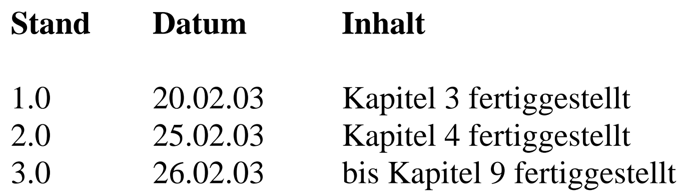
VORWORT

Dieses Dokument enthält meine Musterlösungen zu den Übungen im Buch

Martin Aupperle: Die Kunst der Programmierung mit C++

Vieweg-Verlag 2002, ISBN 3-528-15481-0



Stand	Datum	Inhalt
1.0	20.02.03	Kapitel 3 fertiggestellt
2.0	25.02.03	Kapitel 4 fertiggestellt
3.0	26.02.03	bis Kapitel 9 fertiggestellt

KAPITEL 3

Übung 3-1:

Untersuchen Sie, wie verschiedene Angaben für die Genauigkeit die Rundung der Ausgabe beeinflussen.

Um die unterschiedlichen Genauigkeiten zu testen, verwenden wir eine Schleife von 1 bis 50:

```
float f1   = 2.0;  // Definition einer float-Variablen
double f2  = 2.0;  // Definition einer double-Variablen

f1 = f1 / 3.0;
f2 = f2 / 3.0;



for ( int i = 1; i < 50; i++ )
{
  cout.precision( i );
  cout << "i: " << i << endl << "f1: " << f1  << endl << "f2: " << f2 << endl << endl;
}
```

Mit MSVC6 erhält man folgende Ausgabe:

```
i: 1
f1: 0.7
f2: 0.7

i: 2
f1: 0.67
f2: 0.67

i: 3
f1: 0.667
f2: 0.667

i: 4
f1: 0.6667
f2: 0.6667

i: 5
f1: 0.66667
f2: 0.66667

i: 6
f1: 0.666667
f2: 0.666667

i: 7
f1: 0.6666667
f2: 0.6666667

i: 8
f1: 0.66666669
f2: 0.66666667

i: 9
f1: 0.666666687
f2: 0.666666667

i: 10
f1: 0.6666666865
f2: 0.6666666667

i: 11
f1: 0.66666668653
f2: 0.6666666667

i: 12
f1: 0.666666686535
f2: 0.666666666667

i: 13
f1: 0.6666666865349
f2: 0.6666666666667

i: 14
f1: 0.66666668653488
f2: 0.66666666666667

i: 15
f1: 0.666666686534882
f2: 0.666666666666667

i: 16
f1: 0.6666666865348816
f2: 0.6666666666666666

i: 17
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 18
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 19
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 20
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 21
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 22
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 23
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 24
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 25
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 26
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 27
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 28
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 29
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 30
f1: 0.66666668653488159
f2: 0.66666666666666663
```

```
i: 31
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 32
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 33
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 34
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 35
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 36
f1: 0.66666668653488159
f2: 0.66666666666666663

i: 37
f1: 0.666666686534881590
f2: 0.666666666666666630

i: 38
f1: 0.6666666865348815900
f2: 0.6666666666666666300

i: 39
f1: 0.66666668653488159000
f2: 0.66666666666666663000

i: 40
f1: 0.666666686534881590000
f2: 0.666666666666666630000

i: 41
f1: 0.6666666865348815900000
f2: 0.6666666666666666300000

i: 42
f1: 0.66666668653488159000000
f2: 0.66666666666666663000000

i: 43
f1: 0.666666686534881590000000
f2: 0.666666666666666630000000

i: 44
f1: 0.6666666865348815900000000
f2: 0.6666666666666666300000000

i: 45
f1: 0.66666668653488159000000000
f2: 0.66666666666666663000000000

i: 46
f1: 0.666666686534881590000000000
f2: 0.666666666666666630000000000

i: 47
f1: 0.6666666865348815900000000000
f2: 0.6666666666666666300000000000

i: 48
f1: 0.66666668653488159000000000000
f2: 0.66666666666666663000000000000

i: 49
f1: 0.666666686534881590000000000000
f2: 0.666666666666666630000000000000
```

An dieser Ausgabe lässt sich folgendes ablesen:

- Ein Wert von 0 für `precision` bewirkt die Ausgabe in der Standard-Genauigkeit von 6 Nachkommastellen. Die Rechnung für `float` und `double` bringt das gleiche Ergebnis
- Für Werte zwischen 1 und 17 erhält man die entsprechende Anzahl an Nachkommastellen.
- Ab einer Genauigkeit von 8 Nachkommastellen unterscheiden sich die Ergebnisse für `float` und `double`
- Für Werte zwischen 17 und 36 für `precision` werden unverändert 17 Nachkommastellen ausgegeben.
- Für Werte ab 37 wird die Ausgabe mit Nullen ergänzt.

Übung 3-2:

Untersuchen Sie, ob für ihr System die Verwendung von `long double` an Stelle von `double` eine Erhöhung der Genauigkeit mit sich bringt.

Dies lässt sich z.B. einfach durch die Schleife aus Übung 3-1 feststellen, wenn anstelle der `float/double` Variablen nun `double/long double` Variablen verwendet werden:

```
double        f1  = 2.0;
long double   f2  = 2.0;

f1 = f1 / 3.0;
f2 = f2 / 3.0;


for ( int i = 0; i < 50; i++ )
{
  cout.precision( i );
  cout << "i: " << i << endl << "f1: " << f1  << endl << "f2: " << f2 << endl << endl;
}
```

Mit MSVC6 erhält man folgende Ausgabe:

```
i: 1
f1: 0.7
f2: 0.7

i: 2
f1: 0.67
f2: 0.67

i: 3
f1: 0.667
f2: 0.667

i: 4
f1: 0.6667
f2: 0.6667

i: 5
f1: 0.66667
f2: 0.66667

i: 6
f1: 0.666667
f2: 0.666667

i: 7
f1: 0.6666667
f2: 0.6666667

i: 8
f1: 0.66666667
f2: 0.66666667

i: 9
f1: 0.666666667
f2: 0.666666667

i: 10
f1: 0.6666666667
f2: 0.6666666667

i: 11
f1: 0.66666666667
f2: 0.66666666667

i: 12
f1: 0.666666666667
f2: 0.666666666667

i: 13
f1: 0.6666666666667
f2: 0.6666666666667

i: 14
f1: 0.66666666666667
f2: 0.66666666666667

i: 15
f1: 0.666666666666667
f2: 0.666666666666667

i: 16
f1: 0.6666666666666666
f2: 0.6666666666666666

i: 17
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 18
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 19
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 20
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 21
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 22
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 23
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 24
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 25
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 26
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 27
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 28
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 29
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 30
f1: 0.66666666666666663
f2: 0.66666666666666663
```

```
i: 31
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 32
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 33
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 34
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 35
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 36
f1: 0.66666666666666663
f2: 0.66666666666666663

i: 37
f1: 0.666666666666666630
f2: 0.666666666666666630

i: 38
f1: 0.6666666666666666300
f2: 0.6666666666666666300

i: 39
f1: 0.66666666666666663000
f2: 0.66666666666666663000

i: 40
f1: 0.666666666666666630000
f2: 0.666666666666666630000

i: 41
f1: 0.6666666666666666300000
f2: 0.6666666666666666300000

i: 42
f1: 0.66666666666666663000000
f2: 0.66666666666666663000000

i: 43
f1: 0.666666666666666630000000
f2: 0.666666666666666630000000

i: 44
f1: 0.6666666666666666300000000
f2: 0.6666666666666666300000000

i: 45
f1: 0.66666666666666663000000000
f2: 0.66666666666666663000000000

i: 46
f1: 0.666666666666666630000000000
f2: 0.666666666666666630000000000

i: 47
f1: 0.6666666666666666300000000000
f2: 0.6666666666666666300000000000

i: 48
f1: 0.66666666666666663000000000000
f2: 0.66666666666666663000000000000

i: 49
f1: 0.666666666666666630000000000000
f2: 0.666666666666666630000000000000
```

Wie man sieht, gibt es für MSVC6 keinen Unterschied in der Genauigkeit für `double` und `long double`.

Übung 3-3:

Untersuchen Sie, ab welcher Genauigkeitsangabe für `cout` die Abweichung zu Tage tritt.

Auch hier verwenden wir am Einfachsten wieder eine Schleife, die die Ausgabe mit verschiedenen Genauigkeiten durchführt.

```
double d = 1.0;

d = d + 0.1;
d = d + 0.1;
d = d + 0.1;
d = d + 0.1;
d = d + 0.1;
d = d + 0.1;
d = d + 0.1;
d = d + 0.1;
d = d + 0.1;
d = d + 0.1;

for ( int i = 0; i < 50; i++ )
{
  cout.precision(i);
  cout << "i:" << setw(3) << i << "  d: " << d << endl;
}
```

Mit MSVC6 erhält man folgende Ausgabe:

```
i:  0  d: 2
i:  1  d: 2
i:  2  d: 2
i:  3  d: 2
i:  4  d: 2
i:  5  d: 2
i:  6  d: 2
i:  7  d: 2
i:  8  d: 2
i:  9  d: 2
i: 10  d: 2
i: 11  d: 2
i: 12  d: 2
i: 13  d: 2
i: 14  d: 2
i: 15  d: 2
i: 16  d: 2.000000000000001
i: 17  d: 2.0000000000000009
i: 18  d: 2.0000000000000009
i: 19  d: 2.0000000000000009
i: 20  d: 2.0000000000000009
i: 21  d: 2.0000000000000009
i: 22  d: 2.0000000000000009
i: 23  d: 2.0000000000000009
i: 24  d: 2.0000000000000009
i: 25  d: 2.0000000000000009
i: 26  d: 2.0000000000000009
i: 27  d: 2.0000000000000009
i: 28  d: 2.0000000000000009
i: 29  d: 2.0000000000000009
i: 30  d: 2.0000000000000009
i: 31  d: 2.0000000000000009
i: 32  d: 2.0000000000000009
i: 33  d: 2.0000000000000009
i: 34  d: 2.0000000000000009
i: 35  d: 2.0000000000000009
i: 36  d: 2.0000000000000009
i: 37  d: 2.00000000000000090
i: 38  d: 2.000000000000000900
i: 39  d: 2.0000000000000009000
i: 40  d: 2.00000000000000090000
i: 41  d: 2.000000000000000900000
i: 42  d: 2.0000000000000009000000
i: 43  d: 2.00000000000000090000000
i: 44  d: 2.000000000000000900000000
i: 45  d: 2.0000000000000009000000000
i: 46  d: 2.00000000000000090000000000
i: 47  d: 2.000000000000000900000000000
i: 48  d: 2.0000000000000009000000000000
i: 49  d: 2.00000000000000090000000000000
```

Wie man sieht, wirkt sich die Angabe eines Wertes für die Genauigkeit erst dann aus, wenn die Rundung auf die geforderte Zahl von Stellen ein von der Ganzzahl verschiedenes Ergebnis ergibt.

Interessant sind insbesondere die Ausgaben für Genauigkeiten von 16 und 17 Stellen. Bei 16 Stellen wird ja die 17te Stelle zur Rundung verwendet, bei geforderten 17 Stellen Genauigkeit ist die Grenze der Arithmetik erreicht.

Vollständiges Programm

```
#include <iostream>
#include <iomanip>

using namespace std;


//---------------------------------------------------------------------
//       main
//
int main()
{

  double d = 1.0;

  d = d + 0.1;
  d = d + 0.1;
  d = d + 0.1;
  d = d + 0.1;
  d = d + 0.1;
  d = d + 0.1;
  d = d + 0.1;
  d = d + 0.1;
  d = d + 0.1;
  d = d + 0.1;


  for ( int i = 0; i < 50; i++ )
  {
    cout.precision(i);
    cout << "i:" << setw(3) << i << "  d: " << d << endl;
  }


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 3-4:

Wie notiert man dagegen ein Integer-Literal mit dem Wert `2`? Ein `Long-Integer` Literal mit dem Wert `2`?

Ein Integer-Literal wird ohne Modifizierer einfach als Zahl notiert:

```
int i = 3;  // 3 ist ein Integer Literal
```

Ein Long-Integer Literal erkennt man an einem nachgestellten L:

```
long l = 3L; // 3L ist ein Long-Integer Literal
```

Übung 3-5:

Was ist der Unterschied zwischen `0.5f, .5f` und `0.5L`? Welcher Unterschied besteht zwischen `1.0l` und `1l`?

- `0.5f`, `.5f` sind beides Fließkomma-Literale vom Typ `float` mit dem Wert 0,5 (die Null vor dem Dezimalpunkt darf weggelassen werden).
- `0.5L` ist ein Fließkomma-Literal vom Typ `long double`, ebenfalls mit dem Wert 0,5.
- `1.0l` ist ein Fließkomma-Literal vom Typ `long double` mit dem Wert 1.
- `1l` ist ein Integer-Literal vom Typ `long`, ebenfalls mit dem Wert 1

Die letzten beiden Literale sind schlecht zu lesen, es besteht Verwechslungsgefahr mit `1.01` bzw. `11`. Es wird daher empfohlen, für die Modifizierer immer Großbuchstaben zu verwenden.

Übung 3-6:

Untersuchen Sie, was nach dem Buchstaben `z` folgt. Gibt es eine obere Grenze für sinnvolle Werte?

Die folgende Schleife gibt die Zeichenentsprechungen für die Werte `'a'` bis `'a'+255` aus.

```
char c = 'a';

for ( int i = 0; i < 256; i++ )
{
  char d = c + i;
  cout << "i:" << setw(3) << i << "  " << d << endl;
}
```

Als Ergebnis erhält man

```
i:  0  a
i:  1  b
i:  2  c
i:  3  d
i:  4  e
i:  5  f
i:  6  g
i:  7  h
i:  8  i
i:  9  j
i: 10  k
i: 11  l
i: 12  m
i: 13  n
i: 14  o
i: 15  p
i: 16  q
i: 17  r
i: 18  s
i: 19  t
i: 20  u
i: 21  v
i: 22  w
i: 23  x
i: 24  y
i: 25  z
i: 26  {
i: 27  |
i: 28  }
i: 29  ~
i: 30  ⌂
i: 31  Ç
i: 32  ü
i: 33  é
i: 34  â
i: 35  ä
i: 36  à
i: 37  å
i: 38  ç
i: 39  ê
i: 40  ë
i: 41  è
i: 42  ï
i: 43  î
i: 44  ì
i: 45  Ä
i: 46  Å
i: 47  É
i: 48  æ
i: 49  Æ
i: 50  ô
i: 51  ö
i: 52  ò
i: 53  û
i: 54  ù
i: 55  ÿ
i: 56  Ö
i: 57  Ü
i: 58  ø
i: 59  £
i: 60  Ø
i: 61  ×
i: 62  ƒ
i: 63  á
i: 64  í
i: 65  ó
i: 66  ú
i: 67  ñ
i: 68  Ñ
i: 69  ª
i: 70  º
i: 71  ¿
i: 72  ®
i: 73  ¬
i: 74  ½
i: 75  ¼
i: 76  ¡
i: 77  «
i: 78  »
i: 79  ░
i: 80  ▒
i: 81  ▓
i: 82  │
i: 83  ┤
i: 84  Á
i: 85  Â
i: 86  À
i: 87  ©
i: 88  ╣
i: 89  ║
i: 90  ╗
i: 91  ╝
i: 92  ¢
i: 93  ¥
i: 94  ┐
i: 95  └
i: 96  ┴
i: 97  ┬
i: 98  ├
i: 99  ─
i:100  ┼
i:101  ã
i:102  Ã
i:103  ╚
i:104  ╔
i:105  ╩
i:106  ╦
i:107  ╠
i:108  ═
i:109  ╬
i:110  ¤
i:111  ð
i:112  Ð
i:113  Ê
i:114  Ë
i:115  È
i:116  ı
i:117  Í
i:118  Î
i:119  Ï
i:120  ┘
i:121  ┌
i:122  █
i:123  ▄
i:124  ¦
i:125  Ì
i:126  ▀
i:127  Ó
i:128  ß
i:129  Ô
i:130  Ò
i:131  õ
i:132  Õ
i:133  µ
i:134  þ
i:135  Þ
i:136  Ú
i:137  Û
i:138  Ù
i:139  ý
i:140  Ý
i:141  ¯
i:142  ´
i:143  -
i:144  ±
i:145  ‗
i:146  ¾
i:147  ¶
i:148  §
i:149  ÷
i:150  ¸
i:151  °
i:152  ¨
i:153  ·
i:154  ¹
i:155  ³
i:156  ²
i:157  ■
i:158
i:159
i:160  ☺
i:161  ☻
i:162  ♥
i:163  ♦
i:164  ♣
i:165  ♠
i:166
i:167
i:168
i:169

i:170  ♂
i:171  ♀
i:172
i:173  ♫
i:174  ☼
i:175  ►
i:176  ◄
i:177  ↕
i:178  ‼
i:179  ¶
i:180  §
i:181  ▬
i:182  ↨
i:183  ↑
i:184  ↓
i:185  →
i:186  ←
i:187  ∟
i:188  ↔
i:189  ▲
i:190  ▼
i:191
i:192  !
i:193  "
i:194  #
i:195  $
i:196  %
i:197  &
i:198  '
i:199  (
i:200  )
i:201  *
i:202  +
i:203  ,
i:204  -
i:205  .
i:206  /
i:207  0
i:208  1
i:209  2
i:210  3
i:211  4
i:212  5
i:213  6
i:214  7
i:215  8
i:216  9
i:217  :
i:218  ;
i:219  <
i:220  =
i:221  >
i:222  ?
i:223  @
i:224  A
i:225  B
i:226  C
i:227  D
i:228  E
i:229  F
i:230  G
i:231  H
i:232  I
i:233  J
i:234  K
i:235  L
i:236  M
i:237  N
i:238  O
i:239  P
i:240  Q
i:241  R
i:242  S
i:243  T
i:244  U
i:245  V
i:246  W
i:247  X
i:248  Y
i:249  Z
i:250  [
i:251  \
i:252  ]
i:253  ^
i:254  _
i:255  `
```

Die Ausgabe der auf 'a' folgenden Zeichen zeigt effektiv nacheinander alle Zeichen in dem zur Ausgabe gewählten Schriftfont an. Die Ausgabe wurde in einer DOS-Box auf einem Windows-System erzeugt, Windows hat hier den System-Font voreingestellt.

Wer sich mit der Windows-Programmierung auskennt, kann die Ausgabe auch einmal direkt über das Windows-GDI vornehmen. Je nach Font wird man hier unterschiedliche Ergebnisse erhalten.

Führt man die Ausgabe weiter fort, wiederholt sich die Reihe beginnend bei a immer wieder. Dies zeigt, dass nur die unteren 8 Bit des Integers verwendet werden. Insofern gibt es keine obere Grenze – das Muster wiederholt sich einfach für größere Werte.

Vollständiges Programm

```
#include <iostream>
#include <iomanip>

using namespace std;

//----------------------------------------------------------------------
//       main
//
int main()
{

  char c = 'a';

  for ( int i = 0; i < 256; i++ )
  {
    char d = c + i;
    cout << "i:" << setw(3) << i << "  " << d << endl;
  }


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 3-7:

Untersuchen Sie, welche numerischen Werte die Umlaute ä,ö,ü,Ä,Ö,Ü auf Ihrem System besitzen. Schalten Sie dann Ihr System (falls möglich) auf angloamerikanisch und stellen Sie fest, welchen Zeichen diese Werte dann entsprechen.

Wir erhalten die numerischen Werte durch eine Konvertierung nach `int`. Folgendes Programmsegment führt die Umwandlung für die vier Buchstaben durch und gibt die Ergebnisse aus:

```
int i1 = 'ä';
int i2 = 'ö';
int i3 = 'ü';

int i4 = 'Ä';
int i5 = 'Ö';
int i6 = 'Ü';


cout << 'ä' << "  " << i1 << endl;
cout << 'ö' << "  " << i2 << endl;
cout << 'ü' << "  " << i3 << endl;

cout << 'Ä' << "  " << i4 << endl;
cout << 'Ö' << "  " << i5 << endl;
cout << 'Ü' << "  " << i6 << endl;
```

Mit MSVC erhält man die Ausgabe

```
õ  -28
÷  -10
³  -4
─  -60
Í  -42
■  -36
```

Hier fällt zunächst auf, dass z.B. die Ausgabe des Literals `'ä'` keineswegs den Buchstaben `ä` auf dem Bildschirm produziert. Dies liegt in diesem Fall daran, dass der zum Editieren des Quellcodes verwendete Editor unter Windows lief, die Ausgabe aber unter DOS (genauer: in einer DOS-Box unter Windows) stattfand. Offensichtlich werden in den beiden Systemen Fonts mit unterschiedlicher Belegung der sog. *Codepoints*[1] verwendet.

Das Problem tritt nicht auf, wenn man sich innerhalb eines Systems (DOS oder Windows) bewegt. Es ist letztendlich durch die Tatsache verursacht, dass ein Zeichen mit 8 Bit auskommen muss, was die Anzahl der möglichen Zeichen auf 256 beschränkt. Da dies nicht für alle Anwendungsfälle ausreicht, gibt es (auch aus historischen Gründen) Fonts mit unterschiedlichen Glyphen auf den gleichen Codepoints. Windows und DOS verwenden solche unterschiedlichen Fonts.

Das Problem tritt auch nicht auf, wenn man für ein Zeichen mehr als 8 Bit „spendiert“. Neuere Systeme wie z.B. Java oder .NET benötigen für ein Zeichen 16 Bit. Damit können (derzeit) alle notwendigen Glyphen aller Sprachen auf eindeutige Codepoints abgebildet werden. Das bekannteste System für 16 Bit breite Zeichen ist Unicode, was auch für Java und .NET verwendet wird.

Als nächstes fällt an obiger Ausgabe auf, dass die den Umlauten entsprechenden Integer *negativ* sind. Dies liegt an der vom Compilerbauer auf meiner Maschine gewählten Implementierung für `char`: hier ist nämlich `char` vorzeichenbehaftet implementiert. Der Typ `char` entspricht deshalb `signed char`.

[1] Ein Codepoint (Code-Punkt) ist eine Stelle in einem Font, die zur eine gegebene Zahl identifiziert wird. Das Betriebssystem findet dort die graphische Information, wie das Zeichen auf dem Bildschirm auszusehen hat.

Vollständiges Programm:

```
#include <iostream>

using namespace std;

//--------------------------------------------------------------------
//       main
//
int main()
{

  int i1 = 'ä';
  int i2 = 'ö';
  int i3 = 'ü';

  int i4 = 'Ä';
  int i5 = 'Ö';
  int i6 = 'Ü';


  cout << 'ä' << "  " << i1 << endl;
  cout << 'ö' << "  " << i2 << endl;
  cout << 'ü' << "  " << i3 << endl;

  cout << 'Ä' << "  " << i4 << endl;
  cout << 'Ö' << "  " << i5 << endl;
  cout << 'Ü' << "  " << i6 << endl;


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 3-8:

Schreiben Sie Ausgabeanweisungen, um die Wirkung der Escape-Sequenzen `\t`, `\v`,`\b` *und* `\f` *zu studieren.*

Folgende Anweisungen geben Zeichenketten aus, in die die fraglichen Escape-Sequenzen eingebettet sind:

```
cout << "111" << '\t' << "222" << endl;
cout << "111" << '\v' << "222" << endl;
cout << "111" << '\b' << "222" << endl;
cout << "111" << '\f' << "222" << endl;
```

In einem DOS-Fenster unter Windows erhält man folgende Ausgabe:

```
111     222
111♂222
11222
111♀222
```

Dies lässt die folgenden Schlüsse zu:

- `\t` steht für *Tabulator*. Die nächste Ausgabe beginnt an der nächsten Tabulatorposition[2]. Der Ausgabetreiber für die Zeichenausgabe in die DOS-Box interpretiert das Zeichen also und positioniert die Schreibmarke für die nächste Ausgabe auf die nächste Tab-Position.
- `\v` wird vom Treiber nicht interpretiert. Das entsprechende Zeichen des Fonts wird ausgegeben.
- `\b` steht für *Backspace.* Der Treiber positioniert die Schreibmarke um eine Stelle zurück: die nächste Ausgabe überschreibt das letzte Zeichen (hier die dritte 1).
- `\f` wird vom Treiber nicht interpretiert. Das entsprechende Zeichen des Fonts wird ausgegeben.

Beachten Sie bitte, dass die Interpretation der Escape-Sequenzen nicht vom C++-Programm, sondern vom Ausgabetreiber des jeweiligen Mediums (hier vom Treiber für die DOS-Box) vorgenommen wird.

[2] Für DOS-Boxen unter Windows ist der Tabulator fest auf 8 Zeichen eingestellt.

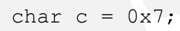
Vollständiges Programm:

```
#include <iostream>

using namespace std;

//-------------------------------------------------------------------
//       main
//
int main()
{
  cout << "111" << '\t' << "222" << endl;
  cout << "111" << '\v' << "222" << endl;
  cout << "111" << '\b' << "222" << endl;
  cout << "111" << '\f' << "222" << endl;


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 3-9:

Wie sieht die Anweisung aus, wenn man an Stelle der dezimalen Schreibweise die hexadezimale Notation für die Escape-Sequenz verwenden will?

In hexadezimaler Notation schreibt man



```
char c = 0x7;
```

Übung 3-10:

Wie lautet die Anweisung, um `c` einen umgekehrten Schrägstrich zuzuweisen?

Man schreibt

```
char c = '\\';
```

um einen umgekehrten Schrägstrich zu erzeugen. Die Ausgabeanweisung

```
cout << c << endl;
```

zeigt dies.

Anmerkung: Diese Umsetzung wird vom Compiler durchgeführt, und muss deshalb in allen Zeichenketten beachtet werden. Die folgende Anweisung produziert wahrscheinlich nicht das gewünschte Ergebnis:

```
string fileName = "c:\tmp\temporary.dat";
```

Korrekt sollte es sicherlich

```
string fileName = "c:\\tmp\\temporary.dat";
```

heißen.

Übung 3-11:

Analysieren Sie die beiden genannten Möglichkeiten zur Repräsentation von Strings. Wo liegen die wesentlichen Unterschiede? Ergeben sich daraus Konsequenzen für die Programmierung?

Beide Möglichkeiten haben ihre Vor- und Nachteile.

Die Codierung in C und C++ mit der nachgestellten 0 erlaubt problemlos Strings beliebiger Länge. Dafür ist jedoch die Bestimmung der Stringlänge aufwendiger: in einer Schleife müssen alle Zeichen untersucht werden, bis die anhängende 0 als Endemarkierung gefunden wird.

Da die C/C++Codierung das Nullzeichen als Ende-Marker verwendet, kann das Nullzeichen selber nicht im String vorkommen.

Bei der Pascal-Codierung (mit voran gestellter Längeninformation) muss man sich entscheiden, wie viele Bytes man für die Längeninformation „spendieren“ will. Da die allermeisten Zeichenketten kürzer als 256 Bytes sind, kommt man normalerweise mit einem Byte aus.

Das Problem hierbei liegt im Wort *normalerweise*: Sind doch einmal längere Strings erforderlich, können sie mit dem Stringtyp der Sprache nicht dargestellt werden. Eine Alternative wäre dann ein zweiter Stringtyp mit zwei Längenbytes. Es wäre Aufgabe des Programmierers, den richtige Typ zu verwenden. Insbesondere muss beachtet werden, dass Strings ja auch dynamisch länger werden können – dann wäre bei Überschreiten der Längengrenze ein Umspeichern in eine andere Variable erforderlich.

Man könnte grundsätzlich Strings mit 2-Byte-Längeninformation verwenden. Dann wäre jedoch der Overhead bei den normalerweise vorkommenden kurzen bis sehr kurzen Strings zu groß.

Betrachtet man Speicherplatz als knappe Resource, erscheint die C/C++-Codierung die bessere Alternative zu sein – sicher auch ein Grund, warum für C diese Art der Codierung gewählt wurde. Heute spielt Speicher keine so große Rolle mehr, und Stringklassen wir z.B. die `std::string` verwenden einen eigenen integer-Wert zur Führung der Länge.

Übung 3-12:

Wie muss die Definition von `str` geschrieben werden, damit bei der Ausgabe die Namen jeweils in einer neuen Zeile stehen?

Man muss die Escape-Sequenz für Zeilenumbruch zwischen die einzelnen String-Anteile platzieren:

```
char str[] =

  "Müller\n"
  "Meier\n"
  "Schulze";
```

Anmerkung 1: Mit gleicher Wirkung kann man auch

```
  char str[] =  "Müller\nMeier\nSchulze";
```

schreiben.

Anmerkung 2: Auf meinem System erhält man als Ausgabe

```
  M³ller
  Meier
  Schulze
```

d.h. die deutschen Umlaute werden nicht richtig dargestellt. Der Grund liegt in der Tatsache, dass der Programmtext unter Windows eingegeben wurde, die Ausgabe aber in einer DOS-BOX erfolgt. Hier werden unterschiedliche Fonts verwendet.

Vollständiges Programm

```
#include <iostream>

using namespace std;

//----------------------------------------------------------------------
//       main
//
int main()
{

char str[] =

  "Müller\n"
  "Meier\n"
  "Schulze";

  cout << str << endl;


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 3-13:

Schreiben Sie ein Programm, dass die Größen aller fundamentalen Datentypen aus obigen Tabellen bestimmt und ausgibt.

Folgende Ausgabeanweisungen leisten das Gewünschte:

```
cout << "sizeof( char )            " << sizeof( char )        << endl;
cout << "sizeof( short int )       " << sizeof( short int )   << endl;
cout << "sizeof( int )             " << sizeof( int )         << endl;
cout << "sizeof( long int )        " << sizeof( long int )    << endl;
cout << "sizeof( wchar_t )         " << sizeof( wchar_t )     << endl;
cout << "sizeof( float )           " << sizeof( float )       << endl;
cout << "sizeof( double )          " << sizeof( double )      << endl;
cout << "sizeof( long double )     " << sizeof( long double ) << endl;
```

Als Ergebnis erhält man z.B. mit MSVC6:

```
sizeof( char )            1
sizeof( short int )       2
sizeof( int )             4
sizeof( long int )        4
sizeof( wchar_t )         2
sizeof( float )           4
sizeof( double )          8
sizeof( long double )     8
```

Anmerkung: Beachten Sie bitte, dass die Ausgabeanweisung durch Verwendung von *horizontalem Whitespace* (i.e. Leerzeichen bzw. Tabulatoren) optisch ansprechend notiert wurden. Vergleichen Sie dazu die Lesbarkeit folgender Schreibweise der Anweisungen:

```
cout<<"sizeof(char)"<<sizeof(char)<<endl;
cout<<"sizeof(short int)"<<sizeof(short int)<<endl;
cout<<"sizeof(int)"<<sizeof(int)<<endl;
cout<<"sizeof(long int"<<sizeof(long int)<<endl;
cout<<"sizeof(wchar_t)"<<sizeof(wchar_t)<<endl;
cout<<"sizeof(float)"<<sizeof(float)<<endl;
cout<<"sizeof(double)"<<sizeof(double)<<endl;
cout<<"sizeof(long double)"<<sizeof(long double)<<endl;
```

Die zweite Formatierung ist deutlich schlechter lesbar. Leider findet man eine solche Ignoranz gegenüber einem (späteren) Leser von Sourcecode häufig in kommerzieller wie privater Software.

Vollständiges Programm

```
#include <iostream>

using namespace std;

//----------------------------------------------------------------------
//       main
//
int main()
{
  cout << "sizeof( char )            " << sizeof( char )        << endl;
  cout << "sizeof( short int )       " << sizeof( short int )   << endl;
  cout << "sizeof( int )             " << sizeof( int )         << endl;
  cout << "sizeof( long int )        " << sizeof( long int )    << endl;
  cout << "sizeof( wchar_t )         " << sizeof( wchar_t )     << endl;
  cout << "sizeof( float )           " << sizeof( float )       << endl;
  cout << "sizeof( double )          " << sizeof( double )      << endl;
  cout << "sizeof( long double )     " << sizeof( long double ) << endl;

  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;
  return 0;
}
```

KAPITEL 4

Übung 4-1:

Ändern Sie die Ausgabe so ab, dass möglichst viele Nachkommastellen ausgegeben werden und beobachten Sie die Rundungseffekte bei beiden Rechnungen!

Aus der Übung 3-1 wissen wir, dass die Genauigkeit auf 16 Stellen begrenzt ist. Die Angabe von weiteren Dezimalstellen bei der Ausgabe bring nicht mehr Genauigkeit, sondern es werden höchstens Nullen ergänzt.

Die Ausgabeanweisungen lauten also

```
cout.precision( 16 );
cout << "Der Wert von i3 ist " << i3 << endl;
cout << "Der Wert von d3 ist " << d3 << endl;
```

Als Ausgabe erhält man

```
Der Wert von i3 ist -75
Der Wert von d3 ist -75
```

Hier hat die Verwendung der Fließkommarithmetik offensichtlich keine Nachteile bei der Genauigkeit gebracht. Rundungsfehler traten nicht auf.

Das Beispiel zeigt weiter, dass die Angabe der Genauigkeit über die Funktion precision für alle nachfolgenden Ausgaben gilt. Man stellt das Standardverhalten durch die Angabe von 0 für die Genauigkeit wieder her:

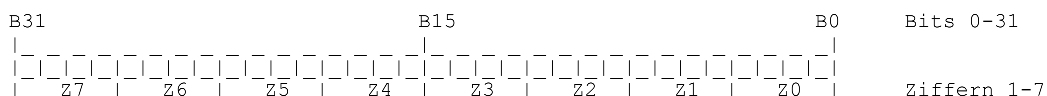
```
cout.precision( 16 );  // für nachfolgende Ausgaben: Standardgenauigkeit
```

Übung 4-2:

Welche Geldbeträge können maximal mit einem `long` ausgedrückt werden, wenn in Einheiten von 1/100 Cent gerechnet wird? Ist dieser Wertebereich für Finanzanwendungen ausreichend?

Auf einer Standard-Architektur mit 32 Bit kann eine Größe vom Typ `long` Werte zwischen –2147483648 und 2147483647 annehmen. Wenn also in Einheiten zu 1/100 Cent gerechnet wird, sind dies –21474836 bis 21474836 Cent, oder –214748 bis 214748 Euro.

Dies ist natürlich für Finanzanwendungen zu wenig. Zum Rechnen mit Geldbeträgen verwendet man daher besser eine andere Repräsentation von Werten. Durchgesetzt hat sich hier der *Binary Coded Decimal (BCD)*-Standard. Dabei wird jede einzelne Ziffer einer Zahl durch eine Anzahl Bits in einem Maschinenwort repräsentiert. Im *packed BCD*-Format zum Beispiel wird jede Ziffer durch 4 Bits codiert. In einer 32-Bit Größe können damit 7 Ziffern unter gebracht werden:



Rechnungen sind komplizierter als mit Standard-`ints` oder `longs`, dafür können mehrere BCD-Speicherblöcke aneinander gereiht werden, um mehr als 7 Ziffern verlustfrei repräsentieren zu können. Insgesamt erhält man eine schlechtere Speicherausnutzung als mit fundamentalen Typen, dafür kann man die BCDs beliebig kaskadieren.

Vollständiges Programm

```
#include <iostream>

using namespace std;

//-------------------------------------------------------------------
//       main
//
int main()
{

  int i1 = 2 + 3;
  int i2 = i1 * 2;
  int i3 = ( i1 + i2 ) * ( i1 - i2 );

  double d1 = 2.0 + 3.0;
  double d2 = d1 * 2.0;
  double d3 = ( d1 + d2 ) * ( d1 - d2 );


  cout.precision( 16 );
  cout << "Der Wert von i3 ist " << i3 << endl;
  cout << "Der Wert von d3 ist " << d3 << endl;


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 4-3:

Welche Arithmetik(en) werden bei der Auswertung des Ausdrucks

```
( 2 / 3 ) * 4.0
```

verwendet? Prüfen Sie Ihr Ergebnis mit Hilfe einer Ausgabeanweisung!

Der in Klammern stehende Teilausdruck wird zuerst berechnet. Da die beiden Operanden 2 und 3 ganze Zahlen sind, wird Integerarithmetik verwendet. Das Ergebnis der Division ist die ganze Zahl 0.

Im zweiten Schritt wird die Multiplikation durch geführt. Hier ist ein Operand eine ganze Zahl, der andere eine Fließkommazahl, es wird also Fließkommaarithmetik verwendet. Das Ergebnis ist die Fließkommazahl 0.0.

Man kann dies leicht durch die Anweisung

```
cout << ( 2 / 3 ) * 4.0 << endl;
```

verifizieren, die die Ausgabe 0 produziert.

Übung 4-4:

Gibt es einen Unterschied im Ergebnis durch die Verwendung unterschiedlicher Genauigkeiten?

Hier geht es um die Anweisungen

```
double d1 = 3.0 / 4;   // normale Genauigkeit: double
double d2 = 3.0f / 4;  // einfache Genauigkeit: float
double d3 = 3.0l / 4;  // erhöhte Genauigkeit: long double
```

Um eventuelle Unterschiede in der Genauigkeit fest zu stellen, stellen wir die Ausgabe auf maximale Genauigkeit ein und geben die drei Werte aus:

```
cout.precision( 16 );
cout << "d1: " << d1 << endl;
cout << "d2: " << d2 << endl;
cout << "d3: " << d3 << endl;
```

Als Ergebnis erhält man (wie wahrscheinlich bereits erwartet)

```
d1: 0.75
d2: 0.75
d3: 0.75
```

Es gibt bei dieser Rechnung also keinen Unterschied in der Genauigkeit.

Übung 4-5:

Ist der Ausdruck `++i3++` zulässig? Was bewirkt er?

Der Ausdruck ist unzulässig.

Aus der Prioritätstafel für Operatoren (Anhang 1 des Buches, Seite 1020) entnehmen wir, dass die Präfix-Form die Priorität 3 und die Postfix-Form die Priorität 2 hat.

Es wird also zuerst `i3++` berechnet. Der Wert von i3 wird erhöht, das Ergebnis der Operation ist jedoch der ursprüngliche Wert von `i3`. Auf dieses Ergebnis soll nun das Präfix-Increment angewendet werden. Der Operator benötigt jedoch eine Variable (*l-value*), nicht nur einen Wert (*r-value*), den wir hier aus dem Post-Increment zur Verfügung haben.

Aus dem gleichen Grunde ist z.B. auch die Anweisung

```
i++++; // Fehler!
```

unzulässig.

Übung 4-6:

Ausgehend vom letzten Beispiel, welchen Wert erhalten die Variablen `i1`, `i2` und `i3` in den folgenden beiden Anweisungen?

```
( i2 = i1 ) /= 3;
i3 = i1 /= 3;
```

Die Anweisungen im letzten Beispiel waren:

```
int i1, i2, i3;        // Definition von 3 Variablen

i1 = i2 = i3 = 0;      // kaskadierte Zuweisung: alle drei erhalten den
                       // Wert 0

i1 += 3;               // entspricht i1 = i1 + 3
i1 *= 2;               // i1 erhält den Wert 6
```

Nach diesen Anweisungen haben die Variablen die folgenden Werte:

Variable	*Wert*
`i1`	6
`i2`	0
`i3`	0

In der Anweisung

```
( i2 = i1 ) /= 3;
```

wird zunächst der Ausdruck in der Klammer berechnet. `i2` erhält also den Wert `6`:

Variable	*Wert*
`i1`	6
`i2`	**6**
`i3`	0

Das Ergebnis der Klammer ist jedoch nicht der Wert `6` (*r-value*), sondern die Variable `i2` (*ein l-value*). Auf dieses Ergebnis wird nun die Operation `/=3` angewendet. Die Variable `i2` erhält den Wert `2`:

Variable	*Wert*
`i1`	6
`i2`	**2**
`i3`	0

Nun folgt die Anweisung

```
i3 = i1 /= 3;
```

Hier wird zuerst die Division durch geführt. Als Ergebnis erhält `i1` den Wert `2`:

Variable	*Wert*
`i1`	**2**
`i2`	2
`i3`	0

Im zweiten Schritt erfolgt die Zuweisung an `i3`, sodass auch `i3` den Wert `2` erhält:

Variable	*Wert*
`i1`	2
`i2`	2
`i3`	**2**

Übung 4-7:

Was liefert der Ausdruck `sizeof( 2*d )` *?*

Die vorausgegangenen Anweisungen waren

```
double d = 1;
cout << "sizeof double: " << sizeof d << endl;
```

Der Ausdruck `2*d` multipliziert eine Fließkommazahl mit `2`. Das Ergebnis ist wiederum eine Fließkommazahl. Die beiden Anweisungen

```
cout << sizeof d << endl;
cout << sizeof( 2*d ) << endl;
```

liefern also den gleichen Wert, nämlich den Platzbedarf einer Fließkommazahl. Auf Systemen mit einer IEEE-754 – konformen Implementierung sind dies 8 Byte.

Übung 4-8:

Schreiben Sie einen Ausdruck, bei dem die Reihenfolge der Auswertung der Operatoren eine Rolle spielt.

Betrachten wir dazu die Anweisungen

```
int i=2;
int j = ++i * ( ++i + 1 );
```

Je nachdem, zu welchen Zeitpunkten die Inkrementierung von i erfolgt, sind die folgenden Ergebnisse möglich:

Rechenweg	***Ergebnis***
`3 * ( 4 + 1 )`	15
`4 * ( 3 + 1 )`	16
`4 * ( 4 + 1 )`	20

Visual C++ liefert z.B. den Wert 20.

Es ist klar, dass man solche Probleme am besten vermeidet.

KAPITEL 5

Übung 5-1:

Welche Auswirkungen hat die Operation `++a = ++b = 0`? (`a` und `b` sollen Integer-Variablen sein).

Zuerst werden die beiden Increment-Operatoren aus geführt, d.h. die Werte von `a` und `b` werden um `1` erhöht. Dann erfolgen die Zuweisungen, und zwar von rechts nach links. Erst erhält `b` den Wert `0`, dann `a`. Im Endergebnis haben beide Variablen den Wert `0`.

Beachten Sie bitte, dass die Anweisung

```
a++ = b++ = 0; // Fehler
```

nicht erlaubt ist. Der Postincrement-Operator liefert einen Wert zurück, und an einen Wert kann nichts zu gewiesen werden. Für eine Zuweisung benötigen wir einen l-value (also „die Variable selber“).

Übung 5-2:

Ist die Anweisungsfolge

```
{
   int i = 0;
}; // Semikolon nach schließendem Block
```

deshalb ein Syntaxfehler?

Nein, dies ist erlaubt, da das Semikolon einfach eine leere Anweisung bewirkt.

Übung 5-3:

Ist folgende Konstruktion syntaktisch zulässig?

```
{{
  int i = 0;
  i++;
}}
```

Ja, dies ist erlaubt. Blöcke können problemlos geschachtelt werden, und leere Blöcke sind zulässig. Der äußere Block ist hier einfach leer.

Übung 5-4:

Schreiben Sie die obige `if`-Anweisung so, dass in der Bedingung auf einen positiven Wert geprüft wird.

Die Original-Anweisung war

```
if ( i<0 )
{
  cout << " Wert kleiner 0: Vorzeichen umdrehen!" << endl;
  i =  -i;
}
else
{
  cout << "Wert gösser oder gleich 0: keine Aktion!" << endl;
}
```

„Anders herum“ formuliert sieht die Schleife so aus:

```
if ( i>=0 )
{
  cout << "Wert gösser oder gleich 0: keine Aktion!" << endl;
}
else
{
  cout << " Wert kleiner 0: Vorzeichen umdrehen!" << endl;
  i =  -i;
}
```

Übung 5-5:

Schreiben Sie die Anweisung der Übersichtlichkeit halber mit Blockklammern.

Die Anweisung war

```
if ( i != 0 )
  if ( i < 0 )
    cout << "i ist kleiner 0" << endl;
  else
    cout << "i ist grösser 0" << endl;
else
  cout << "i ist gleich 0" << endl;
```

Die Übersichtlichkeit steigt, wenn man zumindest die äußere `if`-Anweisung explizit klammert:

```
if ( i != 0 )
{
  if ( i < 0 )
    cout << "i ist kleiner 0" << endl;
  else
    cout << "i ist grösser 0" << endl;
}
else
{
  cout << "i ist gleich 0" << endl;
}
```

Ob man die innere `if`-Anweisung ebenfalls Klammern soll, ist Geschmacksache:

```
if ( i != 0 )
{
  if ( i < 0 )
  {
    cout << "i ist kleiner 0" << endl;
  }
  else
  {
    cout << "i ist grösser 0" << endl;
  }
}
else
{
  cout << "i ist gleich 0" << endl;
}
```

Übung 5-6:

Schreiben Sie die Anweisung so, dass die geschachtelte `if`-Abfrage im `else`-Zweig angeordnet ist.

Die Anweisung war

```
if ( i != 0 )
  if ( i < 0 )
    cout << "i ist kleiner 0" << endl;
  else
    cout << "i ist grösser 0" << endl;
else
  cout << "i ist gleich 0" << endl;
```

Damit die innere Abfrage in den `else`-Zweig wandert, muss die äußere Abfrage umgekehrt formuliert werden:

```
if ( i == 0 )
  cout << "i ist gleich 0" << endl;
else
  if ( i < 0 )
    cout << "i ist kleiner 0" << endl;
  else
    cout << "i ist grösser 0" << endl;
```

Übung 5-7:
Schreiben Sie diese `switch`-Anweisung mit Hilfe einer kaskadierten `if`-Anweisung

Die `switch`-Anweisung war

```
switch ( c )
{
  case 'a' :
  case 'A' : cout << "Es ist der Vokal a" << endl; break;

  case 'e' :
  case 'E' : cout << "Es ist der Vokal e" << endl; break;

  case 'i' :
  case 'I' : cout << "Es ist der Vokal i" << endl; break;

  case 'o' :
  case 'O' : cout << "Es ist der Vokal o" << endl; break;

  case 'u' :
  case 'U' : cout << "Es ist der Vokal u" << endl; break;


  default: cout << "Es ist kein Vokal" << endl;
}
```

Die entsprechende `if-else`-Kaskade sieht folgendermaßen aus:

```
  if ( c == 'a' || c == 'A' )
    cout << "Es ist der Vokal a" << endl;

  else if ( c == 'e' || c == 'E' )
    cout << "Es ist der Vokal e" << endl;

  else if ( c == 'i' || c == 'I' )
    cout << "Es ist der Vokal i" << endl;

  else if ( c == 'o' || c == 'O' )
    cout << "Es ist der Vokal o" << endl;

  else if ( c == 'u' || c == 'U' )
    cout << "Es ist der Vokal u" << endl;

  else
    cout << "Es ist kein Vokal" << endl;
```

Obwohl auch hier Leerzeilen zur Auflockerung des Codes verwendet wurden, ist die Version mit `case` besser lesbar.

Übung 5-8:
Stellen Sie fest, ob das Durchfallen auch bis zum `default`-Teil möglich ist.

Dies ist möglich, wie folgendes (korrektes) Programmsegment zeigt:

```
switch ( c )
{
  case 'a' :
  case 'A' :
  default  : cout << "Durchfallen bis zum default" << endl;

  case 'b' :
  case 'B' : cout << "Es ist ein b!" << endl;
}
```

Wie man sieht, muss der `default`-Teil nicht am Ende der Liste stehen.

Übung 5-9:

Was passiert, wenn man statt dessen den Kontrollausdruck `++i < 10` verwendet?

Die ursprüngliche Anweisung war

```
while( i++ < 10 )
  x*=2;
```

Bei der Auswertung des Kontrollausdrucks wird zuerst `i` mit `10` verglichen, dann wird `i` erhöht. Das Ergebnis des Vergleichs bestimmt, ob die Anweisung `x*=2` aus geführt wird.

Hat `i` also den Wert `10` erreicht, wird die Schleife beendet. Danach hat `i` den Wert `11`.

Schreibt man hingegen

```
while( ++i < 10 )
  x*=2;
```

wird bei der Auswertung des Kontrollausdrucks zuerst `i` erhöht, dann wird der Vergleich durch geführt.

Die Schleife wird also bereits bei einem Wert von `9` beendet, danach hat `i` den Wert `10`.

Übung 5-10:

Ist diese Version identisch zu den beiden vorigen Versionen der Schleife? Betrachten Sie dazu „interessante“ Werte für `d`.

Das Codesegment war

```
cout << "Bitte einen Wert eingeben (0 beendet) : " << endl;
cin >> d;

while ( d != 0.0 )
{
  cout << "Die Quadratwurzel ist: " << sqrt( d ) << endl;

  cout << "Bitte einen Wert eingeben (0 beendet) : " << endl;
  cin >> d;
}
```

Eine *statische Codeanalyse* zeigt, dass die beiden folgenden Fälle zu unterscheiden sind:

- `d` hat den Wert `0.0`
- `d` hat einen anderen Wert.

Im ersten Fall wird die Schleife nicht (mehr) durchlaufen, im zweiten Fall schon.

Vergleichen wir dazu die erste Version der Schleife:

```
double d;

do
{
  cout << "Bitte einen Wert eingeben (0 beendet) : " << endl;
  cin >> d;

  cout << "Die Quadratwurzel ist: " << sqrt( d ) << endl;
} while ( d != 0.0 );
```

Ein wesentlicher Unterschied ist, dass die Quadratwurzel auch für den Wert `0` noch berechnet wird, bevor die Schleife beendet wird[3]. Das Verhalten ist also nicht ganz identisch.

Ganz genau so verhält es sich mit der zweiten Version:

```
bool weiter = true;
while ( weiter )
{
  cout << "Bitte einen Wert eingeben (0 beendet) : " << endl;
  cin >> d;

  cout << "Die Quadratwurzel ist: " << sqrt( d ) << endl;

  weiter = d != 0.0;
}
```

Die beiden Versionen der Schleife sind also funktional nicht identisch zur dritten Version.

[3] Dies ist hier nicht weiter tragisch, da das Ziehen der Quadratwurzel für den Wert 0.0 explizit erlaubt ist. Es ist trotzdem unschön und vor allem deshalb ärgerlich, weil es eine korrekte Lösung gibt, die nicht aufwendiger ist. Leider findet man solche Schlampereien häufig in der Praxis.

Vollständiges Programm

```
#include <iostream>
#include <math.h>

using namespace std;

//----------------------------------------------------------------------
//       main
//
int main()
{

  double d;
  cout << "Bitte einen Wert eingeben (0 beendet) : " << endl;
  cin >> d;

  while ( d != 0.0 )
  {
    cout << "Die Quadratwurzel ist: " << sqrt( d ) << endl;

    cout << "Bitte einen Wert eingeben (0 beendet) : " << endl;
    cin >> d;
  }


  return 0;
}
```

Übung 5-11:

Machen Sie das Programm sicherer, indem Sie verhindern, dass die Wurzel von negativen Werten gebildet werden kann.

Dazu fügen wir eine geeignete `if`-Abfrage ein:

```
  while ( d != 0.0 )
  {
    if ( d > 0.0 )
      cout << "Die Quadratwurzel ist: " << sqrt( d ) << endl;
    else
      cout << "negative Werte nicht erlaubt!" << endl;


    cout << "Bitte einen Wert eingeben (0 beendet) : " << endl;
    cin >> d;
  }
```

Vollständiges Programm

```
#include <iostream>
#include <math.h>

using namespace std;

//----------------------------------------------------------------------
//       main
//
int main()
{

  double d;
  cout << "Bitte einen Wert eingeben (0 beendet) : " << endl;
  cin >> d;

  while ( d != 0.0 )
  {
    if ( d > 0.0 )
      cout << "Die Quadratwurzel ist: " << sqrt( d ) << endl;
    else
      cout << "negative Werte nicht erlaubt!" << endl;


    cout << "Bitte einen Wert eingeben (0 beendet) : " << endl;
    cin >> d;
  }

  return 0;
}
```

Übung 5-12:

Formulieren Sie die Schleife mit Hilfe einer `if`-Anweisung um die `continue`-Anweisung zu vermeiden. Welche Version ist leichter lesbar, wenn man von komplizierterem Code in der Schleife ausgeht?

Die Schleife war

```
for ( int i=0; i<10; i++ )
{
  if ( i%2 != 0 )
    continue; // ungerade Zahlen wollen wir nicht

  cout << i << " ist gerade" << endl;
}
```

„Positiv“ formuliert sieht die Schleife so aus:

```
for ( int i=0; i<10; i++ )
{
  if ( i%2 == 0 )
    cout << i << " ist gerade" << endl;
}
```

Dies ist besser lesbar als die abweisende („negative“) Formulierung in der oberen Version mit `continue`.

Grundsätzlich sollen Bedingungen so formuliert werden, dass sie das prüfen, was man will („positiv“), und nicht, was man nicht will („negativ“). Dadurch erhält man leichter verständlicheren Code.

Hat man jedoch mehrere Abfragen hinter einander, kann die sequentielle Anordnung abweisender Abfragen besser sein:

```
for ( int i=0; i<10; i++ )
{
  if ( i%2 != 0 )
    continue; // ungerade Zahlen wollen wir nicht

  if ( <Bedingung 2> )
    continue; // dies wollen wir auch nicht

  if ( <Bedingung 3> )
    continue; // und dies ebenfalls auch nicht

  ... code ...
}
```

Übung 5-13:

Erstellen Sie aus den Anweisungsfragmenten ein lauffähiges Programm. Formulieren Sie das Programm dann ohne `goto`. Hinweis: Verwenden Sie eine Kontrollvariable, die zunächst auf `false` steht und bei Erkennen der Abbruchanforderung auf `true` gesetzt wird.

Folgendes Programmsegment zeigt eine lauffähige Version der Schleife mit `goto`:

```
for ( int x=0; x<10; x++ )
{
  for( int y=0; y<10; y++ )
  {
    while( true )
    {
      if ( !calculate( x, y ) )
        goto ende;
    }
  }

ende: ;
}
```

Die innere Endlosschleife wird so lange ausgeführt, bis die Funktion `calculate` `false` liefert. Die `goto`-Anweisung verzweigt ans Ende der inneren Schleife, was dann zur nächsten Iteration innerhalb der äußeren Schleife mit einem um `1` erhöhten `x` führt.

Mit Hilfe einer zusätzlichen Kontrollvariablen kann man auf das `goto` verzichten:

```
for ( int x=0; x<10; x++ )
{
  bool done = false;
  for( int y=0; y<10 && !done; y++ )
  {
    while( !done )
    {
      done = !calculate( x, y ) );
    }
  }
}
```

Von Puristen wird diese Version mit der zusätzlichen Kontrollvariablen als überlegen an gesehen. Ich kann dem nicht so ganz folgen: die Version mit `goto` erscheint mir persönlich verständlicher zu sein.

Dies soll nicht für die Verwendung von `goto` im Allgemeinen sprechen, sondern nur dafür, `goto` nicht prinzipiell zu verbannen und grundsätzlich als „schlechten Stil“ zu brandmarken.

KAPITEL 6

Übung 6-1:

Schreiben Sie eine Funktion, die als Argument ein `bool` übernimmt und „true“ oder „false“ druckt. Verwenden Sie diese Funktion, um die Ausgabe in obiger Schleife besser zu gestalten.

Eine solche Funktion lässt sich einfach als

```
inline
string boolToString( bool value )
{
  return value ? "true" : "false";
}
```

notieren.

Beachten Sie bitte, dass `string` kein fundamentaler Datentyp ist, sondern eine Klasse aus der Standardbibliothek. Diese ist jedoch so formuliert, dass sie nahezu wie ein fundamentaler Typ verwendet werden kann. Bei der Rückgabe eines der beiden Zeichenkettenliterale aus der Funktion erfolgt automatisch die Umwandlung in das erforderliche `string`-Objekt.

Die Ausgabeanweisung formuliert man nun z.B. als

```
  for ( int i=0; i<5; i++ )
    cout << i << " ist gerade: " << boolToString( klassifiziere(i) ) << endl;
```

und erhält als Ergebnis

```
0 ist gerade: true
1 ist gerade: false
2 ist gerade: true
3 ist gerade: false
4 ist gerade: true
```

Vollständiges Programm

```
#include <iostream>
#include <string>

using namespace std;

//--------------------------------------------------------------------
//       boolToString
//
inline
string boolToString( bool value )
{
  return value ? "true" : "false";
}

//--------------------------------------------------------------------
//       klassifiziere
//
bool klassifiziere( int a )
{
  if ( a%2 == 0 )
    return true;
  else
    return false;
}

//--------------------------------------------------------------------
//       main
//
int main()
{

  for ( int i=0; i<5; i++ )
    cout << i << " ist gerade: " << boolToString( klassifiziere(i) ) << endl;

  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 6-2:

Was passiert, wenn beim Aufruf der Funktion die abschließende `-1` vergessen wird?

Schreibt man z.B.

```
int k = sum( 1, 2, 3 );
```

wird die Schleife zum Abholen der Werte vom Stack nicht nach dem dritten Wert terminiert, sondern versucht, weitere Werte vom Stack zu holen – so lange, bis im Speicher das Bitmuster für `-1` an getroffen wird. Das Verhalten des Programms ist also bestenfalls undefiniert.

Übung 6-3:

Die derzeitige Implementierung besitzt noch einen offensichtlichen Fehler: der Aufruf von `sum( -1 )` führt zu einem falschen Ergebnis. Korrigieren Sie die Implementierung entsprechend.

Im Buch ist bereits die korrekte Implementierung ab gedruckt. Ein Aufruf von

```
sum( -1 )
```

führt zum korrekten Ergebnis `0`.

Übung 6-4:

Schreiben Sie die Funktion so um, das die Anzahl der zu summierenden Werte im ersten Parameter übergeben wird, sodass die abschließende `-1` entfallen kann.

Folgende Implementierung leistet dies:

```
int sum( int anzahl, ... )
{
   int ergebnis = 0;

   va_list parameterListe;
   va_start( parameterListe, anzahl );

   for( int i=0; i<anzahl; i++ )
   {
     ergebnis += va_arg( parameterListe, int );
   }
   va_end( parameterListe );

   return ergebnis;
}
```

Vollständiges Programm:

```
#include <iostream>
#include <stdarg.h>

using namespace std;


//----------------------------------------------------------------------
//       sum
//
int sum( int anzahl, ... )
{
   int ergebnis = 0;

   va_list parameterListe;
   va_start( parameterListe, anzahl );

   for( int i=0; i<anzahl; i++ )
   {
     ergebnis += va_arg( parameterListe, int );
   }
   va_end( parameterListe );

   return ergebnis;
}


//----------------------------------------------------------------------
//       main
//
int main()
{

  cout << "Summe von 10, 11 und 12: " << sum( 3, 10, 11, 12 ) << endl;

  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 6-5:

Was passiert beim Aufruf der Funktion mit dem Wert `0`?

Es geht um folgende Funktion:

```
int fak( int i )
{
  if ( i==1 )
    return 1;
  else
    return i * fak( i-1 );
}
```

Bei einem Aufruf mit dem Wert `0` wird der `else`-Zweig der `if`-Anweisung ausgeführt, was zu einem erneuten Aufruf von `fak` mit einem um `1` erniedrigten Parameter führt. Da das Abbruchkriterium niemals erreicht wird, erhält man eine endlose Rekursion, die irgendwann auf Grund von Stacküberlauf abbricht (d.h. das Betriebssystem beendet das Programm zwangsweise).

Übung 6-6:

Jede rekursive Funktion kann auch iterativ (d.h. mit einer oder mehreren Schleifen) geschrieben werden. Implementieren Sie die Funktion `fak` mit Hilfe einer Schleife, sodass der rekursive Aufruf vermieden wird.

Folgende Funktion berechnet die Fakultät iterativ:

```
int fak( int argument )
{
  int ergebnis = 1;
  for ( int i=2; i<=argument; i++ )
    ergebnis*=i;

  return ergebnis;
}
```

Vollständiges Programm:

```
#include <iostream>

using namespace std;

//----------------------------------------------------------------------
//       fak
//
int fak( int argument )
{
  int ergebnis = 1;
  for ( int i=2; i<=argument; i++ )
    ergebnis*=i;

  return ergebnis;
}

//----------------------------------------------------------------------
//       main
//
int main()
{

  cout << "Fakultät von 4 ist: " << fak( 4 ) << endl;

  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

KAPITEL 7

Übung 7-1:

Ist das Programmsegment

```
double mult( int i, int j ); // Deklaration

double mult( int x, int y )   // Definition
{
   return i*j;
}
```

gültiger C++-Code?

Nein, dies ist nicht gültig. In der Funktionsdefinition werden für die Parameter die Namen `x` und `y` verwendet, und diese müssen auch in der Implementierung verwendet werden. Die Namen `a` und `b` werden bei der Deklaration verwendet und sind nur innerhalb dieser gültig.

Korrekt muss die Funktion also als

```
double mult( int x, int y )   // Definition
{
   return x*y;
}
```

Übung 7-2:

Ist das Programmsegment

```
double mult( int i, int j );
double mult( int x, int y );
```

gültiger C++-Code?

Dies ist korrektes C++. Es wird in beiden Fällen die gleiche Funktion deklariert. Der Name der Parameter spielt bei einer Funktionsdeklaration keine Rolle Genau so gut hätte man z.B. auch

```
double mult( int, int );
double mult( int, int );
```

schreiben können.

Übung 7-3:

Erweitern Sie das Modul um eine Funktion, um den Startwert für `val1` zu setzen, damit unterschiedliche Folgen produziert werden können. Experimentieren Sie mit unterschiedlichen Startwerten. Gibt es geeignete bzw. weniger geeignete Startwerte?

Die Funktion zum Setzen des Startwertes wird einfach als

```
void setStartValue( unsigned int value )
{
  val1 = value;
}
```

notiert. Die Funktion ist nicht statisch und kann deshalb auch aus anderen Modulen des Programms heraus auf gerufen werden.

Unterschiedliche Startwerte resultieren in unterschiedlichen Zufallsfolgen, die allerdings auf den ersten Blick alle ähnlich aussehen. Der Startwert `1` bringt z.B. diese Zahlen als die ersten 10 der Folge:

```
3079473478 3079473479 1863979662 648485846 2512465508 3160951354 1378449567
244433626 1622883193 1867316819
```

Analysiert man jedoch die Häufigkeiten, mit denen bestimmte Zahlen oder Ziffernkombinationen vor kommen, sieht man deutliche Unterschiede bei den verschiedenen Startwerten.

Folgender Ausdruck zeigt die Verteilung der Zufallszahlen, normiert auf die Werte 0 bis 100, für den vor eingestellten Startwert `0x5324879f`:

```
970 994 1000 958 999 985 1036 941 932 1008 932 1047 1027 968 1025 1037 1004 986
1014 975 1020 964 991 977 996 1026 970 1036 995 1034 1000 984 980 1004 993 972
1007 1004 1065 994 1024 982 1002 990 1026 964 1007 1044 970 979 1011 966 947 998
1013 965 1039 998 1017 935 1025 1060 1014 1038 961 1007 1040 1032 993 1009 1020
970 984 1064 1026 990 994 979 961 1037 1044 1017 1042 999 991 994 991 999 990 1011
1008 1009 997 1026 990 1006 919 997 1041 998
```

Wie man sieht, kommen alle Zahlen mit nahezu der gleichen Häufigkeit vor.

Verwendet man dagegen `1` als Startwert, erhält man folgende Verteilung:

```
0 1231 1232 1205 1240 0 1301 1273 1296 1230 0 1225 1222 1237 1271 0 1233 1264 1288
1252 0 1187 1304 1247 1238 0 1264 1213 1215 1228 0 1267 1161 1264 1194 0 1227
1258 1214 1265 0 1312 1252 1227 1223 0 1222 1214 1257 1316 0 1286 1251 1226 1242 0
1250 1296 1285 1262 0 1290 1231 1269 1235 0 1227 1280 1321 1264 0 1310 1289 1272
1208 0 1226 1305 1255 1336 0 1264 1249 1217 1300 0 1224 1201 1233 1220 0 1197 1263
1198 1211 0 1257 1242 1274 1265
```

Hier kann man deutliche „Löcher“ in der Verteilung sehen: der Startwert `1` ist also für einen guten Zufallszahlengenerator eher ungeeignet.

Vollständiges Programm:

Das folgende Programm erzeugt 100000 Zufallszahlen im Bereich 0 bis 100 und zählt die Häufigkeit jedes Wertes von 0 bis 100:

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

static unsigned int val1 = 0x5324879f;
static unsigned int val2 = 0xb78d0945;

//----------------------------------------------------------------------
//       zufallsZahl
//
unsigned int zufallsZahl()
{
  unsigned int summe = val1 + val2;

  if ( summe < val1 || summe < val2 )
    summe++;

  val2 = val1;
  val1 = summe;

  return summe;
}


//----------------------------------------------------------------------
//       setStartValue
//
void setStartValue( unsigned int value )
{
  val1 = value;
}



//----------------------------------------------------------------------
//       main
//
int main()
{

setStartValue( 1 );

//-- a ist das Feld für die Häufigkeit jeder Zahl im Bereich 1 bis 100.
//
int a[100];
for ( int i=0; i<100; i++ )
  a[i]=0;

for ( i=0; i<100000; i++ )
{
  a[ zufallsZahl() % 100 ] ++;
}

for ( i=0; i<100; i++ )
  cout << a[i] << " ";


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 7-4:

Die Zufallszahlen sind alle sehr groß. Erweitern Sie das Modul um eine weitere Funktion, die Zufallszahlen in einem Bereich `0..maxZufallsZahl` zurückliefert. Die obere Grenze `maxZufallszahl` soll durch den Benutzer des Moduls angegeben werden können. Hinweis: Hier kann der Modulo-Operator „`%`“sinnvoll eingesetzt werden.

Wir ergänzen das Modul um eine weitere Funktion, die als Parameter den maximal zu liefernden Wert erhält. Diese Funktion kann ebenfalls den Namen `zufallsZahl` erhalten, da beim Aufruf eine Unterscheidung an Hand der Parameterliste möglich ist:

```
unsigned int zufallsZahl( unsigned int maxValue )
{
  return zufallsZahl() % maxValue;
}
```

Die neue Funktion bedient sich der bereits vorhandenen Funktionalität, modifiziert aber das Ergebnis entsprechend.

Um Zahlen im Bereich 1 bis 50 zu erhalten, kann man nun z.B.

```
  for ( int i=0; i<10; i++ )
    cout << zufallsZahl( 50 ) << " ";
```

schreiben. Als Ergebnis erhält man die Folge

```
25 18 43 11 9 20 34 9 43 2
```

Vollständiges Programm:

```
#include <iostream>

using namespace std;

static unsigned int val1 = 0x5324879f;
static unsigned int val2 = 0xb78d0945;

//---------------------------------------------------------------------
//       zufallsZahl
//
unsigned int zufallsZahl()
{
  unsigned int summe = val1 + val2;

  if ( summe < val1 || summe < val2 )
    summe++;

  val2 = val1;
  val1 = summe;

  return summe;
}

//---------------------------------------------------------------------
//       zufallsZahl
//
unsigned int zufallsZahl( unsigned int maxValue )
{
  return zufallsZahl() % maxValue;
}


//---------------------------------------------------------------------
//       setStartValue
//
void setStartValue( unsigned int value )
{
  val1 = value;
}



//---------------------------------------------------------------------
//       main
//
int main()
{

  for ( int i=0; i<10; i++ )
    cout << zufallsZahl( 50 ) << " ";


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 7-5:

Schreiben Sie schließlich eine weitere Funktion, die Zufallszahlen im Bereich `0..1` als `double`-Werte zurückliefert.

Auch hier verwenden wir eine weitere Funktion, die allerdings nun einen anderen Namen erhalten muss, da sie sich von den vorhandenen Funktionen nur im Rückgabetyp unterscheidet:

```
//--------------------------------------------------------------------
//       zufallsZahlAsDouble
//
double zufallsZahlAsDouble()
{
  return (double)zufallsZahl() / UINT_MAX;
}
```

Bei der Implementierung gehen wir davon aus, dass die von `zufallsZahl` gelieferten Werte im Bereich `0.. UINT_MAX` gleichverteilt sind. Die Wandlung nach `double` vor der Division ist erforderlich, damit Fließkommaarithmetik verwendet wird. Als Ergebnis erhält man Werte zwischen `0` und `1`.

Folgende Schleife gibt 10 solche Zahlen aus:

```
  for ( int i=0; i<10; i++ )
    cout << zufallsZahlAsDouble() << " ";
```

Als Ergebnis erhält man

```
0.0417719 0.366548 0.40832 0.774868 0.183188 0.958056 0.141244 0.0993007
0.240545 0.339846
```

Vollständiges Programm:

```
#include <iostream>
#include <limits.h>

using namespace std;

static unsigned int val1 = 0x5324879f;
static unsigned int val2 = 0xb78d0945;

//----------------------------------------------------------------------
//       zufallsZahl
//
unsigned int zufallsZahl()
{
  unsigned int summe = val1 + val2;

  if ( summe < val1 || summe < val2 )
    summe++;

  val2 = val1;
  val1 = summe;

  return summe;
}

//----------------------------------------------------------------------
//       zufallsZahl
//
unsigned int zufallsZahl( unsigned int maxValue )
{
  return zufallsZahl() % maxValue;
}

//----------------------------------------------------------------------
//       zufallsZahlAsDouble
//
double zufallsZahlAsDouble()
{
  return (double)zufallsZahl() / UINT_MAX;
}

//----------------------------------------------------------------------
//       setStartValue
//
void setStartValue( unsigned int value )
{
  val1 = value;
}

//----------------------------------------------------------------------
//       main
//
int main()
{

  for ( int i=0; i<10; i++ )
    cout << zufallsZahlAsDouble() << " ";


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 7-6:

Verwenden Sie nun die neuen Funktionen, um 1000 zufällige Buchstaben aus dem Bereich `a-z` zu erzeugen und auszugeben. Hinweis: Der Typ `char` ist ein integraler Typ und kann daher in arithmetischen Ausdrücken verwendet werden.

Zur Lösung dieser Aufgabe benötigen wir Zufallszahlen im Bereich von `'a'` bis `'z'`. Wir erhalten eine solche Zahl durch den Aufruf von

```
(char)(zufallsZahl( 'z'-'a' ) + 'a' )
```

Beachten Sie bitte, wie die maximal zulässige Zufallszahl bestimmt wird. Zunächst liefert der Ausdruck `'z'-'a'` die Anzahl der Buchstaben im Alphabet[4]. Zufallszahlen aus diesem Bereich müssen vom Wert her nach `'a'` angeordnet werden, daher erfolgt die Transposition der Zahlen um den numerischen Wert von `'a'`. Der so erhaltene Wert wird schließlich wieder als Zeichen interpretiert – dazu dient die explizite Wandlung zum Typ `char`.

Um 1000 zufällige Buchstaben zu erhalten, schreibt man also einfach

```
  for ( int i=0; i<1000; i++ )
    cout << (char)(zufallsZahl( 'z'-'a' ) + 'a' );
```

und erhält als Ergebnis die Ausgabe

```
assljujjscuwwyvabbcipxsqokyjiwkhryquqlcsusslepyoshfrwolalqhdkndvefjoxrummyqplgrx
pndqypojdrawwyvuvvwxaxdgjpetdcfhmyqplgrxusnleuytxwuwwtqplgrxuxsqokeosmfrcybabgms
kisbyayeirarwtvplgrxussleueenwkhwjlullwikxnlykoescachovklvhiuidqypoescuwwybaglrd
uddgoujjxhfrctbabbcdknxqtpoescuwwtvpqgcikxnqjftedhpwmogallcnunivefjodwachjqfbgms
kisbtutoicprhequllcsunngtfejshahhobpvqmdpxsqokyjncuwrtlkblmdpxnqeueenwpmhygflvhd
pxnqjajoxmpcryvabbcdknxqtpjenrkhrebflqcxfdilypoeswuwrtlfvghsfxdgoattnmfrcybflqcs
assqoftyxwawwybflqcxaddgoattnmfwcelpbqwspidqypojdrummevavbciksdbjktjirfchjvklvmi
uiivjfoyswpmhygkvgcnpdxgjueydhkrheqaqqmiudxbektexhkrcybaglrifnsgeptoiwkmcoqkblri
anngyfejncuwwyvavvwxaxdgjpyosmfrwtvplgwiksdvefjodrawwtqkbqwnkxilykoescaccjqfvghn
uiiqeaeeirfwhjqaqqhdpxnleueeimarrjbkqgcikxnqeueeimammequqqhxknxqtketxrurrjgubvcx
adilykjynmfwcegkvlhsfdiqeueyimuhcjlaqqhxknxqtketxrurrjbpvlmdpsnlykoynrkhryvuvqmi
ansljueeimarwolfvbciksibjpetxrummequlgwiksdbjpytsmkcrtlkvghsassqofyjirarwtvpqlcn
pdxgeptjiwkhrevabbcikxiltfeoxrurrjgpvqrn
```

[4] Die Differenz wird übrigens bereits vom Compiler gebildet, und nicht bei jedem Funktionsaufruf zur Laufzeit. Dies ist möglich, da in dem Ausdruck nur Übersetzungszeitkonstanten vor kommen, der Ausdruck mithin selber eine Übersetzungszeitkonstante ist.

Vollständiges Programm:

```
#include <iostream>
#include <limits.h>

using namespace std;

static unsigned int val1 = 0x5324879f;
static unsigned int val2 = 0xb78d0945;

//----------------------------------------------------------------------
//       zufallsZahl
//
unsigned int zufallsZahl()
{
  unsigned int summe = val1 + val2;

  if ( summe < val1 || summe < val2 )
    summe++;

  val2 = val1;
  val1 = summe;

  return summe;
}

//----------------------------------------------------------------------
//       zufallsZahl
//
unsigned int zufallsZahl( unsigned int maxValue )
{
  return zufallsZahl() % maxValue;
}

//----------------------------------------------------------------------
//       zufallsZahlAsDouble
//
double zufallsZahlAsDouble()
{
  return (double)zufallsZahl() / UINT_MAX;
}

//----------------------------------------------------------------------
//       setStartValue
//
void setStartValue( unsigned int value )
{
  val1 = value;
}

//----------------------------------------------------------------------
//       main
//
int main()
{

  for ( int i=0; i<1000; i++ )
    cout << (char)(zufallsZahl( 'z'-'a' ) + 'a' );


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 7-7:

Verändern Sie den Aufrufzähler so, dass die Anzahl der Aufrufe über eine weitere Funktion (z.B. int `f_count()`) abgefragt werden kann. Verallgemeinern Sie das Konzept für mehrere Funktionen.

Eine solche Abfrage erfordert, dass die zum Zählen notwendige Variable nicht lokal statisch, sondern global statisch definiert wird (sonst hätte man ja außerhalb der Funktion keinen Zugriff). Dies wiederum beschränkt die Anzahl der möglichen Funktionen, deren Aufruf zu zählen ist, auf 1.

Zur Lösung des Problems kann man ein (statisches) Feld von Zählern anlegen und jeder Funktion einen „Slot“ in diesem Feld zuordnen. Folgendes Beispiel zeigt die Vorgehensweise:

```
//-- Aufrufzähler für mehrer Funktionen
//
static const int maxFunctions = 10;
static int counterArray[ maxFunctions ];
```

Folgende Funktion erledigt den Ausdruck:

```
void printCounterArray()
{
  for ( int i=0; i<maxFunctions; i++ )
    cout << "Funktion Nr. " << i
         << " Aufrufhäufigkeit: " << counterArray[i] << endl;
}
```

Nun muss man den Funktionen, deren Aufruf man zählen will, eindeutige Indizes zuordnen. Folgendes Listing zeigt dies beispielhaft für zwei Funktionen `f1` und `f2`:

```
void f1()
{
  counterArray[0]++;

  /* ... eigentliche Implementierung */
}

void f2()
{
  counterArray[1]++;

  /* ... eigentliche Implementierung */
}
```

Schreibt man nun

```
  f1();
  f1();
  f2();
  printCounterArray();
```

erhält man als Ausgabe

```
Funktion Nr. 0 Aufrufhõufigkeit: 2
Funktion Nr. 1 Aufrufhõufigkeit: 1
Funktion Nr. 2 Aufrufhõufigkeit: 0
Funktion Nr. 3 Aufrufhõufigkeit: 0
Funktion Nr. 4 Aufrufhõufigkeit: 0
Funktion Nr. 5 Aufrufhõufigkeit: 0
Funktion Nr. 6 Aufrufhõufigkeit: 0
Funktion Nr. 7 Aufrufhõufigkeit: 0
Funktion Nr. 8 Aufrufhõufigkeit: 0
Funktion Nr. 9 Aufrufhõufigkeit: 0
```

Dieser Ansatz ist natürlich für die Praxis nicht zu gebrauchen. Insbesondere die Notwendigkeit zur eindeutigen Zuordnung von Indizes zu Funktionen ist störend und fehleranfällig. Besser wäre es, wenn man statt der numerischen Indizes deskriptive Zei-

chenketten verwenden könnte. Mit den Mitteln der Standardbibliothek ist dies einfach möglich. Anstelle des Feldes verwenden wir nun eine *Zuordnung* (*map*).

```
map< string, int > counterMap;
```

Der Container kann also mit einem String indiziert werden, als Ergebnis steht eine Referenz auf ein int. Der Operator [] der map ist so überladen, dass man z.B.

```
counterMap["abc"]=3;
int v = counterMap["abc"];
```

schreiben kann. `v` erhält in diesem Fall den Wert `3`. An Stelle der numerischen Indizes werden nun Zeichenketten zur Adressierungen der „Slots“ verwendet.

Eine weitere positive Eigenschaft einer map ist, dass man keinen Speicher für die Elemente anlegen muss. Ein Ausdruck wie z.B. `map["abc"]` legt automatisch einen neuen Slot an, wenn der betreffende Slot noch nicht vorhanden ist.

Folgendes Listing zeigt, wie die Funktionen `f1` und `f2` nun notiert werden:

```
void f1()
{
  counterMap["f1"]++;

  /* ... eigentliche Implementierung */
}

void f2()
{
  counterMap["f2"]++;

  /* ... eigentliche Implementierung */
}
```

Die Schlüssel `"f1"` und `"f2"` werden ebenfalls in der map gespeichert. Verwendet man die Funktionsnamen als Schlüssel, erhält man in der Ausgabe eine direkte, lesbare Zuordnung zur Funktion. Die Ausgabefunktion wird als

```
void printCounterMap()
{
  for ( map< string, int >::const_iterator cit = counterMap.begin();
        cit != counterMap.end();
        ++cit
      )

    cout << "Funktion " << setw(20) << cit->first
         << " Aufrufhäufigkeit: "   << cit->second << endl;
}
```

notiert. Mit den Aufrufen

```
f1();
f1();
f2();
printCounterMap();
```

erhält man nun die Ausgabe

```
Funktion                   f1    Aufrufhõufigkeit: 2
Funktion                   f2    Aufrufhõufigkeit: 1
```

Vollständiges Programm (mit Feld)

```
#include <iostream>

using namespace std;

//-- Aufrufzähler für mehrer Funktionen
//
static const int maxFunctions = 10;
static int counterArray[ maxFunctions ];

void printCounterArray()
{
  for ( int i=0; i<maxFunctions; i++ )
    cout << "Funktion Nr. " << i
         << " Aufrufhäufigkeit: " << counterArray[i] << endl;
}


//----------------------------------------------------------------------
//       f1 und f2
//
void f1()
{
  counterArray[0]++;

  /* ... eigentliche Implementierung */
}

void f2()
{
  counterArray[1]++;

  /* ... eigentliche Implementierung */
}

//----------------------------------------------------------------------
//       main
//
int main()
{

  f1();
  f1();
  f2();
  printCounterArray();

  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Vollständiges Programm (mit map)

```
#include <iostream>
#include <iomanip>
#include <map>
#include <string>

using namespace std;

//-- Aufrufzähler für mehrer Funktionen
//
map< string, int > counterMap;

void printCounterMap()
{
  for ( map< string, int >::const_iterator cit = counterMap.begin();
        cit != counterMap.end();
        ++cit
      )

    cout << "Funktion " << setw(20) << cit->first
         << " Aufrufhäufigkeit: "   << cit->second << endl;
}


//-------------------------------------------------------------------
//       f1 und f2
//
void f1()
{
  counterMap["f1"]++;

  /* ... eigentliche Implementierung */
}

void f2()
{
  counterMap["f2"]++;

  /* ... eigentliche Implementierung */
}

//-------------------------------------------------------------------
//       main
//
int main()
{

  f1();
  f1();
  f2();
  printCounterMap();

  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Anmerkung:

Wie immer bei der Verwendung der Standardbibliothek kann es vorkommen, dass Bezeichner länger als 255 Zeichen werden. Dies hängt mit der Instanziierung der diversen Schablonen zusammen, die die Mechanik der Standardbibliothek intern erfordert. Manche Compiler geben in diesem Fall Warnungen aus, die jedoch für die Funktion der Beispiele ohne Bedeutung sind.

Für VC kann man die Warnung über abgeschnittene Namen durch die folgende Pragma-Direktive abschalten:

```
#pragma warning( disable : 4786 )
```

Diese Zeile sollte grundsätzlich vor der Einbindung von Dateien der Standarbibliothek stehen.

Übung 7-8:

Verändern Sie den Rekursionszähler so, dass im Falle einer endlosen Rekursion eine Meldung ausgegeben wird. Verwenden Sie z.B. `100` als Limit für die Rekursionstiefe.

Die ursprüngliche Funktion zum Test des Rekursionszählers war

```
void f()
{
  static int count = 0;
  cout << "Rekursiontiefe: " << count << endl;

  count++;

  ... // weitere Implementierung f

  count--;
}
```

Um eine endlose Rekursion fest zu stellen, reicht es aus, den Wert von `count` zu verfolgen:

```
void f()
{
  static int count = 0;
  if ( count >= 100 )
  {
    cout << "*** Limit von 100 rekursiven Aufrufen erreicht!" << endl;
    exit(1);
  }

  count++;

  ... // weitere Implementierung f

  count--;
}
```

Übung 7-9:

Verändern Sie den Rekursionszähler so, dass bei Beendigung der Funktion die maximal erreichte Rekursionstiefe ausgegeben wird.

Um die maximal Rekursionstiefe fest zu stellen, muss man sich den maximalen Wert des Rekursionszählers merken, wie hier am Beispiel der Fakultät gezeigt:

```
static countMax = 0;

int fak( int i )
{
  static int count = 0;
  if ( count > countMax )
    countMax = count;

  count++;

  int ergebnis;

  if ( i==1 )
    ergebnis = 1;
  else
    ergebnis = i * fak( i-1 );

  count--;
  return ergebnis;
}
```

Schreibt man nun z.B.

```
  int ergebnis = fak( 5 );
  cout << "maximale Rekursionstiefe: " << countMax << endl;
```

erhält man als Ausgabe

```
  maximale Rekursionstiefe: 4
```

KAPITEL 9

Übung 9-1:

Gibt es einen Unterschied in der Initialisierung des Mitglieds `im` der Struktur `Complex` in den beiden folgenden Anweisungen?

```
Complex c5;
Complex c6 = { 0 };
```

Ja, hier gibt es einen Unterschied. `c5` bleibt uninitialisiert, d.h. die Mitglieder von `c5` erhalten undefinierte (zufällige) Werte. `c6` hingegen wird über eine Initialisiererliste initialisiert – auch wenn diese nicht vollständig ist. Alle nicht von der Initialisiererliste initialisierten Mitglieder erhalten den Wert `0`.

Übung 9-2:

Gibt es einen Unterschied zwischen den Anweisungen

```
Bruch b4 = b2;
```

und

```
Bruch b5; b5 = b2;
```

Nein, in der Praxis nicht.

Zumindest nicht in unserer derzeitigen Definition der Klasse `Bruch`:

```
struct Bruch
{
  int zaehler;
  int nenner;
};
```

In beiden Fällen erhalten die Mitglieder der linken Seite die Werte der Mitglieder der rechten Seite. Korrekterweise handelt es sich im ersten Fall um eine *Initialisierung*, im zweiten Fall um eine *Zuweisung*, die hier die gleiche Wirkung haben[5].

[5] und in der Praxis auch immer die gleichen Ergebnisse bringen sollen.

Übung 9-3:

Wo liegt der Unterschied zwischen den beiden folgenden Anweisungen:

```
Bruch b6 = { b2.zaehler };
```

und

```
Bruch b7 = ( b2.zaehler );
```

Die zweite Anweisung ist mit dem derzeitigen Stand der Klasse `Bruch` ungültig. Es handelt sich um einen Druckfehler im Buch – die Übung sollte später kommen, nämlich nach der Einführung der Konstruktoren. Dann handelt es sich bei der zweiten Anweisung um eine Initialisierung über Konstruktor, während bei der ersten Anweisung die Initialisierung über Initialisiererliste erfolgt.

Übung 9-4:

Schreiben Sie eine Funktion, die einen `Bruch` als Parameter erhält und als Ergebnis den Wert als `double` zurückliefert.

Analog zur Funktion `print` schreiben wir die Funktion wert als

```
double wert( Bruch b )
{
  return (double)b.zaehler / b.nenner;
}
```

Folgendes Beispiel zeigt den Aufruf:

```
  Bruch b = { 1, 3 };

  cout << "Der Wert ist: " << wert( b ) << endl;
```

Und als Ergebnis erhält man wie erwartet

```
  Der Wert ist: 0.333333
```

Vollständiges Beispiel:

```
#include <iostream>

using namespace std;

//----------------------------------------------------------------------
//       Bruch
//
struct Bruch
{
  int zaehler;
  int nenner;
};

//----------------------------------------------------------------------
//       wert
//
double wert( Bruch b )
{
  return (double)b.zaehler / b.nenner;
}

//----------------------------------------------------------------------
//       main
//
int main()
{

  Bruch b = { 1, 3 };

  cout << "Der Wert ist: " << wert( b ) << endl;

  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 9-5:

Machen Sie die Funktion „sicher“, indem Sie überlegen, was schief gehen kann und reagieren Sie entsprechend.

Die Funktion `wert` ist als

```
double wert( Bruch b )
{
  return (double)b.zaehler / b.nenner;
}
```

implementiert. Die einzige problematische Situation wäre der Wert `0` für den Nenner. Damit es nicht zu einer Division durch `0` kommt, prüfen wir diesen Fall explizit ab:

```
double wert( Bruch b )
{
  if ( b.nenner == 0 )
  {
    cout << "*** Nenner ist 0!" << endl;
    exit(1);
  }

  return (double)b.zaehler / b.nenner;
}
```

Übung 9-6:

Wie ist der Platzbedarf, wenn man die Anordnung der beiden Mitglieder wie im folgenden Beispiel umdreht?

```
struct S
{
  int   m2;    // zuerst int, dann short
  short m1;
};
```

Auch in dieser Anordnung werden normalerweise 8 Byte benötigt, wie man mit der folgenden Ausgabeanweisung einfach fest stellen kann:

```
  cout << sizeof S << endl;
```

Übung 9-7:

Braucht die Version mit jeweils eigenen Datentypen für DM, Dollar, Pfund etc. mehr Speicherplatz oder Laufzeit als eine Version, die direkt mit `int` arbeitet?

Nein, es werden keinerlei zusätzliche Ressourcen benötigt. Weder der Speicherbedarf noch die Laufzeit werden negativ verändert. Um so verwunderlicher ist es[6], dass man diese Technik in der Praxis eigentlich nie findet.

[6] Na ja, es kostet mehr Überlegung und mehr Schreibarbeit, und bringt zunächst nichts für die Funktionalität. Erfüllung der geforderten Funktionalität gemäß Lastenheft ist aber nun mal das, wofür bezahlt wird – für sonstige positive Eigenschaften gibt's kein Geld.

Übung 9-8:

Bestimmen Sie den Platzbedarf eines Objekts einer leeren Struktur für Ihren Compiler.

Folgendes Codesegment druckt diesen Platzbedarf aus:

```
struct S
{
};

...

cout << sizeof S << endl;
```

Für VC erhält man z.B. den Wert 1, d.h. ein Objekt einer leeren Struktur benötigt ein Byte.

Übung 9-9:

Schreiben Sie eine Funktion, die ein `short int` übernimmt, die beiden Bytes vertauscht und das Ergebnis zurückliefert. Verwenden Sie die Funktion, um die Wirkung des Vertauschens auf die Zahlen 0..1000 zu erkunden.

Mit Hilfe von `TwoChars` und `ShortChars` kann die Funktion als

```
short swapBytes( short value )
{

  struct TwoChars
  {
    char c1;
    char c2;
  };

  union ShortChars
  {
    short    shortWert;
    TwoChars charWert;
  };

  ShortChars sc;

  sc.shortWert = value;

  char c = sc.charWert.c1;
  sc.charWert.c1 = sc.charWert.c2;
  sc.charWert.c2 = c;

  return sc.shortWert;
}
```

notiert werden.

Die Ausgabe der Werte für 1 bis 999 erfolgt z.B. mit der Schleife

```
  for ( int i = 0; i < 1000; i++ )
  {
    cout << setw( 10 ) << i << setw( 10 ) << swapBytes( i ) << endl;
  }
```

Folgende Tabelle zeigt die ersten 50 Werte:

0	0	20	5120	40	10240
1	256	21	5376	41	10496
2	512	22	5632	42	10752
3	768	23	5888	43	11008
4	1024	24	6144	44	11264
5	1280	25	6400	45	11520
6	1536	26	6656	46	11776
7	1792	27	6912	47	12032
8	2048	28	7168	48	12288
9	2304	29	7424	49	12544
10	2560	30	7680	50	12800
11	2816	31	7936		
12	3072	32	8192		
13	3328	33	8448		
14	3584	34	8704		
15	3840	35	8960		
16	4096	36	9216		
17	4352	37	9472		
18	4608	38	9728		
19	4864	39	9984		

Vollständiges Programm:

```
#include <iostream>
#include <iomanip>

using namespace std;

//---------------------------------------------------------------------
//       swapBytes
//
short swapBytes( short value )
{

  struct TwoChars
  {
    char c1;
    char c2;
  };

  union ShortChars
  {
    short    shortWert;
    TwoChars charWert;
  };

  ShortChars sc;

  sc.shortWert = value;

  char c = sc.charWert.c1;
  sc.charWert.c1 = sc.charWert.c2;
  sc.charWert.c2 = c;

  return sc.shortWert;
}

//---------------------------------------------------------------------
//       main
//
int main()
{

  for ( int i = 0; i < 1000; i++ )
  {
    cout << setw( 10 ) << i << setw( 10 ) << swapBytes( i ) << endl;
  }


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 9-10:

Die Struktur `TwoChars` wird nur innerhalb von `ShortChars` verwendet. Formulieren Sie `ShortChars` mit Hilfe einer anonymen Struktur um die explizite Definition von `TwoChars` einzusparen.

An Stelle von

```
struct TwoChars
{
  char c1;
  char c2;
};

union ShortChars
{
  short    shortWert;
  TwoChars charWert;
};
```

schreibt man nun

```
union ShortChars
{
  short    shortWert;
  struct
  {
    char c1;
    char c2;
  } charWert;
};
```

Übung 9-11:

Schreiben Sie eine Funktion, die ein Objekt vom Typ `Variant` übernimmt und den Wert ausgibt.

Die Struktur `Variant` ist wie folgt definiert:

```
struct Variant
{
  int  typ; // 0: int, 1: double, 2: char

  union
  {
    int i;
    double d;
    char c;
  } wert;

};
```

Eine Funktion zum Ausdrucken muss an Hand des Typs auf den richtigen Teil der Union zu greifen:

```
void print( Variant var )
{
  switch ( var.typ )
  {
    case 0: cout << var.wert.i;  break;
    case 1: cout << var.wert.d;  break;
    case 2: cout << var.wert.c;  break;

    default:
      cout << "*** Fehler - ungültiger Typ!" << endl;
      exit(1);
  }

}
```

Nun kann man z.B.

```
  Variant v;

  //-- Speicherung eines integers
  //
  v.typ = 0;
  v.wert.i = 10;
  print( v );
  cout << endl;

  //-- Speicherung eines double
  //
  v.typ = 1;
  v.wert.d = 2.1415;
  print( v );
  cout << endl;
```

schreiben und erhält als Ergebnis

```
  10
  2.1415
```

Vollständiges Programm:

```
#include <iostream>

using namespace std;

//---------------------------------------------------------------------
//       Variant
//
struct Variant
{
  int  typ; // 0: int, 1: double, 2: char

  union
  {
    int i;
    double d;
    char c;
  } wert;

};

//---------------------------------------------------------------------
//       print
//
void print( Variant var )
{
  switch ( var.typ )
  {
    case 0: cout << var.wert.i;  break;
    case 1: cout << var.wert.d;  break;
    case 2: cout << var.wert.c;  break;

    default:
      cout << "*** Fehler - ungültiger Typ!" << endl;
      exit(1);
  }
}

//---------------------------------------------------------------------
//       main
//
int main()
{

  Variant v;

  //-- Speicherung eines integers
  //
  v.typ = 0;
  v.wert.i = 10;
  print( v );
  cout << endl;

  //-- Speicherung eines double
  //
  v.typ = 1;
  v.wert.d = 2.1415;
  print( v );
  cout << endl;


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```

Übung 9-12:

Schreiben Sie eine Funktion, die ein Objekt vom Typ `Variant` übernimmt und die Werte addiert, sofern dies sinnvoll ist (also nicht bei `char`). Die Funktion soll das Ergebnis wieder als `Variant` zurückliefern. Überlegen Sie, was man im Falle des Versuchs, eine Addition mit Zeichen durchzuführen, tun könnte.

Die Funktion muss die unterschiedlichen Kombinationsmöglichkeiten von Werten berücksichtigen. Dadurch entsteht die Notwendigkeit zu längeren `if-else`-Kaskaden bzw. `switch`-Anweisungen.

Wir betrachten eine Addition, an der ein Zeichen beteiligt ist, als nicht sinnvoll. Wir prüfen die Parameter vorher ab und beenden das Programm wenn die Addition von Zeichen versucht wird.

```
Variant add ( Variant lhs, Variant rhs )
{
  //-- Prüfung der Parameter auf Gültigkeit
  //
  switch ( lhs.typ )
  {
    case 0 : break;
    case 1 : break;

    case 2 :
      cout << "*** Fehler - linker Op ist char - Addition nicht möglich!" << endl;
      exit( 1 );

    default:
      cout << "*** Fehler - ungültiger Typ für linken Op! " << lhs.typ << endl;
      exit(1);
  }

  switch ( rhs.typ )
  {
    case 0 : break;
    case 1 : break;

    case 2 :
      cout << "*** Fehler - rechter Op ist char - Addition nicht möglich!" << endl;
      exit( 1 );

    default:
      cout << "*** Fehler - ungültiger Typ für rechten Op! " << rhs.typ << endl;
      exit(1);
  }
```

Die eigentliche Routine muss sich nun nur noch auf vier Fälle konzentrieren:

```
  //-- beide Parameter sind gültig.
  //   Die einzelnen Kombinationen ab arbeiten
  //

  Variant result;

  switch ( lhs.typ )
  {
    case 0: // lhs-Typ ist integer
      switch ( rhs.typ )
      {
        case 0 :
          //-- Zwei Integer sollen addiert werden. Ergebnis ist integer.
          //
          result.typ = 0;
          result.wert.i = lhs.wert.i + rhs.wert.i;
          break;

        case 1 :
          //-- Ein double soll zu einem integer addiert werden. Ergebnis ist double
          //
          result.typ = 1;
          result.wert.d = lhs.wert.i + rhs.wert.d;
          break;

      }
      break;

    case 1: // lhs-Typ ist double
      switch ( rhs.typ )
      {
        case 0 :
          //-- ein Integer soll zu einem double addiert werden. Ergebnis ist double.
          //
          result.typ = 1;
          result.wert.d = lhs.wert.d + rhs.wert.i;
          break;

        case 1 :
          //-- Zwei double sollen addiert werden. Ergebnis ist double
          //
          result.typ = 1;
          result.wert.d = lhs.wert.d + rhs.wert.d;
          break;

      }
      break;


  } // switch lhs.typ

  return result;
}
```

Da die Addition kommutativ ist (d.h. es gilt `a+b == b+a`), könnte man die vier Fälle noch auf zwei explizit zu implementierende Fälle zurück führen. Die Implementierung sei dem Leser als weitere Übung überlassen.

Mit folgenden Anweisungen kann man einen Test durchführen:

```
  Variant v1, v2;

  //-- Speicherung eines integers
  //
  v1.typ = 0;
  v1.wert.i = 10;

  //-- Speicherung eines double
  //
  v2.typ = 1;
  v2.wert.d = 2.1415;

  Variant result = add( v1, v2 );
  print( result );
  cout << endl;
```

Als Ergebnis erhält man den erwarteten `(double-)` Wert `12.1415`.

Vollständiges Programm:

```
#include <iostream>
#include <iomanip>

using namespace std;

//----------------------------------------------------------------------
//       Variant
//
struct Variant
{
  int  typ; // 0: int, 1: double, 2: char

  union
  {
    int i;
    double d;
    char c;
  } wert;

};

//----------------------------------------------------------------------
//       print
//
void print( Variant var )
{
  switch ( var.typ )
  {
    case 0: cout << var.wert.i;  break;
    case 1: cout << var.wert.d;  break;
    case 2: cout << var.wert.c;  break;

    default:
      cout << "*** Fehler - ungültiger Typ!" << endl;
      exit(1);
  }

}

//----------------------------------------------------------------------
//       add
//
Variant add ( Variant lhs, Variant rhs )
{
  //-- Prüfung der Parameter auf Gültigkeit
  //
  switch ( lhs.typ )
  {
    case 0 : break;
    case 1 : break;

    case 2 :
      cout << "*** Fehler - linker Op ist char - Addition nicht möglich!" << endl;
      exit( 1 );

    default:
      cout << "*** Fehler - ungültiger Typ für linken Op! " << lhs.typ << endl;
      exit(1);
  }

  switch ( rhs.typ )
  {
    case 0 : break;
    case 1 : break;

    case 2 :
      cout << "*** Fehler - rechter Op ist char - Addition nicht möglich!" << endl;
      exit( 1 );

    default:
      cout << "*** Fehler - ungültiger Typ für rechten Op! " << rhs.typ << endl;
      exit(1);
  }


  //-- beide Parameter sind gültig.
  //   Die einzelnen Kombinationen ab arbeiten
  //

  Variant result;
```

```
  switch ( lhs.typ )
  {
    case 0: // lhs-Typ ist integer
      switch ( rhs.typ )
      {
        case 0 :
          //-- Zwei Integer sollen addiert werden. Ergebnis ist integer.
          //
          result.typ = 0;
          result.wert.i = lhs.wert.i + rhs.wert.i;
          break;

        case 1 :
          //-- Ein double soll zu einem integer addiert werden. Ergebnis ist double
          //
          result.typ = 1;
          result.wert.d = lhs.wert.i + rhs.wert.d;
          break;

      }
      break;

    case 1: // lhs-Typ ist double
      switch ( rhs.typ )
      {
        case 0 :
          //-- ein Integer soll zu einem double addiert werden. Ergebnis ist double.
          //
          result.typ = 1;
          result.wert.d = lhs.wert.d + rhs.wert.i;
          break;

        case 1 :
          //-- Zwei double sollen addiert werden. Ergebnis ist double
          //
          result.typ = 1;
          result.wert.d = lhs.wert.d + rhs.wert.d;
          break;

      }
      break;


  } // switch lhs.typ

  return result;
}


//----------------------------------------------------------------------
//       main
//
int main()
{

  Variant v1, v2;

  //-- Speicherung eines integers
  //
  v1.typ = 0;
  v1.wert.i = 10;

  //-- Speicherung eines double
  //
  v2.typ = 1;
  v2.wert.d = 2.1415;

  Variant result = add( v1, v2 );
  print( result );
  cout << endl;


  //-- Anfordern einer Eingabe vom Benutzer, die mit ENTER
  //   abgeschlossen werden muss
  //
  char dummy;
  cin >> dummy;

  return 0;
}
```