abarbeitung einer Teilaufgabe (zum Beispiel Addition einer Zahl) wieder an den Anfang zurückgekehrt, um die gleiche Aufgabe noch einmal auszuführen (Addition der nächsten Zahl). Durch bestimmte Bedingungen gesteuert, zum Beispiel Ende der Tabelle, bricht irgendwann die Schleife ab.

Schleifen mit while

Abbildung 1.7 zeigt die Syntax von `while`-Schleifen. *AnweisungOderBlock* ist wie auf Seite 67 definiert. Die Bedeutung einer `while`-Schleife ist: Solange die Bedingung wahr ist, die Auswertung also ein Ergebnis ungleich 0 oder `true` liefert, wird die Anweisung bzw. der Block ausgeführt. Die Bedingung wird auf jeden Fall zuerst geprüft. Wenn die Bedingung von vornherein unwahr ist, wird die Anweisung gar nicht erst ausgeführt (siehe Abbildung 1.8). Die Anweisung oder der Block innerhalb der Schleife heißt *Schleifenkörper*. Schleifen können wie `if`-Anweisungen beliebig geschachtelt werden.

Abbildung 1.7: Syntaxdiagramm einer `while`-Schleife

```
while(Bedingung1)       // geschachtelte Schleifen, ohne und mit geschweiften Klammern
   while(Bedingung2) {
        .....
        while(Bedingung3) {
            .....
        }
   }
```

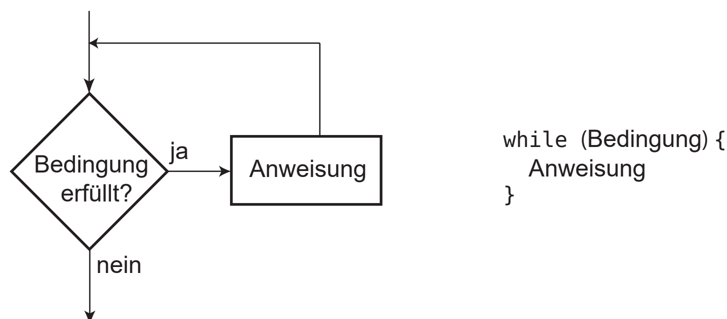

Abbildung 1.8: Flussdiagramm für eine `while`-Anweisung

Beispiele

- Unendliche Schleife:

```
while(true)
   Anweisung
```

- Anweisung wird nie ausgeführt (unerreichbarer Programmcode):

```
while(false)
   Anweisung
```

- Summation der Zahlen 1 bis 99:

```
int sum = 0;
int n = 1;
int grenze = 99;
while(n <= grenze) {
   sum += n++;
}
```

- Berechnung des größten gemeinsamen Teilers ggT(x, y) für zwei natürliche Zahlen x und y nach Euklid. Es gilt:
  - ggT(x, x), also x = y: Das Resultat ist x.
  - ggT(x, y) bleibt unverändert, falls die größere der beiden Zahlen durch die Differenz ersetzt wird, also ggT(x, y) == ggT(x, y-x), falls x < y.

  Das Ersetzen der Differenz geschieht im folgenden Beispiel iterativ, also durch eine Schleife.

Listing 1.14: Beispiel für `while`-Schleife

```
// cppbuch/k1/ggt.cpp    Berechnung des größten gemeinsamen Teilers
#include <iostream>
using namespace std;

int main() {
   unsigned int x;
   unsigned int y;
   cout <<"2 Zahlen > 0 eingeben :";
   cin >> x >> y;
   cout << "Der ggT von " << x << " und " << y << " ist ";
   while( x!= y) {
      if(x > y) {
         x -= y;
      }
      else {
         y -= x;
      }
   }
   cout << x << '\n';
}
```

Innerhalb einer Schleife muss es eine Veränderung derart geben, dass die Bedingung irgendwann einmal unwahr wird, sodass die Schleife abbricht (man sagt auch *terminiert*). Unbeabsichtigte »unendliche« Schleifen sind ein häufiger Programmierfehler. Im ggT-Beispiel ist leicht erkennbar, dass die Schleife irgendwann beendet sein *muss*:

1. Bei jedem Durchlauf wird mindestens eine der beiden Zahlen kleiner.
2. Die Zahl 0 kann nicht erreicht werden, da immer eine kleinere von einer größeren Zahl subtrahiert wird. Die `while`-Bedingung schließt die Subtraktion gleich großer Zahlen aus, und nur die könnte 0 ergeben.

Daraus allein ergibt sich, dass die Schleife beendet wird, und zwar in weniger als `x` Schritten, wenn `x` die anfangs größere Zahl war. Im Allgemeinen sind es erheblich weniger, wie eine genauere Analyse ergibt.

Tipp
Die Anweisungen zur Veränderung der Bedingung sollen möglichst an das Ende des Schleifenkörpers gestellt werden, um sie leicht finden zu können.

Schleifen mit do while

Abbildung 1.9 zeigt die Syntax einer `do while`-Schleife. *AnweisungOderBlock* ist wie auf Seite 67 definiert. Die Anweisung oder der Block einer `do while`-Schleife wird ausgeführt, und *erst anschließend* wird die Bedingung geprüft. Ist sie wahr, wird die Anweisung ein weiteres Mal ausgeführt usw. Die Anweisung wird also mindestens einmal ausgeführt.

Im Flussdiagramm ist die Anweisung ein Block (siehe rechts in der Abbildung 1.10). `do while`-Schleifen eignen sich unter anderem gut zur sicheren Abfrage von Daten, indem die Abfrage so lange wiederholt wird, bis die abgefragten Daten in einem plausiblen Bereich liegen, wie im Primzahlprogramm unten zu sehen ist.

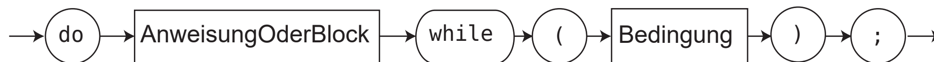

Abbildung 1.9: Syntaxdiagramm einer `do while`-Schleife

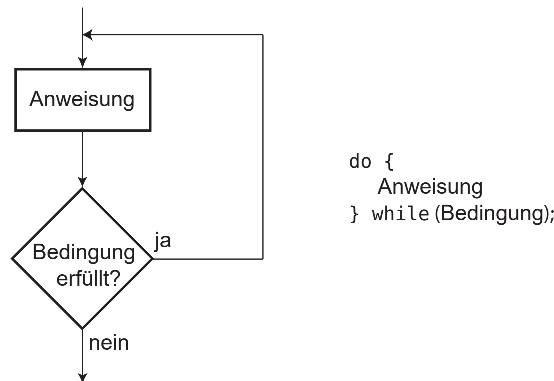

Abbildung 1.10: Flussdiagramm für eine `do while`-Anweisung

Es empfiehlt sich zur besseren Lesbarkeit, `do while`-Schleifen strukturiert zu schreiben. Die schließende geschweifte Klammer soll genau unter dem ersten Zeichen der Zeile stehen, die die öffnende geschweifte Klammer enthält. Dadurch und durch Einrücken des dazwischen stehenden Textes ist sofort der Schleifenkörper erkennbar.

```
do {
    Anweisungen
} while(Bedingung);
```

Das *direkt hinter* die abschließende geschweifte Klammer geschriebene `while` macht unmittelbar deutlich, dass dieses `while` zu einem `do` gehört. Das ist besonders wichtig, wenn der Schleifenkörper in einer Programmliste über die Seitengrenze ragt. Eine `do while`-Schleife kann stets in eine `while`-Schleife umgeformt werden (und umgekehrt).

Listing 1.15: Beispiel für `do while`-Schleifen

```
// cppbuch/k1/primzahl.cpp: Berechnen einer Primzahl, die auf eine gegebene Zahl folgt
#include <iostream>
#include <cmath>
using namespace std;

int main() {
  // Mehrere, durch " getrennte Texte ergeben eine lange Zeile in der Ausgabe.
  cout << "Berechnung der ersten Primzahl, die >="
          " der eingegebenen Zahl ist\n";
  long z;
  // do while-Schleife zur Eingabe und Plausibilitätskontrolle
  do {                                    // Abfrage, solange z ≤ 3 ist
      cout << "Zahl > 3 eingeben :";
      cin >> z;
  } while(z <= 3);
  if(z % 2 == 0) {                  // Falls z gerade ist: nächste ungerade Zahl nehmen
     ++z;
  }
  bool gefunden {false};
  do {
      // limit = Grenze, bis zu der gerechnet werden muss.
      // sqrt() arbeitet mit double, daher wird der Typ explizit umgewandelt.
      long limit {1 + static_cast<long>( sqrt(static_cast<double>(z)))};
      long rest;
      long teiler {1};
      do {                          // Kandidat z durch alle ungeraden Teiler dividieren
          teiler += 2;
          rest    = z % teiler;
      } while(rest > 0 && teiler < limit);
      if(rest > 0 && teiler >= limit) {
         gefunden = true;
      }
      else {                            // sonst nächste ungerade Zahl untersuchen:
         z += 2;
      }
  } while(!gefunden);
  cout << "Die nächste Primzahl ist " << z << '\n';
}
```

Schleifen mit for

Die letzte Art von Schleifen ist die `for`-Schleife. Sie wird häufig eingesetzt, wenn die Anzahl der Wiederholungen vorher feststeht, aber das muss durchaus nicht so sein. Abbildung 1.11 zeigt die Syntax einer `for`-Schleife.

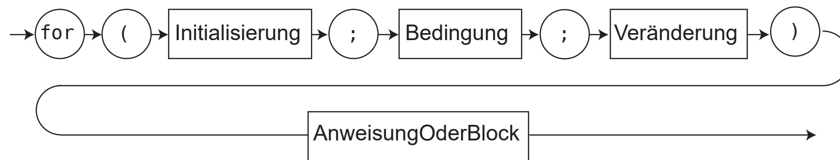

Abbildung 1.11: Syntaxdiagramm einer `for`-Schleife

Der zu wiederholende Teil (Anweisung oder Block) wird auch Schleifenkörper genannt. Beispiel: ASCII-Tabelle im Bereich 65 ... 69 ausgeben

```
for(int i = 65; i <= 69; ++i) {
    cout << i << "␣" << static_cast<char>(i) << '\n';
}
```

Bei der Abarbeitung werden die folgenden Schritte durchlaufen:

1. Durchführung der Initialisierung, zum Beispiel Startwert für eine Laufvariable festlegen. Eine Laufvariable wird wie `i` in der Beispielschleife als Zähler benutzt.
2. Prüfen der Bedingung.
3. Falls die Bedingung wahr ist, zuerst die Anweisung und dann die Veränderung ausführen.

Die Laufvariable `i` kann auch außerhalb der runden Klammern deklariert werden, dies gilt aber als schlechter Stil. Der Unterschied besteht darin, dass außerhalb der Klammern deklarierte Laufvariablen noch über die Schleife hinaus gültig sind.

```
int i;                                          // nicht empfohlen
for(i = 0; i < 100; ++i) {
    // Programmcode, i ist hier bekannt
}
// i ist weiterhin bekannt ...
```

Im Fall der Deklaration innerhalb der runden Klammern bleibt die Gültigkeit auf den Schleifenkörper beschränkt:

```
for(int i = 0; i < 100; ++i) {                  // empfohlen
    // Programmcode, i ist hier bekannt
}
// i ist hier nicht mehr bekannt
```

Die zweite Art erlaubt es, `for`-Schleifen als selbstständige Programmteile hinzuzufügen oder zu entfernen, ohne Deklarationen in anderen Schleifen ändern zu müssen. Derselbe Mechanismus gilt für Deklarationen in den runden Klammern von `if`-, `while`- und `switch`-Anweisungen.

Listing 1.16: Beispiel für `for`-Schleife

```
// cppbuch/k1/fakultaet.cpp
#include <iostream>
using namespace std;

int main() {
   cout << "Fakultät␣berechnen.␣Zahl␣>=␣0?␣:";
   unsigned int n;
```

```
   cin >> n;
   unsigned long fak {1L};
   for(unsigned int i = 2; i <= n; ++i) {
      fak *= i;
   }
   cout << n << "!␣␣␣=␣␣␣" << fak << '\n';
}
```

Verändern Sie niemals die Laufvariable innerhalb des Schleifenkörpers! Das Auffinden von Fehlern würde durch die Änderung erschwert.

```
for(int i = 65; i < 70; ++i) {
    // eine Seite Programmcode
    --i; // irgendwo dazwischen erzeugt eine unendliche Schleife
    // noch mehr Programmcode
}
```

Auch wenn der Schleifenkörper nur aus einer Anweisung besteht, wird empfohlen, ihn in geschweiften Klammern `{ }` einzuschließen.

Äquivalenz von for und while

Eine `for`-Schleife entspricht direkt einer `while`-Schleife, sie ist im Grunde nur eine Umformulierung, solange nicht `continue` vorkommt (das im folgenden Abschnitt beschrieben wird):

```
for(Initialisierung; Bedingung; Veraenderung)
  Anweisung
```

ist äquivalent zu:

```
{
   Initialisierung;
   while(Bedingung) {
      Anweisung
      Veraenderung;
   }
}
```

Die äußeren Klammern sorgen dafür, dass in der Initialisierung deklarierte Variablen wie bei der `for`-Schleife nach dem Ende nicht mehr gültig sind. `Anweisung` kann wie immer auch eine Verbundanweisung (Block) sein, in der mehrere Anweisungen stehen können, durch geschweifte Klammern begrenzt. Die umformulierte Entsprechung des obigen Beispiels (ASCII-Tabelle von 65 ... 69 ausgeben) lautet:

```
{
  int i {65};                                                    // Initialisierung
  while(i < 70) {                                                // Bedingung
      cout << i << "␣" << static_cast<char>(i) << '\n';          // Anweisung
      ++i;                                                       // Veränderung
  }
}
```