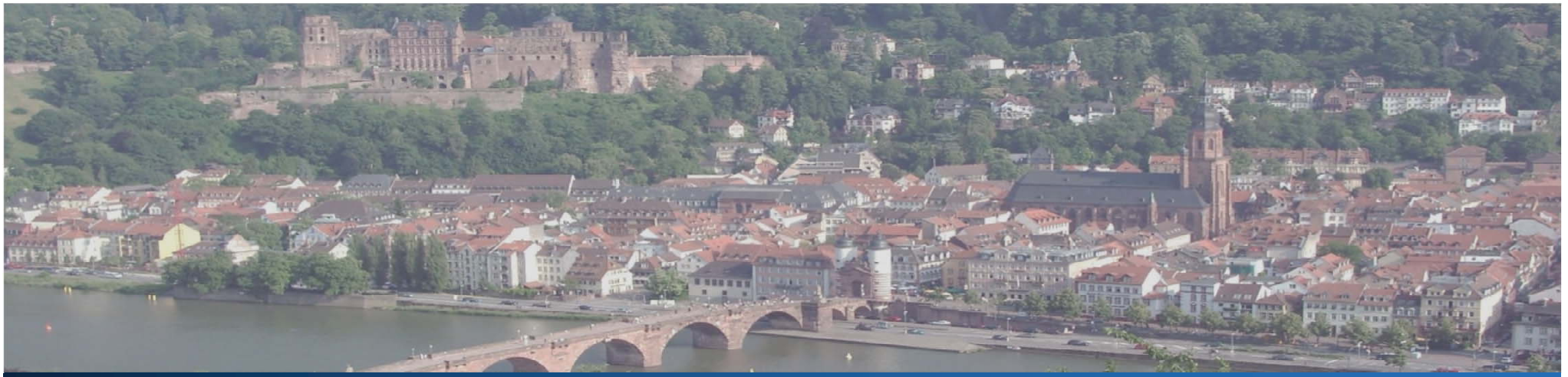
Einführung in die Praktische Informatik

Prof. Björn Ommer **HCI, IWR**

Computer Vision Group



Globale Vars & Umgebungsmodell

- Globale Variablen
- Das Umgebungsmodell
- Stapel
- Monte-Carlo Methode zur Bestimmung von π

Globale Variablen und das Umgebungsmodell

Globale Variablen

Bisher: Funktionen haben kein Gedächtnis! Ruft man eine Funktion zwei den selben Argumenten auf, so liefert sie auch dasselbe Ergebnis.

Grund:

- Berechnung einer Funktion hängt nur von ihren Parametern ab.
- Die lokale Umgebung bleibt zwischen Funktionsaufrufen nicht erhalten.

Das werden wir jetzt ändern!

Beispiel: Konto

Ein Konto kann man einrichten (mit einem Anfangskapital versehen), man kann abheben (mit negativem Betrag auch einzahlen), und man kann den Kontostand abfragen.

Programm: **(Konto [konto1.cc])**

```
#include "fcpp.hh"

int konto; // die GLOBALE Variable

void einrichten( int betrag )
{
  konto = betrag;
}
```

```
int kontostand()
{
  return konto;
}

int abheben( int betrag )
{
  konto = konto - betrag;
  return konto;
}

int main()
{
  einrichten( 100 );
  print( abheben( 25 ) );
  print( abheben( 25 ) );
  print( abheben( 25 ) );
}
```

Bemerkung:

- Die Variable `konto` ist außerhalb jeder Funktion definiert.
- Die Variable `konto` wird zu Beginn des Programmes erzeugt und nie mehr zerstört.
- Alle Funktionen können auf die Variable `konto` zugreifen. Man nennt sie daher eine globale Variable.
- Die Funktionen `einrichten`, `kontostand` und `abheben` stellen die **Schnittstelle** zur Bearbeitung des Kontos dar.

Frage: Oben haben wir eingeführt, dass Ausdrücke relativ zu einer Umgebung ausgeführt werden. In welcher Umgebung liegt `konto`?

Das Umgebungsmodell

Die Auswertung von Funktionen und Ausdrücken mit Hilfe von Umgebungen nennt man *Umgebungsmodell* (im Gegensatz zum Substitutionsmodell).

Definition: **(Umgebung)**

- Eine Umgebung enthält eine Bindungstabelle, d. h. eine Zuordnung von Namen zu Werten.
- Es kann beliebig viele Umgebungen geben. Umgebungen werden während des Programmlaufes implizit (automatisch) oder explizit (bewusst) erzeugt bzw. zerstört.
- Die Menge der Umgebungen bildet eine Baumstruktur. Die Wurzel dieses Baumes heißt globale Umgebung.

Das Umgebungsmodell

- Zu jedem Zeitpunkt des Programmablaufes gibt es eine aktuelle Umgebung. Die Auswertung von Ausdrücken erfolgt relativ zur aktuellen Umgebung.

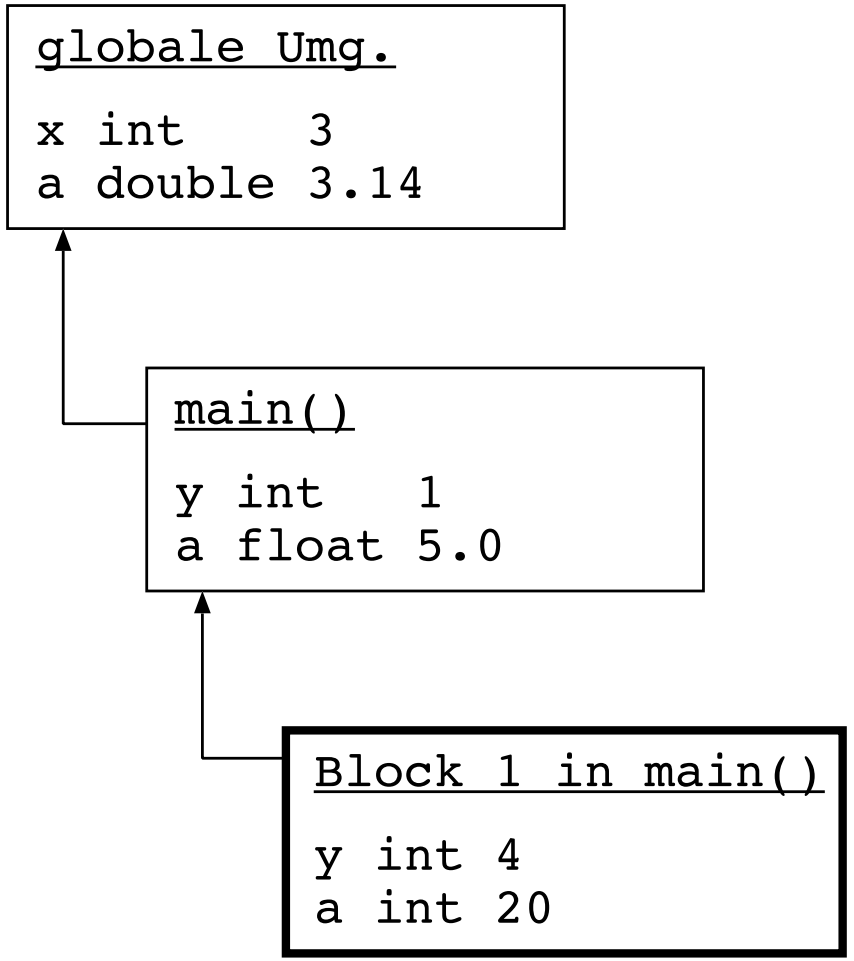
- Die Auswertung relativ zur aktuellen Umgebung versucht den Wert eines Namens in dieser Umgebung zu ermitteln, schlägt dies fehl, wird rekursiv in der nächst höheren („umschließenden“) Umgebung gesucht.

Eine Umgebung ist also relativ kompliziert. Das Umgebungsmodell beschreibt, wann Umgebungen erzeugt/zerstört werden und wann die Umgebung gewechselt wird.

Beispiel:

```
int x = 3;
double a = 4.3;          // 1
int main()
{
   int y = 1;
   float a = 5.0;        // 2
   {
      int y = 4;
      int a = 8;         // 3
      a = 5 * y;         // 4
      ::a = 3.14;        // 5
   }
}                        // 6
```

Nach Marke 5:



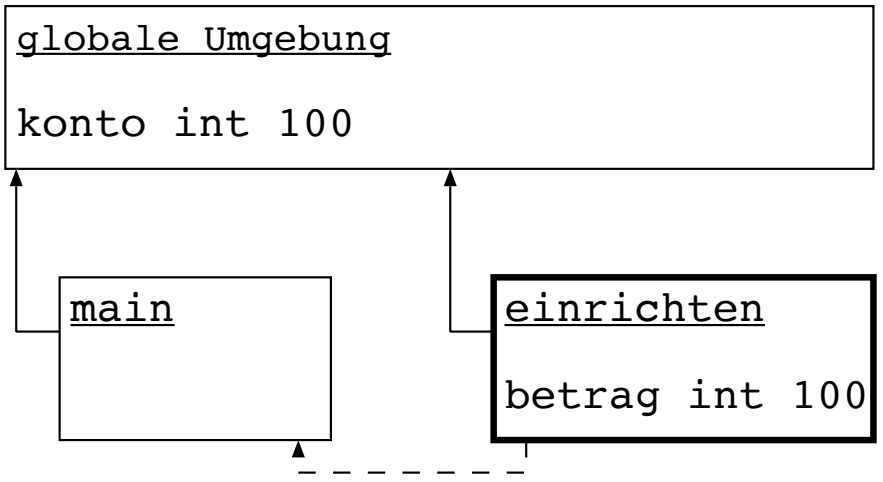
Eigenschaften:

- In einer Umgebung kann ein Name nur höchstens einmal vorkommen. In verschiedenen Umgebungen kann ein Name mehrmals vorkommen.
- Kommt auf dem Pfad von der aktuellen Umgebung zur Wurzel ein Name mehrmals vor, so verdeckt das erste Vorkommen die weiteren.
- Eine Zuweisung wirkt immer auf den sichtbaren Namen. Mit vorangestelltem `::` erreicht man die Namen der globalen Umgebung.
- Eine Anweisungsfolge in geschweiften Klammern bildet einen sogenannten Block, der eine eigene Umgebung besitzt.
- Eine Schleife `for ( int i=0; ...` wird in einer eigenen Umgebung ausgeführt. Diese Variable `i` gibt es im Rest der Funktion nicht.

Beispiel: (Funktionsaufruf)

```
int konto;

void einrichten( int betrag )
{
  konto = betrag;      // 2
}

int main()
{
  einrichten( 100 ); // 1
}
```

Nach Marke 2:



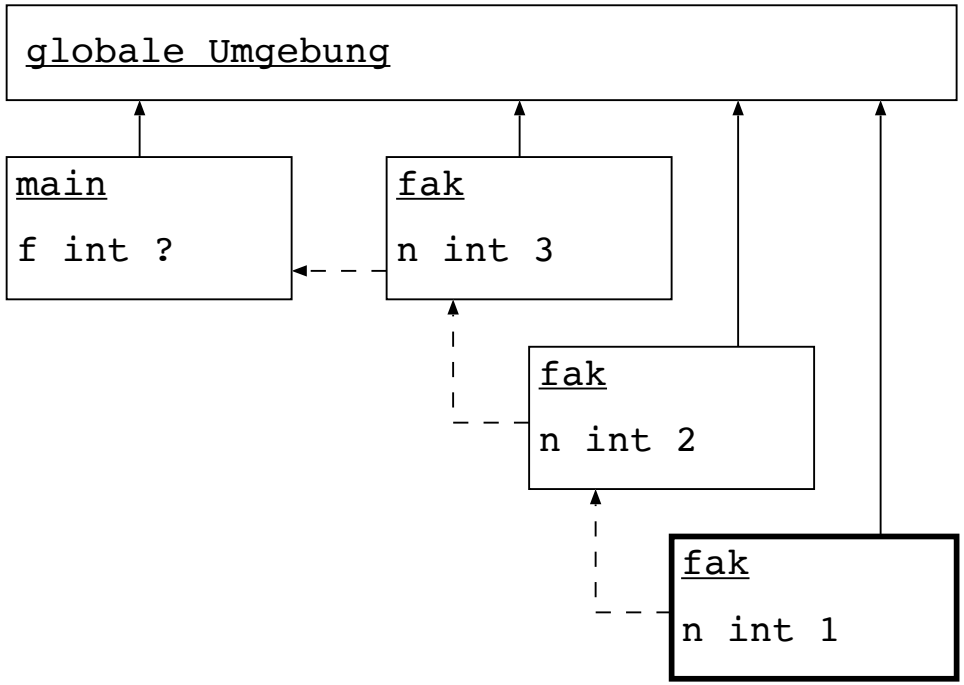
Bemerkung:

- Jeder Funktionsaufruf startet eine neue Umgebung unterhalb der globalen Umgebung. Dies ist dann die aktuelle Umgebung.
- Am Ende einer Funktion wird ihre Umgebung vernichtet und die aktuelle Umgebung wird die, in der der Aufruf stattfand (gestrichelte Linie).
- Formale Parameter sind ganz normale Variablen, die mit dem Wert des aktuellen Parameters initialisiert sind.
- Sichtbarkeit von Namen ist in C++ am Programmtext abzulesen (statisch) und somit zur Übersetzungszeit bekannt. Sichtbar sind: Namen im aktuellen Block, nicht verdeckte Namen in umschließenden Blöcken und Namen in der globalen Umgebung.

Beispiel: (Rekursiver Aufruf)

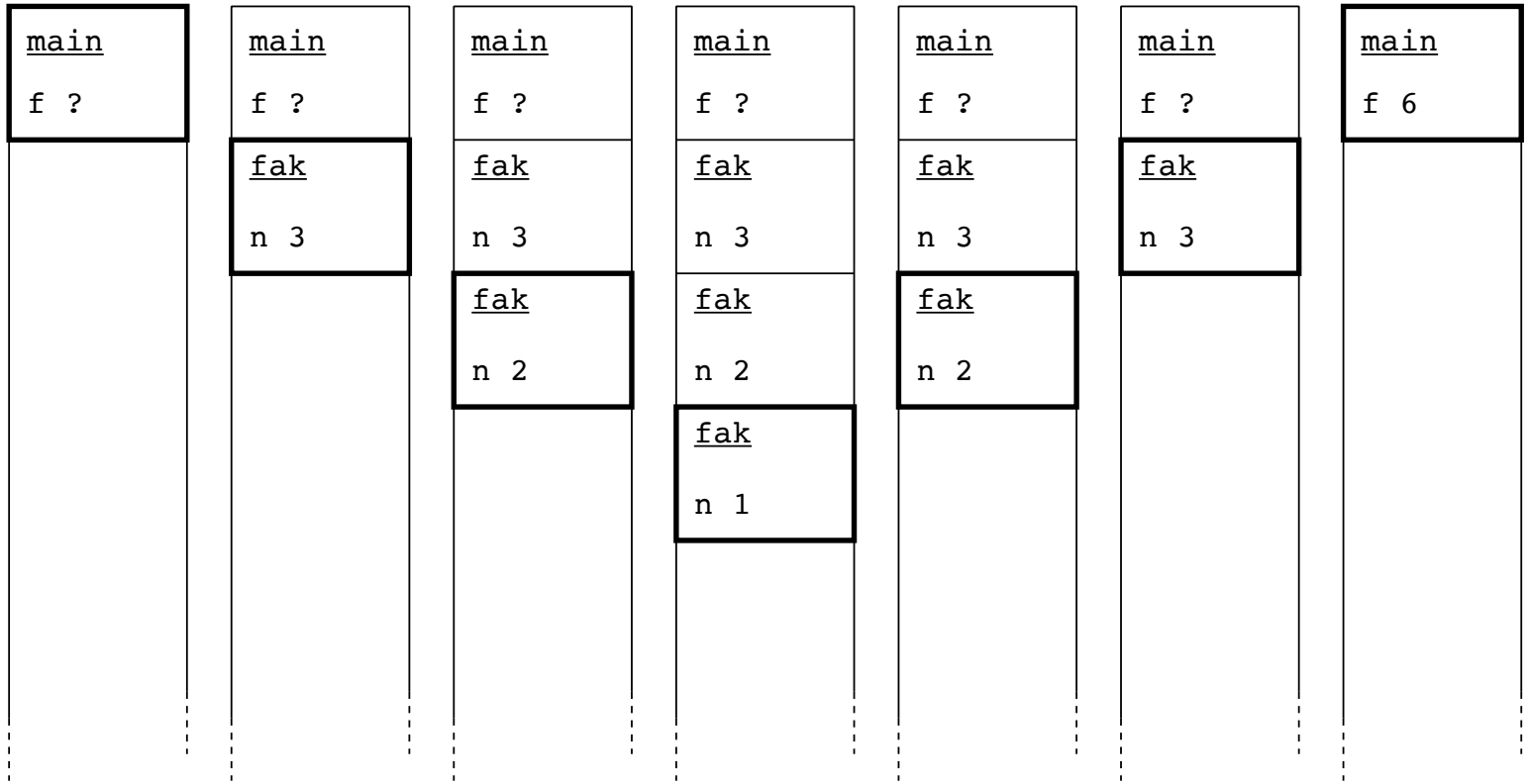
```
int fak( int n )
{
  if ( n == 1 )
    return 1;
  else
    return n * fak( n-1 );
}

int main()
{
  int f = fak( 3 ); // 1
}
```

Während Auswertung von `fak(3)`:



Bemerkung: Im obigen Beispiel gibt es zusätzlich noch eine „versteckte“ Variable für den Rückgabewert einer Funktion. Return kann als **Zuweisung über Umgebungsgrenzen** hinweg verstanden werden.

Beispiel: Die Berechnung von `fak(3)` führt zu:



Globale Vars & Umgebungsmodell

- Globale Variablen
- Das Umgebungsmodell
- Stapel
- Monte-Carlo Methode zur Bestimmung von π

Stapel

Bezeichnung: Die Datenstruktur, mit der das Umgebungsmodell normalerweise implementiert wird, nennt man Stapel, Stack oder LIFO (last in, first out). Im Deutschen ist, historisch bedingt, auch die Bezeichnung Keller gebräuchlich.

Definition: Ein Stapel ist eine Datenstruktur, welche folgende Operationen zur Verfügung stellt:

- Erzeugen eines leeren Stapels (*create*)
- Ablegen eines neuen Elements auf dem Stapel (*push*)
- Test, ob Stapel leer (*empty*)
- Holen des zuletzt abgelegten Elements vom Stapel (*pop*)

Programm: (Stapel [stack1.cc])

```
#include "fcpp.hh"

typedef int element_type;    // Integer-Stack

// START stack-library ...

const int stack_size = 1000;

element_type stack[stack_size];
int top = 0;  // Stapelzeiger (zeigt auf naechstes freies Element)

// Stack-Operationen

void push( element_type e )
{
  stack[top] = e;
  top = top + 1;
```

```
}

bool empty()
{
  return top == 0;
}

element_type pop()
{
  top = top - 1;
  return stack[top];
}

int main()
{
  push( 4 );
  push( 5 );
  while ( !empty() )
    print( pop() );
  return 0;
}
```

Bemerkung: Die Stapel-Struktur kann man verwenden, um rekursive in nicht rekursive Programme zu transformieren (wen es interessiert, findet unten eine nichtrekursive Variante für das Wechselgeld Beispiel). Dies ist aber normalerweise nicht von Vorteil, da der für Rekursion verwendete Stapel höchstwahrscheinlich effizienter verwaltet wird.

Programm: **(Wechselgeld nichtrekursiv [wg−stack.cc])**

```
#include "fcpp.hh"

int nennwert( int nr ) // Muenzart -> Muenzwert
{
  if ( nr == 1 ) return 1;  if ( nr == 2 ) return 2;
  if ( nr == 3 ) return 5;  if ( nr == 4 ) return 10;
  if ( nr == 5 ) return 50;
  return 0;
}
```

```
struct Arg                  // Stapelelemente
{
  int betrag;               // das sind die Argumente der
  int muenzarten;           // rekursiven Variante
};

const int N = 1000;         // Stapelgroesse

int wechselgeld2( int betrag )
{
  Arg stapel[N];            // hier ist der Stapel
  int sp = 0;               // der "stack pointer"
  int anzahl = 0;           // das Ergebnis
  int b, m;                 // Hilfsvariablen in Schleife

  stapel[sp].betrag = betrag;   // initialisiere St.
  stapel[sp].muenzarten = 5;    // Startwert
```

```
sp = sp + 1;                          // ein Element mehr

while ( sp > 0 )                      // Solange Stapel nicht leer
{
  sp = sp - 1;                        // lese oberstes Element
  b = stapel[sp].betrag;              // lese Argumente
  m = stapel[sp].muenzarten;

  if ( b == 0 )
    anzahl = anzahl + 1;              // Moeglichkeit gefunden
  else if ( b>0 && m>0 )
  {
    if ( sp >= N )
    {
      print( "Stapel zu klein" );
      return anzahl;
    }
    stapel[sp].betrag = b;                // Betrag b
    stapel[sp].muenzarten = m - 1; // mit m - 1 Muenzarten
```

```
        sp = sp + 1;

        if ( sp >= N )
        {
          print( "Stapel zu klein" );
          return anzahl;
        }
        stapel[sp].betrag = b - nennwert(m);
        stapel[sp].muenzarten = m;      // mit m Muenzarten
        sp = sp + 1;
      }
    }

    return anzahl; // Stapel ist jetzt leer
}

int main()
{
    print( wechselgeld2( 300 ) );
}
```

Wiederholung

Zusammengesetzte Datentypen: **struct**, **union**

Erweiterung des Umgebungsmodels um globale Umgebung.

Globale Umgebung ist Wurzel einer Baumstruktur.

Funktionsaufruf startet neue Umgebung unterhalb der globalen Umgebung.

$\{...\}$ startet eine neue Umgebung unterhalb der aktuellen Umgebung.

Namen werden von der aktuellen Umgebung zur Wurzel hin gesucht. Globale Variable sind daher immer sichtbar, es sei denn, der Name ist verdeckt.

Umgebungen werden mittels eines Stapels gespeichert.

Umwandlung rekursiver in nichtrekursive Programme

Globale Vars & Umgebungsmodell

- Globale Variablen
- Das Umgebungsmodell
- Stapel
- Monte-Carlo Methode zur Bestimmung von π

Monte-Carlo Methode zur Bestimmung von π

Folgender Satz soll zur (näherungsweisen) Bestimmung von π herangezogen werden (randomisierter Algorithmus):

Satz: *Die Wahrscheinlichkeit q, dass zwei Zahlen $u, v \in \mathbb{N}$ keinen gemeinsamen Teiler haben, ist $\frac{6}{\pi^2}$. Zu dieser Aussage siehe [Knuth, Vol. 2, Theorem D].*

Um π zu approximieren, gehen wir daher wie folgt vor:

- Führe N „Experimente“ durch:
  - Ziehe „zufällig“ zwei Zahlen $1 \leq u_i, v_i \leq n$.
  - Berechne ggT(u_i, v_i).
  - Setze

$$e_i = \begin{cases} 1 & \text{falls ggT}(u_i, v_i) = 1 \\ 0 & \text{sonst} \end{cases}$$

- Berechne relative Häufigkeit $p(N) = \frac{\sum_{i=1}^{N} e_i}{N}$. Nach obigem Satz erwarten wir

$$\lim_{N \to \infty} p(N) = \frac{6}{\pi^2}.$$

- Also gilt $\pi \approx \sqrt{6/p(N)}$ für große N.

Pseudo-Zufallszahlen

Um Zufallszahlen zu erhalten, könnte man physikalische Phänomene heranziehen, von denen man überzeugt ist, dass sie „zufällig“ ablaufen (z. B. radioaktiver Zerfall). Solche Zufallszahl-Generatoren gibt es tatsächlich, sie sind allerdings recht teuer.

Daher begnügt man sich stattdessen oft mit Zahlenfolgen $x_k \in \mathbb{N}$, $0 \leq x_k < n$, welche deterministisch sind, aber zufällig „aussehen“. Für die „Zufälligkeit“ gibt es verschiedene Kriterien. Beispielsweise sollte jede Zahl gleich oft vorkommen, wenn man die Folge genügend lang macht:

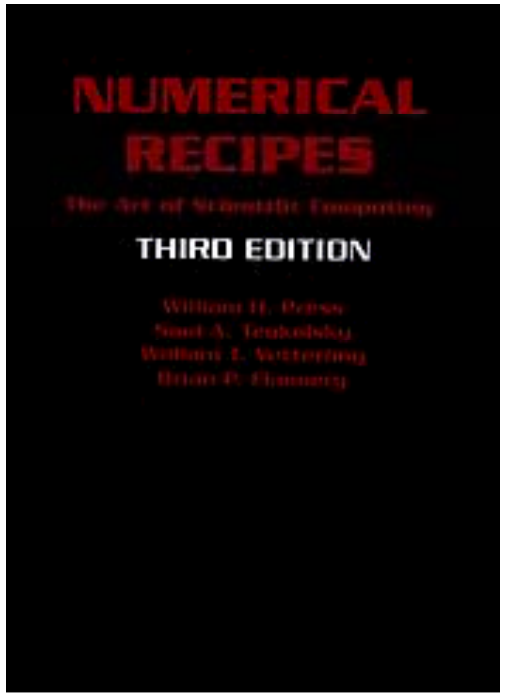
$$\lim_{m\to\infty} \frac{|\{i|1 \leq i \leq m \wedge x_i = k\}|}{m} = \frac{1}{n}, \quad \forall k = 0, \ldots, n-1.$$

Einfachste Methode: (Linear Congruential Method) Ausgehend von einem x_0 verlangt man für $x_1, x_2, \ldots$ die Iterationsvorschrift

$$x_{n+1} = (ax_n + c) \bmod m.$$

Damit die Folge zufällig aussieht, müssen $a, c, m \in \mathbb{N}$ gewisse Bedingungen erfüllen, die man in [Knuth, Vol. 2, Kapitel 3] nachlesen kann.

Programm: (π mit Monte Carlo Methode [montecarlo1.cc])




```
#include "fcpp.hh"

int x = 93267;

unsigned int zufall()
{
  const int ia = 16807,  im = 2147483647;
  const int iq = 127773, ir = 2836;
  int k;

  k = x / iq;                          // LCG xneu = (a * xalt) mod m
  x = ia * ( x - k*iq ) - ir*k;        // a = 7^5, m = 2^31-1
  if ( x < 0 ) x = x + im;             // keine lange Arithmetik
  return x;                            // s. Numerical Recipes
}                                      // in C, Kap. 7.
```

```
unsigned int ggT( unsigned int a, unsigned int b )
{
  if ( b == 0 ) return a;
  else          return ggT( b, a % b );
}

int experiment()
{
  unsigned int x1, x2;

  x1 = zufall(); x2 = zufall();
  if ( ggT(x1, x2) == 1 )
    return 1;
  else
    return 0;
}
```

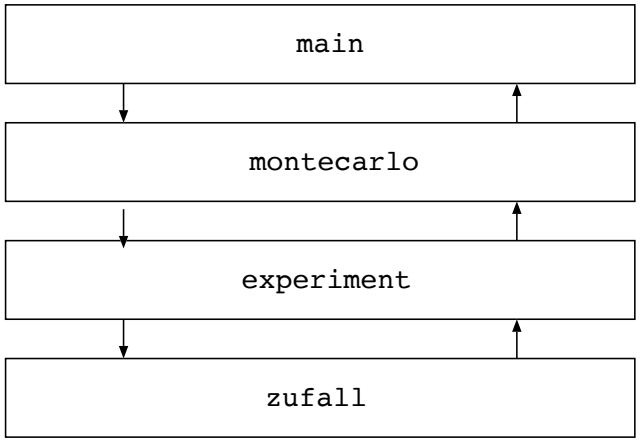
```
double montecarlo( int N )
{
  int erfolgreich = 0;
  for ( int i=0; i<N; i=i+1 )
    erfolgreich = erfolgreich + experiment();

  return ((double) erfolgreich) / ((double) N);
}

int main( int argc, char *argv[] )
{
  print( sqrt( 6.0/montecarlo( readarg_int( argc, argv, 1 ) ) ) )
}
```

Monte-Carlo funktional

Die Funktion `zufall` widerspricht offenbar dem funktionalen Paradigma (sonst müsste sie ja immer denselben Wert zurückliefern!). Stattdessen hat sie „Gedächtnis“ durch die globale Variable `x`.

Frage: Wie würde eine funktionale(re) Version des Programms ohne globale Variable aussehen?

Antwort: Eine Möglichkeit wäre es, `zufall` den Parameter x zu übergeben, woraus dann ein neuer Wert berechnet wird. Dieser Parameter müsste aber von `main` aus durch alle Funktionen hindurchgetunnelt werden:



Für `experiment`→`montecarlo` ist obendrein die Verwendung eines zusammengesetzten Datentyps als Rückgabewert nötig.

Beobachtung: In dieser Situation entstünde durch Beharren auf einem funktionalen Stil zwar kein Effizienzproblem, die Struktur des Programms würde aber deutlich komplizierter.