2 Begriffliche Grundlagen der imperativen Programmierung

2.1 Programmierstile

2.2 Von Neumanns Rechnermodell

2.3 Variablenbegriff

2.4 Zuweisung

2.5 Fallunterscheidung

2.6 Schleifen
- Beispiel: Wurzel
- Schleifenformen
- Schleifeninvariante
- Beispiel: KGV
- Beispiel: Telefonbuchproblem
- Beispiel: Primzahlen

2.7 Prozedurbegriff
- Kapselung von Befehlsfolgen
- Arten der Parameterübergabe

2.8 Zusammenfassung

2.1 Programmierstile

Die Informatik nutzt Rechner, um die Lösung von Problemen in anderen Disziplinen zu unterstützen. Um ein Problem mittels eines Rechners zu lösen, benötigt man:

(i) eine Theorie des Wissensgebietes, aus dem das Problem stammt,
(ii) einen gebietsentsprechenden begrifflichen Apparat, im Idealfall einen Kalkül,
(iii) eine dem Problembereich angepaßte Notation, eine problemorientierte Programmiersprache,
(iv) einen Rechner samt Implementation der genutzten Programmiersprache.

Daher existieren eine Fülle spezieller Programmiersprachen, z. B. Sprachen zur künstliche Intelligenz, Sprachen für die kaufmännische Datenverarbeitung, Sprachen für Simulationen, Sprachen zur Beschreibung von Modellen, Sprachen zur Abfrage und Pflege von Datenbanken, Sprachen zur Prozeßsteuerung, Sprachen zur numerischen Steuerung von Schneidemaschinen, u.s.w.
In dem Buch "Programming Languages – History and Fundamentals" aus dem Jahre 1969 beschreibt Jean E. Sammet etwa 120 Programmiersprachen, von denen sie etwa 15 als allgemein und weitverbreitet ansieht. Seitdem sind zahlreiche neue Programmiersprachen entwickelt, implementiert und standardisiert worden. In dieser Vorlesung wird die universelle Programmiersprache C++ zur Beschreibung von Algorithmen genutzt.

Beispiele von Programmiersprachen:

Ada
Algol
APL
awk
Basic
C
C++
Cobol
Eiffel
Forth
Fortran
Haskell
Java
Javascript
Lisp
Logo
Mathematica
MatLab
Modula-2
Oberon
Pascal
Perl
PL/I
Postscript
Prolog
Python
Rexx
Scheme
SGML
Simscript
Simula
Sisal
Smalltalk
Snobol
SQL

Den vielen Problemgebieten entsprechen viele Programmierstile, u. a.:

- **funktionale Programmierung,**
- **logische Programmierung,**
- **deklarative Programmierung,**
- **objektorientierte Programmierung,**
- **imperative Programmierung.**

Eine klare Abgrenzung der einzelnen Programmierstile ist nicht möglich. Die imperative Programmierung entspricht am ehesten dem von Neumannschen Rechnermodell. Die herausragenden Eigenschaften der imperativen Programmierung sind:

- **ein partielles Gedächtnis in Form von Variablen,**
- **die Feinsteuerung von Befehlsfolgen.**

Ein Beispiel zum imperativen Programmierstil:

Beispiel: **Fibonacci-Zahlen von Leonardo Fibonacci aus Liber Abaci (1202 n.Chr.)**

Die ersten zwölf Fibonacci-Zahlen:

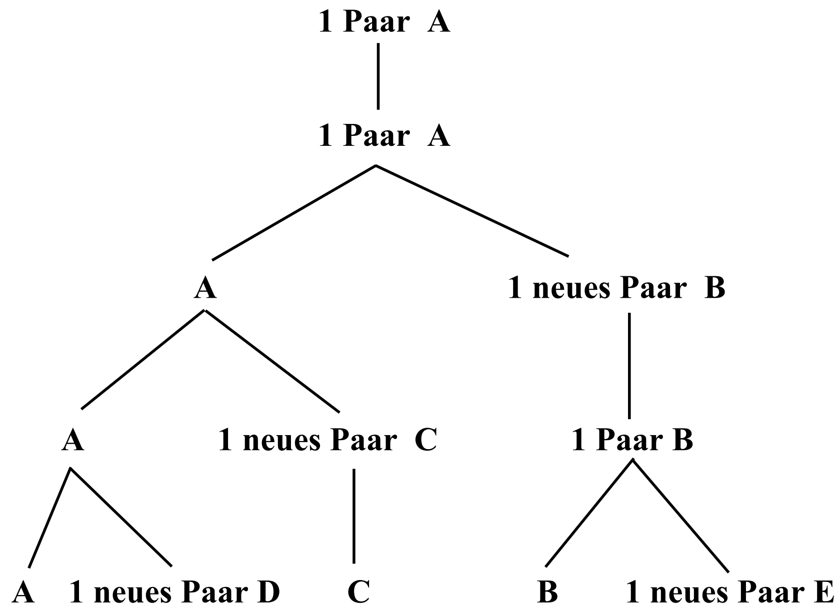
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, . . .

Bildungsgesetz: $f_1 = 1, \quad f_2 = 1$

$$f_n = f_{n-1} + f_{n-2} \quad (n > 2)$$

Bemerkung: **Fibonacci-Zahlen beschreiben exponentielle Wachstumsprozesse.**

Idealisiertes Wachstum einer Kaninchenpopulation:




```
//  Erzeugung von Fibonacci-Zahlen ohne Berücksich-
//  tigung der Beschränktheit von Computerzahlen

unsigned  Fibonacci  (unsigned n)  {
      if  (n == 0)
            return 0;
      else {  unsigned  f0 = 0;
              unsigned  f1 = 1;
              for (unsigned index = 2;  index < n+1;  ++index)
                  {  unsigned  fneu = f0 + f1;
                     f0 = f1;   f1 = fneu;  }
              return f1;  }
}//Fibonacci
```

Es existiert auch eine geschlossene Form für die Fibonacci–Zahlen:

Sei $\varphi = \sqrt{5}$,

seien $p = \dfrac{1+\varphi}{2}$ und $q = \dfrac{1-\varphi}{2}$,

dann gilt für die n-te Fibonacci–Zahl:

$$f(n) = \frac{p^n - q^n}{\varphi} \qquad \text{für } n = 0, 1, 2, 3, \ldots$$

Die Größen p und q sind die Wurzeln der quadratischen Gleichung $x^2 - x - 1 = 0$.

Ein Programmfragment zur Berechnung der n-ten Fibonacci–Zahl sähe dann so aus:

```
unsigned fibonacci (unsigned n) {
      double  p = (1.0 + sqrt (5.0)) / 2.0;
      double  q = (1.0 - sqrt (5.0)) / 2.0;
      double  b = 1.0;
      double  c = 1.0;
      for  (unsigned i = 0;   i < n;   ++i) {
            b  =  b*p;
            c  =  c*q;
      }
return unsigned ((b - c) / sqrt (5.0) + 0.5);
}//fibonacci
```

Fibonacci-Zahlen in Ganzzahlarithmetik berechnet.

66. Fibonacci-Zahl =	27.777.890.035.288
67. Fibonacci-Zahl =	44.945.570.212.853
68. Fibonacci-Zahl =	72.723.460.248.141
69. Fibonacci-Zahl =	117.669.030.460.994
70. Fibonacci-Zahl =	190.392.490.709.135
71. Fibonacci-Zahl =	308.061.521.170.129
72. Fibonacci-Zahl =	498.454.011.879.264
73. Fibonacci-Zahl =	806.515.533.049.393
74. Fibonacci-Zahl =	1.304.969.544.928.657
75. Fibonacci-Zahl =	2.111.485.077.978.050
76. Fibonacci-Zahl =	3.416.454.622.906.707
77. Fibonacci-Zahl =	5.527.939.700.884.757
78. Fibonacci-Zahl =	8.944.394.323.791.464
79. Fibonacci-Zahl =	14.472.334.024.676.221
80. Fibonacci-Zahl =	23.416.728.348.467.685
81. Fibonacci-Zahl =	37.889.062.373.143.906
82. Fibonacci-Zahl =	61.305.790.721.611.591
83. Fibonacci-Zahl =	99.194.853.094.755.497
84. Fibonacci-Zahl =	160.500.643.816.367.088
85. Fibonacci-Zahl =	259.695.496.911.122.585
86. Fibonacci-Zahl =	420.196.140.727.489.673
87. Fibonacci-Zahl =	679.891.637.638.612.258
88. Fibonacci-Zahl =	1.100.087.778.366.101.931
89. Fibonacci-Zahl =	1.779.979.416.004.714.189
90. Fibonacci-Zahl =	2.880.067.194.370.816.120
91. Fibonacci-Zahl =	4.660.046.610.375.530.309
92. Fibonacci-Zahl =	7.540.113.804.746.346.429
93. Fibonacci-Zahl =	12.200.160.415.121.876.738

Fibonacci-Zahlen in Gleitpunktarithmetik berechnet.

Genauigkeitsverlust ab 71. Fibonacci-Zahl

66. Fibonacci-Zahl =	27.777.890.035.288
67. Fibonacci-Zahl =	44.945.570.212.853
68. Fibonacci-Zahl =	72.723.460.248.141
69. Fibonacci-Zahl =	117.669.030.460.994
70. Fibonacci-Zahl =	190.392.490.709.135
71. Fibonacci-Zahl =	308.061.521.170.130
72. Fibonacci-Zahl =	498.454.011.879.265
73. Fibonacci-Zahl =	806.515.533.049.395
74. Fibonacci-Zahl =	1.304.969.544.928.660
75. Fibonacci-Zahl =	2.111.485.077.978.055
76. Fibonacci-Zahl =	3.416.454.622.906.715
77. Fibonacci-Zahl =	5.527.939.700.884.770
78. Fibonacci-Zahl =	8.944.394.323.791.484
79. Fibonacci-Zahl =	14.472.334.024.676.254
80. Fibonacci-Zahl =	23.416.728.348.467.736
81. Fibonacci-Zahl =	37.889.062.373.143.992
82. Fibonacci-Zahl =	61.305.790.721.611.736
83. Fibonacci-Zahl =	99.194.853.094.755.728
84. Fibonacci-Zahl =	160.500.643.816.367.456
85. Fibonacci-Zahl =	259.695.496.911.123.200
86. Fibonacci-Zahl =	420.196.140.727.490.688
87. Fibonacci-Zahl =	679.891.637.638.613.888
88. Fibonacci-Zahl =	1.100.087.778.366.104.576
89. Fibonacci-Zahl =	1.779.979.416.004.718.592
90. Fibonacci-Zahl =	2.880.067.194.370.823.680
91. Fibonacci-Zahl =	4.660.046.610.375.542.784
92. Fibonacci-Zahl =	7.540.113.804.746.366.976
93. Fibonacci-Zahl =	9.223.372.036.854.775.808

Wandlungsfehler bei 93. Fibonacci-Zahl ???

2.2 Von Neumanns Rechnermodell:

Speicher:

Ausführungseinheit:

Programmfragment:

Lade	703
Addiere	708
Multipliziere	701
Speichere	712
Teste	
Springe	. . .

Bemerkung: Die imperative Programmierung stellt die erste Abstraktionsebene über dem von Neumann Modell dar.

2.3 Variablenbegriff

Eine Variable ist gekennzeichnet durch drei Angaben:

Referenz,
Behälter,
Wert.

Graphische Veranschaulichung:

mille	Referenz
1000	Behälter

Der Inhalt des Behälters ist der Wert, in diesem Fall die Zahl 1000.

Bemerkungen:

(i) Der Wert einer Variablen kann geändert werden. Wertfolgen für die Variablen f0 f1 fneu:

f0	f1	fneu
0	1	1
1	1	2
1	2	3
2	3	5

(ii) Das Fassungsvermögen eines Behälters ist begrenzt. Eine Nichtberücksichtigung dieses Sachverhalts führt manchmal zu kaum entdeckbaren Fehlern, da der im Behälter aufbewahrte Teilwert in keinem algorithmischen Zusammenhang mit dem aufzubewahrenden Gesamtwert steht.

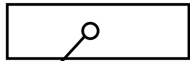
Veranschaulichung:

tau

45678932145 67

(iii) Referenzen können Werte von Variablen sein.

drei — Urreferenz

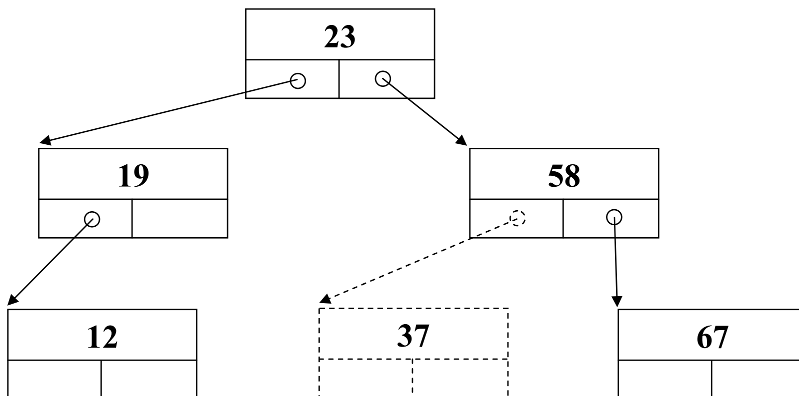


Zwischenreferenz als Wert

Wert 3

(iv) Die Verwendung von Referenzen als Werte führt zu dynamischen Datenstrukturen.

Beispiel: Sortierbaum



Baumelemente können dynamisch hinzugefügt werden.

(v) Es ist üblich, Variablen einen Typ zuzuordnen; der Typ schränkt den Verwendungszweck von Variablen ein.

Beispiele für Einschränkungen sind:

Eine Variable kann nur Ganzzahlwerte,
Gleitpunktwerte,
Logische Formeln,
Prozeduren mit drei Ganzzahlparametern,
geometrische Objekte,
u.s.w. annehmen.

(vi) Für Variable gilt in Programmiersprachen oft ein Deklarationszwang.

```
int   mille            =  1000
double const pi   =  3.141592653589793
```

Der Zusatz const soll eine spätere Veränderung des Wertes von pi verhindern.

(vii) Es müssen nicht immer alle drei Attribute einer Variablen vorhanden sein. Man kennt benennungslose Variablen – Variablen ohne Referenz – als Temporärvariablen und Variablen, denen noch kein Wert zugewiesen wurde. Die Entdeckung nichtinitialisierter Variablen ist oft problematisch.

(viii) Eine Variable kann mehrere Referenzen besitzen.

2.4 Zuweisung

Form:

Variablenreferenz := Ausdruck

Beispiele:

alpha := 8
beta := 30.6 * (1 + Zinssatz)

Austausch zweier Werte:
hilf := a
a := b
b := hilf

Bemerkungen:

- **(i)** **Bei der Zuweisung achtet man auf die Typverträglichkeit von Ausdruck und Variable.**
- **(ii)** **Die Bildung von Ausdrücken kann komplizierten Gesetzen unterliegen, ein Beispiel:**
  m := if x < b then x else b fi
  In diesem Fall sollten die Ausdrücke x und b mit der Variablen m typverträglich sein.
- **(iii)** **Auch die Variablenreferenz kann das Resultat einer aufwendigen Berechnung sein.**
- **(iv)** **In C++ verwendet man als Zuweisungsoperator das Zeichen '='.**

2.5 Fallunterscheidung:

Form:

Falls ***Bedingung***
Dann
Anweisungsfolge 1
Sonst
Anweisungsfolge 2
Ende

In C++ kennt man die Formen:

- **(i)** **if (*condition*) *statement***
- **(ii)** **if (*condition*) *statement* else *statement***
- **(iii)** **switch (*expression*) *statement***

Bemerkung: **In C++ kann an die Stelle einer Anweisung eine Anweisungsfolge treten, falls die Folge in geschweifte Klammern gefaßt wird.**

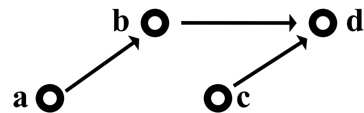
Für die switch-Anweisung gelten eine Reihe von Einschränkungen, u. a. sollte *expression* ein Ganzzahlergebnis liefern, in die switch-Anweisung sollte nicht von außen gesprungen werden, die switch-Marken sollten bis auf default konstante Ganzzahlausdrücke sein.

Beispiel:

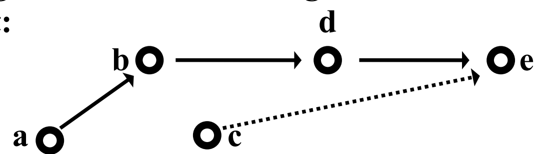
```
switch (expression) {
        case 7:  /*   fall through   */
        case 0:  /*      */
        default:  /*      */
}//switch
```

Beispiel: Sortieren von 5 Zahlen a, b, c, d, e durch 7 Vergleiche, dies ist die Minimalzahl.

Bild: Seien $a \leq b$, $c \leq d$ und $b \leq d$:



Einfügen von e durch 2 Vergleiche in die Kette $a \leq b \leq d$.
Damit:



Durch 2 weitere Vergleiche wird c in die Kette $a \leq b \leq d \leq e$ eingefügt.

Programmfragment zum Sortieren von 5 Zahlen durch 7 Vergleiche:

```
if  (a > b) {
        int h = a;  a = b;  b = h;
}

assert  (a <= b);

if  (c > d) {
        int h = c;  c = d;  d = h;
}

assert  (c <= d);

if  (b > d) {
        int h = b;  b = d;  d = h;
            h = a;  a = c;  c = h;
}

assert (a <= b && b <= d && c <= d);

if  (b > e)
        if (a > e) {
                int  h = e;  e = d;
                    d = b;  b = a;  a = h;
        } else {
                int h = e;  e = d;  d = b;  b = h;
        }
else if (d > e) {
        int h = e;  e = d;  d = h;
}

assert (a <= b && b <= d && d <= e && c <= e);

if  (c > b) {
        if  (c > d) {
                int h = c;  c = d;  d = h;
        }
} else if (c > a) {
        int  h = b;  b = c;  c = h;
} else {
        int  h = c;  c = b;  b = a;  a = h;
}

assert (a <= b && b <= c && c <= d && d <= e);
```

```
//  Nutzung von assert
#include <cassert>
#include <iostream>
using namespace std;

int main () {


   int a = 27, b = 10, c = 38,  d = 2,  e =15;

   if  (a > b) {
            int h = a;  a = b;  b = h;
      }

      assert  (a <= b);

      if  (c > d) {
            int h = c;  c = d;  d = h;
      }

      assert  (c <= d);


      if  (b > d) {
            int h = b;  b = d;  d = h;
               h = a;  a = c;  c = h;
      }

      assert (a <= b && b <= d && c <= d);

      if  (b > e)
            if (a > e) {
                  int  h = e;  e = d;
                       d = b;  b = a;  a = h;
            } else {
                  int h = e;  e = d;  d = c;  c = h;
            }
      else if (d > e) {
            int h = e;  e = d;  d = h;
      }

/*42*/      assert (a <= b && b <= d && d <= e && c <= e);

      if  (c > b) {
            if  (c > d) {
                  int h = c;  c = d;  d = h;
            }
      } else if (c > a) {
            int  h = b;  b = c;  c = h;
      } else {
            int  h = c;  c = b;  b = a;  a = h;
      }

      assert (a <= b && b <= c && c <= d && d <= e);

      cout << a << ' ' << b << ' ' << c << ' ' << d
            << ' ' << e << endl;

}//main

/*
Ein Testlauf:

Assertion failed: a <= b && b <= d && d <= e && c <= e,
file E: tassert.cpp, line 42
*/
```

2.6 Schleifen

Berechnung der Quadratwurzel:
Näherungsformel zur Nullstellenberechnung:

$$f(x + h) \approx f(x) + h * f'(x)$$

$$\Rightarrow \quad f(x) + h * f'(x) = 0$$

$$\Rightarrow \quad h = -\frac{f(x)}{f'(x)} \qquad \text{, falls } f'(x) \neq 0 \text{ ,}$$

$$\Rightarrow \quad \text{neue Näherung: } x_{neu} = x + h = x - \frac{f(x)}{f'(x)} \text{ .}$$

Für die Quadratfunktion $f(x) = x^2 - a$ erhält man

$$x_{neu} = x - \frac{x^2 - a}{2 * x} = \frac{x}{2} + \frac{a}{2 * x} = \frac{1}{2} * (x + \frac{a}{x})$$

Damit Linearprogramm:

x_0 := Startwert

$x_1 := (x_0 + a/x_0) / 2$

$x_2 := (x_1 + a/x_1) / 2$

•
•
•

$x_{n+1} := (x_n + a/x_n) / 2$

Fragen: (i) Welchen Startwert soll man wählen?
(ii) Wie viele Schritte soll man durchführen?

Bekannt ist: Bei einem guten Startwert sind drei bis vier Verbesserungsschritte ausreichend.

Man kennt drei Schleifenarten:

(i) die abweisende Schleife:

F1: **Solange *Bedingung* erfüllt**
Wiederhole
Anweisungsfolge
Ende

F2: **Wiederhole**
Anweisungsfolge
Solange *Bedingung* erfüllt

Bemerkung: Bei normaler Beendigung gilt *Bedingung* nicht.

(ii) die zielorientierte Schleife:

F3: **Bis *Bedingung* erfüllt**
Wiederhole
Anweisungsfolge
Ende

F4: **Wiederhole**
Anweisungsfolge
Bis *Bedingung* erfüllt

Bemerkung: Bei normaler Beendigung gilt *Bedingung*.

(iii) die zählende Schleife:

F5: **Für** ***Laufvariable*** **von** ***Startwert*** **bis** ***Endwert***
Wiederhole
Anweisungsfolge
Ende

F6: **Für** ***Laufvariable*** **von** ***Startwert*** **bis** ***Endwert***
mit ***Schrittweite***
Wiederhole
Anweisungsfolge
Ende

Die Schleife gibt einem die Möglichkeit, über Programmeinheiten sinnvolle Aussagen zu treffen.

z.B.

```
while  B
      Schleifenkörper
while_end
```

Unmittelbar nach Schleifenende gilt die Aussage: Nicht B

Diese Eigenschaft kann man zur Konstruktion von Programmen nutzen;
Vorgehensweise:

1. **Formulierung des Ziels: Z**
2. **Bildung der Schleifenbedingung: nicht Z**
3. **Nachliefern der Details.**

Schleifenformen in der Sprache C++:

Formen der abweisenden Schleife:

(i)

```
while  (expression)
      statement;
assert (!expression)
```

(ii)

```
do
      statement;
while (expression);
assert (!expression)
```

Form der allgemeinen Schleife:

```
for  (init-statement;  expression-1;  expression-2)
      statement;
assert (!expression-1)
```

Diese Form läßt sich im wesentlichen zurückführen auf die abweisende Form:

```
init-statement;
while  (expression-1) {
      statement;
      expression-2;
}
```

Bemerkungen:

(i) Die Sprache C++ bietet die Möglichkeit, über eine break- oder goto-Anweisung jederzeit eine Schleife zu verlassen. Hiervon sollte man fast nie Gebrauch machen, denn dadurch invalidiert man wahrscheinlich die Aussagen über den Programmzustand, die sich aus dem Steuerausdruck herleiten lassen.

(ii) Eine Endlos-Schleife läßt sich einfach bilden:

```
for  (   ;   ;   )
      statement;
```

(iii) Der nächste Standard für C++ wird auch eine bereichsbasierte Form der for-Schleife enthalten.

Ein C++-Beispiel:

```
//  Beispiel:  Summe der ersten n Quadratzahlen.

int const n = ...;    //  Setzen der Reihenlänge

//  Form F1:
        int  qsum  = 0;
        int  z     = n;
        while  (z > 0) {
               qsum = qsum + z * z;
               --z;
        }
```

```
//  Form F2:
        int qsum  = 0;
        int z     = n;
        do  {
               qsum = qsum + z * z;
               --z;
         }  while (z != 0);
```

```
//  Form F3:
        int qsum = 0;
        for  (int z = 1;   z <= n;   ++z) {
               qsum = qsum + z * z;
         }
```

Bemerkung: **Es existieren noch weitere Möglichkeiten das obige Programm zu formulieren. Falls man die implizite Randbedingung − n ist eine positive Ganzzahl − einhält, ist es weitgehend eine Frage des persönlichen Geschmacks, welche Form man wählt. Im Falle einer Verletzung der impliziten Randbedingung erhält man unterschiedliche Verhaltensweisen.**

```
//  Beispiel:  Berechnung des kleinsten gemeinsamen
//             Vielfachen dreier natürlicher Zahlen  a,  b,  c.

      int a =  36;       //  Zahl  1
      int b =  30;       //  Zahl  2
      int c =  49;       //  Zahl  3

//  Definition des KGV:
//  KGV (a, b, c) = Minimum { i ∈ ℕ   mit i mod a = 0
//                     ∧ i mod b = 0 ∧ i mod c = 0);

      int    i = 1;     //   i außerhalb Schleife definiert
      while  (i % a != 0 || i % b != 0 || i % c != 0)
            ++i;
     //  Wert (i) == KGV (a, b, c)

//  Bemerkung: Obige Berechnung erscheint zu aufwendig.

//  Eine zweite Art der Berechnung:

      int  A = a;      int  B = b;      int  C = c;

      //  Ziel:  A == B ∧ B == C ∧ A == C
      while  (A != B || B != C) {
            if  (A < B)
                  A = A + a;
            else if (B < C)
                  B = B + b;
            else
                  C = C + c;
      }
      assert (A == B && B == C);
```

Telefonbuchproblem:

Welcher Nachname ist der häufigste im Telefonbuch?

Bekannt: Telefonbuch ist nach Nachname sortiert.

Erläuterung zum Telefonbuchproblem: Als Stellvertreter für Namen wählen wir ganze Zahlen.

Folge x: 1, 2, 7, 7, 7, 12, 12, 12, 14, 17, 22, 22

Kompression zur Häufigkeitstabelle:

1:	1
2:	1
7:	3
12:	3
14:	1
17:	1
22:	2

Auswahl einer Maximalhäufigkeit: 3

Antwort 12 ist ein häufigstes Element.

Bemerkung: Problemlösung erscheint ressourcenintensitiv.

Eine formelmäßige Problemformulierung führt direkt zu einer besseren Lösung.

Folge x: 1, 2, 7, 7, 7, 12, 12, 12, 14, 17, 22, 22
Zeigerpaar: ⇑ ⇑

Damit Problemreduktion auf :
Sei x ein sortierter Array der Länge n;
gesucht ist ein maximales k mit
x[i] = x[i+1] = . . . = x[i+k-1]

Programm:

```
array  [1 .. n]  x     //  x ist sortiert
i  := 1
k := 0
while   i <= n         //  Mindestlänge ist 1
repeat
      if  x [i]  =  x [i-k]  then
            in :=  i
            k  :=  k + 1
      fi
      i  := i + 1
while_end
//  Das Element x [in] ist ein häufigstes,
//  es tritt k-mal in x auf.
```

```
//  Algorithmus formuliert als C++-Programm.


#include <iostream>
using namespace std;

//  C-Arrays sollte man in C++ möglichst nicht benutzen,
//  besser ist Nutzung der Datenstruktur vector.
int a [] = {1, 1, 1, 2, 7, 7, 7, 12, 12, 12, 12, 14, 17, 22, 22, 22,
              22, 25, 30, 31, 31, 31, 31, 31, 37, 50};
//  Anzahl der Elemente des Arrays a
int n  =  sizeof (a) / sizeof (int);

int main () {
   int  i  =  0;
   int  k  =  0;
   int in  =  0;  // Position eines häufigsten Elementes
   while  (i < n) {
      if  (a [i] == a [i-k]) {
           in  =  i;
           ++k;
      }
      ++i;
   }

   cout << "Ein häufigstes Element ist " << a [in] << endl;
}//main


/*  Ein Lauf:

Ein häufigstes Element ist 31

*/
```

// **Berechnung der ersten n Primzahlen:**

```
const int n = ...;              //  Parameter der Rechnung

int primzahlen [n+1];       //  Reservierung Speicherplatz

      int  j               = 1;       //  Primzahlkandidat
      primzahlen [1]  = 2;       //  kleinste Primzahl

      for  (int k = 2;   k <= n;  ++k) {
             //  primzahlen [1] .. primzahlen [k-1] sind
             //  die k-1 kleinsten Primzahlen
             bool  prim = false;    // Attribut von j
             while  (!prim) {
                    j = j + 2;
                    prim = true;    //  Annahme: j ist prim
                                         //  Überprüfung der Annahme
                    for  (int i = 2;
                          (i < k) && prim &&
                            (primzahlen [i] * primzahlen [i] <= j);
                          ++i)
                          prim = (j % primzahlen [i] != 0);
             }
             //  nächste Primzahl gefunden
             primzahlen [k] = j;
      }
      //  primzahlen [1] ... primzahlen [n] sind die
      //  ersten n Primzahlen;
```

Bemerkung: Die innere Schleife läßt sich vereinfachen.

2.7 Prozedurbegriff

Eine einfache Verschlüsselungsmethode ist Cäsars Verschlüsselung.

Beispiel:
Urtext: `Dies ist eine geheime Nachricht.`
Vortransformation: `DIESISTEINEGEHEIMENACHRICHT`
Caesar-Chiffrat: `GLHVLVWHLQHJHKHLPHQDFKULFKW`

Um Nachrichten zu verschlüsseln und zu entschlüsseln, wird man zwei Prozeduren

encode und decode

zur Verfügung stellen mit der Eigenschaft:

decode (encode (Nachricht)) = Nachricht.

Technisch gesehen läßt sich aus jedem Codefragment eine Prozedur bilden.

Beispiel-Fragment:

```
sum := 0;
für  z  von  1 bis  n
wiederhole
        sum := sum + t
 Ende
 r := sum
```

Als Parameter betrachten wir die Größen z, n, t, r und bezeichnen das parametrisierte Codefragment mit

Allgemeine-Summe (z, n, t, r).

Die Prozedur Allgemeine-Summe kann man nun vielfältig nutzen, z. B. zur Berechnung eines Produkts 12*5,3:

Allgemeine-Summe (—, 12, (5,3), Ergebnis)

Nach Ausführung der Prozedur erhält man:

Ergebnis = 63,6.

Die Vorgehensweise besteht im textuellen Ersetzen der Parameter durch interessierende Größen, somit steht

Allgemeine-Summe (—, 12, (5,3), Ergebnis) für

```
sum := 0;
für  z  von  1 bis  12
wiederhole
        sum := sum + 5,3
 Ende
 Ergebnis := sum
```

Wählt man eine andere Parameterbelegung, z. B.

Allgemeine Summe (i, m–k+1, a [k–1+i], Erg),

dann berechnet man $\text{Erg} = \sum_{i=k}^{m} a_i$ **;**

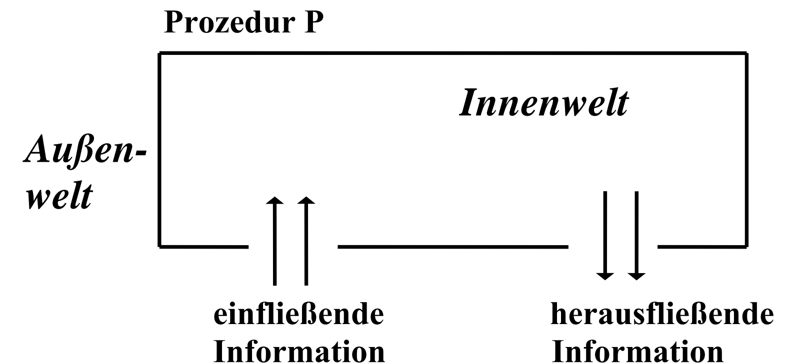
dies entspricht:

```
sum := 0;
für  i  von  1  bis  m–k+1
wiederhole
        sum := sum + a [k–1+i]
 Ende
 Erg := sum
```

Dies einfache Beispiel zeigt schon, daß man bei der Wahl der Parameter sehr sorgfältig vorgehen muß, sonst erfordert die Nutzung von Prozeduren einen zu hohen Vorbereitungsaufwand.

Bemerkungen:

(i) Eine Prozedur stellt eine mit einem Namen versehene parametrisierte Folge von Befehlen dar. Die Namensgebung schafft eine gedankliche Trennung zwischen zwei Welten, der Prozedur-Innenwelt und der Prozedur-Außenwelt. Die Beziehung zwischen beiden Welten ist kompliziert, so stellt sich die Frage, welche Kenntnis der Außenwelt in der Innenwelt vorliegen soll und welche Kenntnis der Innenwelt in der Außenwelt?



(ii) Der direkt sichtbare Informationsfluß erfolgt über explizite Parameter. Nach Flußrichtung unterscheidet man zwischen Eingabeparameter, Ausgabeparameter und transiente Parameter.

(iii) Falls man Parameter als Variable mit ihren drei Komponenten Behälter, Wert und Referenz ansieht, stellt sich die Frage, welche Rechte die Prozedur an ihren Parametern hat. So kann man z. B. das Recht, den Wert eines Parameters zu ändern, einschränken.

(iv) Man kennt viele Arten der Parameterübergabe, die wichtigsten sind:

- Wertübergabe
- Referenzübergabe
- Ergebnisübergabe
- Wert-Ergebnis-Übergabe
- Prozedurübergabe
- Ausdrucksübergabe ("call by name")
- Textuelle Substitution mit Bezeichneridentifizierung

Wertübergabe: Bei der Wertübergabe wird ein Parameter wie eine lokale Variable betrachtet, die bei Prozeduraufruf initialisiert wird. Die Prozedur kann jederzeit den Wert ändern, es fließt keine Information über einen Wertparameter in die Außenwelt.

Referenzübergabe: Bei einem Referenzparameter stellt der Aufrufer die Parametervariable zur Verfügung. Die gerufene Prozedur erhält nur die Referenz auf die Variable, Wertänderungen werden ohne Verzögerung an der Urvariablen vollzogen, die Urvariable muß nicht initialisiert sein.

Ergebnisübergabe: Die Ergebnisübergabe ist das Gegenstück zur Wertübergabe. Der Aufrufer stellt eine Variable zur Verfügung, die bei Prozedurabschluß einen neuen Wert erhält.

Wert-Ergebnis-Übergabe: Bei dieser Art der Parameterübergabe stellen Rufer und gerufene Prozedur eine Variable zur Verfügung. Die Prozedurvariable wird bei Prozeduraufruf initialisiert, beim Verlassen der Prozedur wird ihr momentaner Wert in die Variable des Rufers kopiert.

Prozedurübergabe: An eine Prozedur kann man auch eine vollständige Rechenvorschrift übergeben. Dies erfolgt in der Regel in Form einer Prozedur. Die schwierigen Fragen über die Behandlung der Parameter des Parameters wollen wir an dieser Stelle nicht betrachten.

Ausdrucksübergabe: Hier wird ein Parameter durch eine parameterlose Routine repräsentiert. Für jeden Aktualparameter wird eine Routine geschaffen, die innerhalb der Prozedur an den Parameterverwendungsorten aufgerufen wird. In Algol 60 Implementationen nennt diese Routine einen "thunk".

Textuelle Substitution: Hierbei wird ein Prozeduraufruf durch den Text des Prozedurkörpers ersetzt, wobei natürlich die Formalparameter durch die Aktualparameter substituiert werden. Diese Art der Parameterübergabe kann man als die natürliche ansehen, sie wird aber kaum realisiert, da der Mensch einer langen Substitutionskette fast nie folgen kann.

Beispiel zur Parameterübergabe:

Deklarationen:

```
        variable   i
        variable   j
        procedure  p  (x, y)
            variable   j      //  innere Variable
            i  :=  2
            x  :=  x  +  3
            j  :=  20
            print  (y)        //  Wertübergabe
        end of procedure
```

Anweisungen:

```
        i  :=  1
        j  :=  10
        p (i,  i*j)
        print (i)             //  Wertübergabe
```

Bemerkung: Obiges Beispiel zeigt, daß unterschiedliche Methoden der Parameterübergabe verschiedene Ergebnisse liefern.

Liste möglicher Ergebnisse:

Wertübergabe:	y = 10	i = 2
Referenzübergabe:	y = 10	i = 5
Ergebnisübergabe:	nicht sinnvoll	
Wert-Ergebnis-Übergabe:	y = 10	i = 4
Ausdrucksübergabe:	y = 50	i = 5
Textersetzung:	y = 100	i = 5

Teilweise Nachbildung des Beispiels als C++ - Programme:

```
/*  Beispiel zur Parameterübergabe:  Wertübergabe  */

#include <iostream>
using namespace std;

// globale Variable
int  i;
int  j;

void p (int x, int y) {
      int  j;
      i  =  2;
      x  =  x  +  3;
      j  =  20;
      cout  <<  "    y  =  "  <<  y  <<endl;
}

int main () {
      i  =  1;
      j  =  10;
      cout  <<  "Wertübergabe:"  <<  endl;
      p (i,  i*j);
      cout  <<  "    i  =  "  <<  i  <<  endl  <<  endl;
}//main

/*  Ausgabe:
                        Wertübergabe:
                            y  = 10
                            i  =   2
*/
```

```
/*  Beispiel zur Parameterübergabe:  Referenzübergabe  */

#include <iostream>
using namespace std;

// globale Variable
int  i;
int  j;

void p (int& x,  int const& y) {
      int  j;
      i  =  2;
      x  =  x  +  3;
      j  =  20;
      cout  <<  "    y  =  "  <<  y  <<  endl;
}


int main () {
      i  =  1;
      j  =  10;
      cout  <<  "Referenzübergabe:"  <<  endl;
      p (i,  i*j);
      cout  <<  "    i  =  "  << i  <<  endl  <<  endl;

}//main

/*  Ausgabe:
                         Referenzübergabe:
                             y  =  10
                             i  =    5
*/
```

```
/* Beispiel zur Parameterübergabe: Ausdrucksübergabe */

#include <iostream>
using namespace std;

// globale Variable
int  i,  j;

//  Form mit Referenzen
typedef  int&  ithunk ();
int&  f1 ()  {return  i;}
int   NEU_;    int&  f2 () {return   (NEU_  =  i  *  j);}

void p (ithunk  x,  ithunk  y) {
      int  j;
      i  =  2;
      x()  =  x()  +  3;
      j  =  20;
      cout  <<  "    y  =  "  <<  y()  <<  endl;
}

int main () {
      i  =  1;
      j  =  10;
      cout  <<  "Ausdrucksübergabe:"  <<  endl;
      p (f1,  f2);
      cout  <<  "    i  =  "  <<  i  <<  endl  <<  endl;
}//main

/*  Ausgabe:
                   Ausdrucksübergabe:
                       y  =  50
                       i  =    5
*/
```

```
/*    Beispiel zur Parameterübergabe:
      Wert-Ergebnis-Übergabe
*/



#include <iostream>
using namespace std;

//  globale Variable
int  i;
int  j;


//  Hilfsdefinitionen zur Nachbildung
//  der Wert-Ergebnis-Parameterübergabe,
//  Nutzung von Zeigern
typedef  int*  (*ithunk) ();
typedef  int*  address;

int*  g1 ()  { return  &i;}
int   X_Y_W;
int*  g2 ()  {return  &(X_Y_W  =  i  *  j);}

void p (ithunk  x,  ithunk  y) {
      //  lokale Variable für x und y
      int         ix,  iy;
      address  aix,  aiy;

      //  Übernahme in lokale Variable
      aix  =  x();
      aiy  =  y();

      ix  =  *aix;
      iy  =  *aiy;

      //  Prozedurkörper
      int  j;
      i  =  2;
      ix  =  ix  +  3;
      j  =  20;
      cout  <<  "     y  =  "  <<  iy  <<  endl;

      /*  Rückgabe der Werte  */
      *aix = ix;
      *aiy = iy;
}



int main () {

      i  =  1;
      j  =  10;

      cout  <<  "Wert-Ergebnis-Übergabe:"  <<  endl;
      p (g1,  g2);
      cout << "     i =  "  <<  i  <<  endl  <<  endl;

}//main

/*  Ausgabe:

                  Wert-Ergebnis-Übergabe:
                      y  =  10
                      i  =    4
*/
```

2.8 Zusammenfassung:

- **Das Rechnermodell der imperativen Programmierung ist das von Neumannsche Rechnermodell.**
- **Ein imperatives Programm ist eine Folge von Befehlen. Man kennt drei einfache Kompositionsmuster für die Befehle: die Sequenz, die Fallunterscheidung und die Schleife.**
- **Eine sinnvolle Folge von Befehlen faßt man unter einem Bezeichner zusammen und parametrisiert sie zwecks allgemeiner Verwendbarkeit.**
- **In einem Programm kann man den Programmzustand vor und nach Ausführung eines Befehls durch logische Formeln über Inhalte von Variablen des Programms beschreiben.**
- **Innerhalb einzelner Prozeduren ist die Schleifeninvariante die wichtigste Teilbeschreibung eines Programmzustands; die Schleifeninvariante sollte man formulieren, bevor man eine Schleife programmiert.**