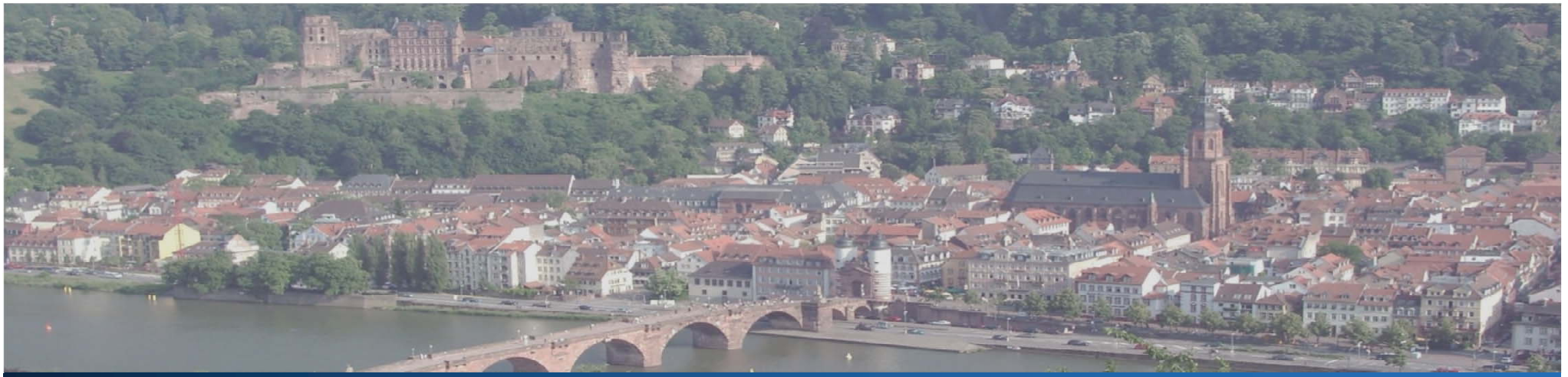
Einführung in die Praktische Informatik

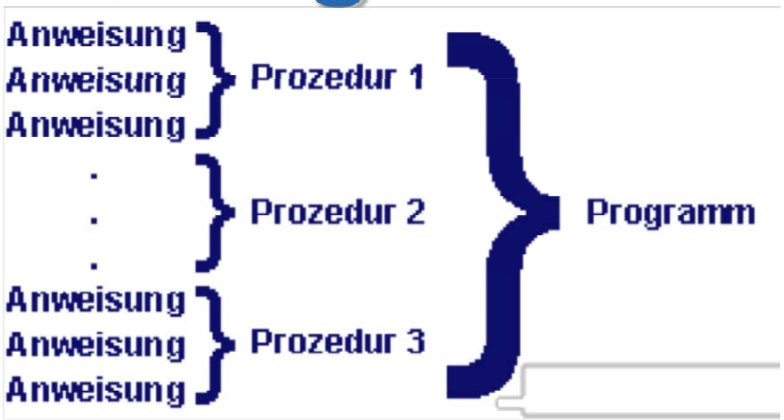
Prof. Björn Ommer **HCI, IWR**

Computer Vision Group

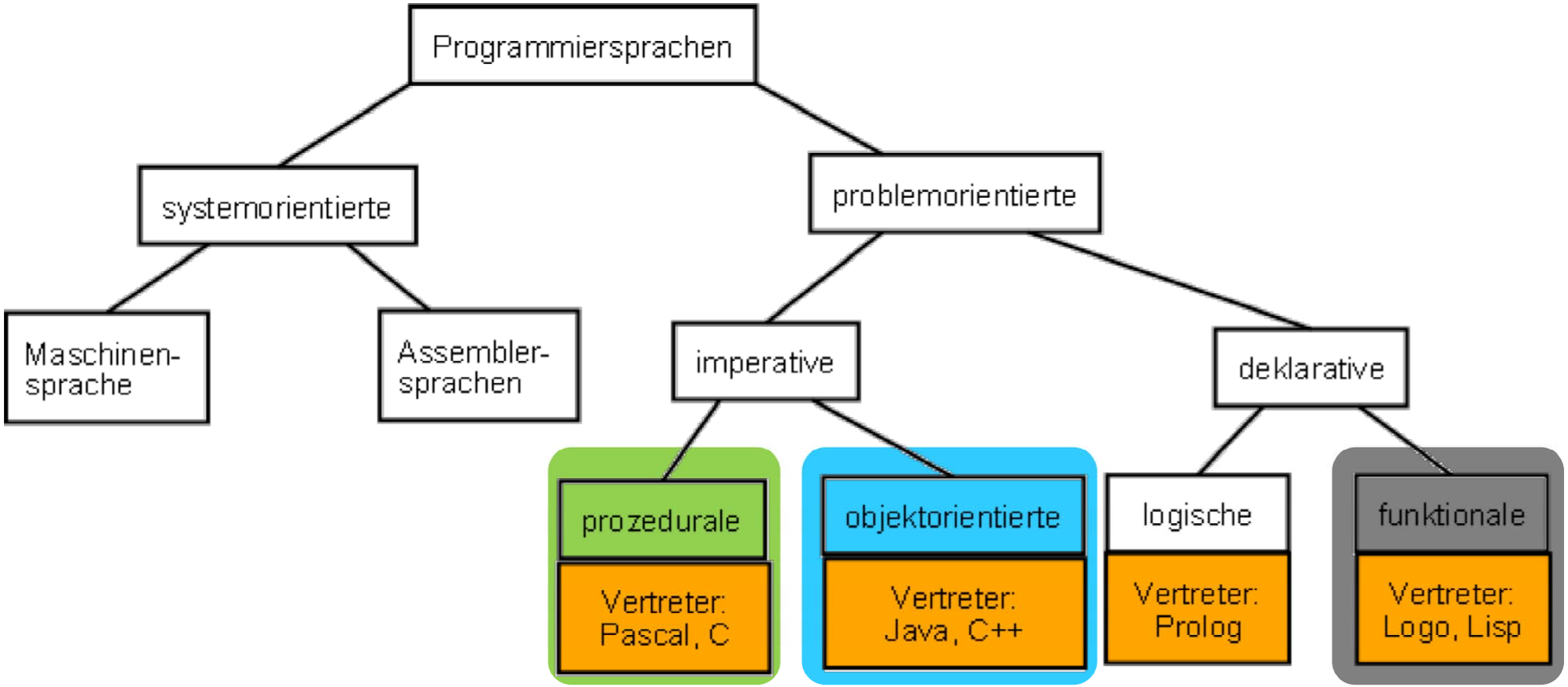


Prozedurale Programmierung

- Lokale Variablen und die Zuweisung
- Syntax von Variablen-definition und Zuweisung
- Anweisungsfolgen (Sequenzen)
- Bedingte Anweisung (Selektion)
- Schleifen
- Formale Programmverifikation
- Prozeduren und Funktionen



Programmiersprachen und -paradigmen



Prozedurale Programmierung

Bisher besteht unser Berechnungsmodell aus folgenden Bestandteilen:

1. Auswertung von zusammengesetzten Ausdrücken aus Zahlen, Funktionsaufrufen und Selektionen.

2. Konstruktion neuer Funktionen.

Namen treten dabei in genau zwei Zusammenhängen auf:

1. Als Symbole für Funktionen.

2. Als formale Parameter in Funktionen.

Im Substitutionsmodell werden bei der Auswertung einer Funktion die formalen Parameter im Rumpf durch die aktuellen Werte ersetzt und dann der Rumpf ausgewertet.

Prozedurale Programmierung (Forts.)

Unter Vernachlässigung von Resourcenbeschränkungen (endlich große Zahlen, endlich viele rekursive Funktionsauswertungen) kann man zeigen, dass dieses Berechnungsmodell äquivalent zur Turingmaschine ist.

In der Praxis erweist es sich als nützlich, weitere Konstruktionselemente einzuführen um einfachere, übersichtlichere und wartbarere Programme schreiben zu können.

Der Preis dafür ist, dass wir uns von unserem einfachen Substitutionsmodell verabschieden müssen!

Lokale Variablen und die Zuweisung

Konstanten

In C++ kann man konstante Werte wie folgt mit Namen versehen:

Beispiel:

```
float umfang( float r )
{
  const double pi = 3.14159265;
  return 2*r*pi;
}

int hochacht( int x )
{
  const int x2 = x*x;          // jetzt gibt es ein x2
  const int x4 = x2*x2;        // nun ein x4
  return x4*x4;
}
```

Bemerkung:

- Wir können uns vorstellen, dass einem Namen ein Wert zugeordnet wird.
- Einem formalen Parameter wird der Wert bei Auswertung der Funktion zugeordnet.
- Einer Konstanten kann nur einmal ein Wert zugeordnet werden.
- Die Auswertung solcher Funktionsrümpfe erfordert eine Erweiterung des Substitutionsmodells:
  - Ersetze formale Parameter durch aktuelle Parameter.
  - Erzeuge (der Reihe nach !) die durch die Zuweisungen gegebenen Name-Wert Zuordnungen und ersetze neue Namen im Rest des Rumpfes durch den Wert.

Variablen

C++ erlaubt aber auch, die Zuordnung von Werten zu Namen zu ändern:

Beispiel: (Variablen)

```
int hochacht( int x )
{
  int y = x*x;   // Zeile 1: Definition/Initialisierung
  y = y*y;       // Zeile 2: Zuweisung
  return y*y;
}
```

Bemerkung:

- Zeile 1 im Funktionsrumpf definiert eine Variable `y`, die Werte vom Typ `int` annehmen kann und initialisiert diese mit dem Wert des Ausdrucks `x*x`.
- Zeile 2 nennt man eine Zuweisung. Die links des = stehende Variable erhält den Wert des rechts stehenden Ausdrucks als neuen Wert.
- Beachte, dass der boolsche Operator „ist gleich“ (also die Abfrage nach Gleichheit) in C++ durch == notiert wird!
- Der Wert von `y` wird erst geändert, nachdem der Ausdruck rechts ausgewertet wurde.
- Der Typ einer Variablen kann aber nicht geändert werden!

Problematik der Zuweisung

Beispiel:

```
int bla( int x )
{
  int y = 3;                  // Zeile 1
  const int x1 = y*x;         // Zeile 2
  y = 5;                      // Zeile 3
  const int x2 = y*x;         // Zeile 4
  return x1*x2;               // Zeile 5
}
```

Bemerkung:

- Obwohl `x1` und `x2` durch denselben Ausdruck `y*x` definiert werden, haben sie im allgemeinen verschiedene Werte.
- Dies bedeutet das Versagen des Substitutionsmodells, bei dem ein Name im ganzen Funktionsrumpf durch seinen Wert ersetzt werden kann.
- Die Namensdefinitionen und Zuweisungen werden der Reihe nach abgearbeitet. Das Ergebnis hängt auch von dieser Reihenfolge ab. Dagegen war die Reihenfolge der Auswertung von Ausdrücken im Substitutionsmodell egal.

Umgebungsmodell

Wir können uns die Belegung der Variablen als Abbildung bzw. Tabelle vorstellen die jedem Namen einen Wert zuordnet:

$$w : \{ \text{ Menge der gültigen Namen } \} \to \{ \text{ Menge der Werte } \}.$$

Beispiel: Abbildung w bei Aufruf von `bla(4)` nach Zeile 4

Name	Typ	Wert
`x`	`int`	4
`y`	`int`	5
`x1`	`int`	12
`x2`	`int`	20

Definition: Der Ort, an dem diese Abbildung im System gespeichert wird, heißt Umgebung. Die Abbildung w heißt auch Bindungstabelle. Man sagt, w bindet einer Namen an einen Wert.

Bemerkung:

- Ein Ausdruck wird in Zukunft immer relativ zu einer Umgebung ausgewertet, d. h. nur Ausdruck und Umgebung zusammen erlauben die Berechnung des Wertes eines Ausdruckes.

- Die Zuweisung können wir nun als Modifikation der Bindungstabelle verstehen: nach der Ausführung von y=5 gilt $w(\mathrm{y}) = 5$.

- Die Bindungstabelle ändert sich dynamisch während der Programmausführung. Um herauszufinden „was ein Programm tut“ muss sich der Programmierer die fortwährende Entwicklung der Bindungstabelle vorstellen.

Syntax von Variablendefinition und Zuweisung

Syntax:

$$
\begin{array}{lll}
\langle \text{Def} \rangle & ::= & \langle \text{ConstDef} \rangle \mid \langle \text{VarDef} \rangle \\
\langle \text{ConstDef} \rangle & ::= & \underline{\text{const}}\ \langle \text{Typ} \rangle\ \langle \text{Name} \rangle\ \equiv\ \langle \text{Ausdruck} \rangle \\
\langle \text{VarDef} \rangle & ::= & \langle \text{Typ} \rangle\ \langle \text{Name} \rangle\ [\ \equiv\ \langle \text{Ausdruck} \rangle\ ]
\end{array}
$$

Syntax:

$$
\langle \text{Zuweisung} \rangle \quad ::= \quad \langle \text{Name} \rangle\ \equiv\ \langle \text{Ausdruck} \rangle
$$

Bemerkung:

- Wir erlauben zunächst Variablendefinitionen nur innerhalb von Funktionsdefinitionen. Diese Variablen bezeichnet man als lokale Variablen.

- Bei der Definition von Variablen $kann$ die Initialisierung weggelassen werden. In diesem Fall ist der Wert der Variablen bis zur ersten Zuweisung unbestimmt. Aber: Fast immer ist es empfehlenswert, auch Variablen gleich bei der Definition zu initialisieren!

Lokale Umgebung

Wie sieht die Umgebung im Kontext mehrerer Funktionen aus?

Programm:

```
int g( int x )
{
  int y = x*x;
  y = y*y;
  return h( y*(x+y) );
}

int h( int x )
{
  return cond( x<1000, g(x), 88 );
}
```

Es gilt folgendes:

- Jede Auswertung einer Funktion erzeugt eine eigene lokale Umgebung. Mit Beendigung der Funktion wird diese Umgebung wieder vernichtet!
- Zu jedem Zeitpunkt der Berechnung gibt es eine aktuelle Umgebung. Diese enthält die Bindungen der Variablen der Funktion, die gerade ausgewertet wird.
- In Funktion `h` gibt es keine Bindung für `y`, auch wenn `h` von `g` aufgerufen wurde.
- Wird eine Funktion n mal rekursiv aufgerufen, so existieren n verschiedene Umgebungen für diese Funktion.

Bemerkung: Man beachte auch, dass eine Funktion kein Gedächtnis hat: wird sie mehrmals mit gleichen Argumenten aufgerufen, so sind auch die Ergebnisse gleich. Diese fundamentale Eigenschaft funktionaler Programmierung ist also (bisher) noch erhalten.

Bemerkung: Tatsächlich wäre obiges Konstrukt auch nach Einführung einer `main`-Funktion nicht kompilierbar, weil die Funktion `h` beim Übersetzen von `g` noch nicht bekannt ist. Um dieses Problem zu umgehen, erlaubt C++ die vorherige Deklaration von Funktionen. In obigem Beispiel könnte dies geschehen durch Einfügen der Zeile

```
int h( int x );
```

vor die Funktion `g`.

Anweisungsfolgen (Sequenz)

- Funktionale Programmierung bestand in der Auswertung von Ausdrücken.
- Jede Funktion hatte nur eine einzige Anweisung (`return`).
- Mit Einführung von Zuweisung (oder allgemeiner Nebeneffekten) macht es Sinn, die *Ausführung mehrer Anweisungen* innerhalb von Funktionen zu erlauben. Diesen Programmierstil nennt man auch *imperative Programmierung*.

Erinnerung: Wir kennen schon eine Reihe wichtiger Anweisungen:

- Variablendefinition (ist in C++ eine Anweisung, nicht aber in C),
- Zuweisung,
- `return`-Anweisung in Funktionen.

Bemerkung:

- Jede Anweisung endet mit einem Semikolon.
- Überall wo eine Anweisung stehen darf, kann auch eine durch geschweifte Klammern eingerahmte Folge ($Sequenz$) von Anweisungen stehen.
- Auch die leere Anweisung ist möglich indem man einfach ein Semikolon einfügt.
- Anweisungen werden der Reihe nach abgearbeitet.

Syntax: (Anweisung)

$$
\begin{array}{lll}
<\text{Anweisung}> & ::= & <\text{EinfacheAnw}> \mid \underline{\{} \ \{ <\text{EinfacheAnw}> \}^{+} \ \underline{\}} \\
<\text{EinfacheAnw}> & ::= & <\text{VarDef}> \ \underline{;} \mid <\text{Zuweisung}> \ \underline{;} \mid \\
 & & <\text{Selektion}> \mid \ldots
\end{array}
$$

Beispiel

Die folgende Funktion berechnet `fib(4)`. `b` enthält die letzte und `a` die vorletzte Fibonaccizahl.

```
int f4()
{
  int a = 0;        // a = fib(0)
  int b = 1;        // b = fib(1)
  int t;

  t = a+b; a = b; b = t;     // b = fib(2)
  t = a+b; a = b; b = t;     // b = fib(3)
  t = a+b; a = b; b = t;     // b = fib(4)
  return b;
}
```

Bemerkung: Die Variable `t` wird benötigt, da die beiden Zuweisungen

$$\left\{ \begin{array}{lcl} \texttt{b} & \leftarrow & \texttt{a+b} \\ \texttt{a} & \leftarrow & \texttt{b} \end{array} \right\}$$

nicht gleichzeitig durchgeführt werden können.

Bemerkung: Man beachte, dass die Reihenfolge in

```
t = a+b;
a = b;
b = t;
```

nicht vertauscht werden darf. In der funktionalen Programmierung mussten wir hingegen weder auf die Reihenfolge achten noch irgendwelche „Hilfsvariablen“ einführen.

Bedingte Anweisung (Selektion)

Anstelle des `cond`-Operators wird in der imperativen Programmierung die bedingte Anweisung verwendet.

Syntax: **(Bedingte Anweisung, Selektion)**

$$<\text{Selektion}> \quad ::= \quad \underline{\text{if}}\ \underline{(}\ <\text{BoolAusdr}>\ \underline{)}\ <\text{Anweisung}> \\ [\ \underline{\text{else}}\ <\text{Anweisung}>\ ]$$

Ist die Bedingung in runden Klammern wahr, so wird die erste Anweisung ausgeführt, ansonsten die zweite Anweisung nach dem `else` (falls vorhanden).

Genauer bezeichnet man die Variante **ohne** `else` als bedingte Anweisung, die Variante **mit** `else` als Selektion.

Beispiel: Die funktionale Form

```
int absolut( int x )
{
  return cond( x<=0, -x, x );
}
```

ist äquivalent zu

```
int absolut( int x )
{
  if ( x <= 0 )
    return -x;
  else
    return x;
}
```

While-Schleife

Iterative Prozesse sind so häufig, dass man hierfür eine Abstraktion schaffen will. In C++ gibt es dafür verschiedene imperative Konstrukte, die wir jetzt kennenlernen.

Programm: (Fakultät mit While-Schleife [fakwhile.cc])

```
int fak( int n )
{
  int ergebnis = 1;
  int zaehler = 2;

  while ( zaehler <= n )
  {
    ergebnis = zaehler*ergebnis;
    zaehler  = zaehler+1;
  }
  return ergebnis;
}
```

Syntax: (While-Schleife)

<WhileSchleife> ::= while (<BoolAusdr>) <Anweisung>

Die Anweisung wird solange ausgeführt wie die Bedingung erfüllt ist.

Wir überlegen informell warum das Beispiel funktioniert.

- Ist $n = 0$ oder $n = 1$, also $n < 2$ (andere Zahlen sind nicht erlaubt), so ist die Schleifenbedingung nie erfüllt und das Ergebnis ist 1, was korrekt ist.

- Ist $n \geq 2$ so wird die Schleife mindestens einmal durchlaufen. In jedem Durchlauf wird der zaehler dranmultipliziert und dann erhöht. Es werden also sukzessive die Zahlen $2, 3, \ldots$ an den aktuellen Wert multipliziert. Irgendwann erreicht zaehler den Wert n und damit ergebnis den Wert $n!$. Da zaehler nun den Wert $n + 1$ hat, wird die Schleife verlassen.

Später werden wir eine formale Methode kennenlernen, mit der man beweisen kann, dass das Programm korrekt funktioniert.

For-Schleife

Die obige Anwendung der `while`-Schleife ist ein Spezialfall, der so häufig vorkommt, dass es dafür eine Abkürzung gibt:

Syntax: (For-Schleife)

<ForSchleife> ::= for (<Init> ; <BoolAusdr> ; <Increment>)
<Anweisung>
<Init> ::= <VarDef> | <Zuweisung>
<Increment> ::= <Zuweisung>

Init entspricht der Initialisierung des Zählers, BoolAusdr der Ausführungsbedingung und Increment der Inkrementierung des Zählers.

Programm: (Fakultät mit For-Schleife [fakfor.cc])

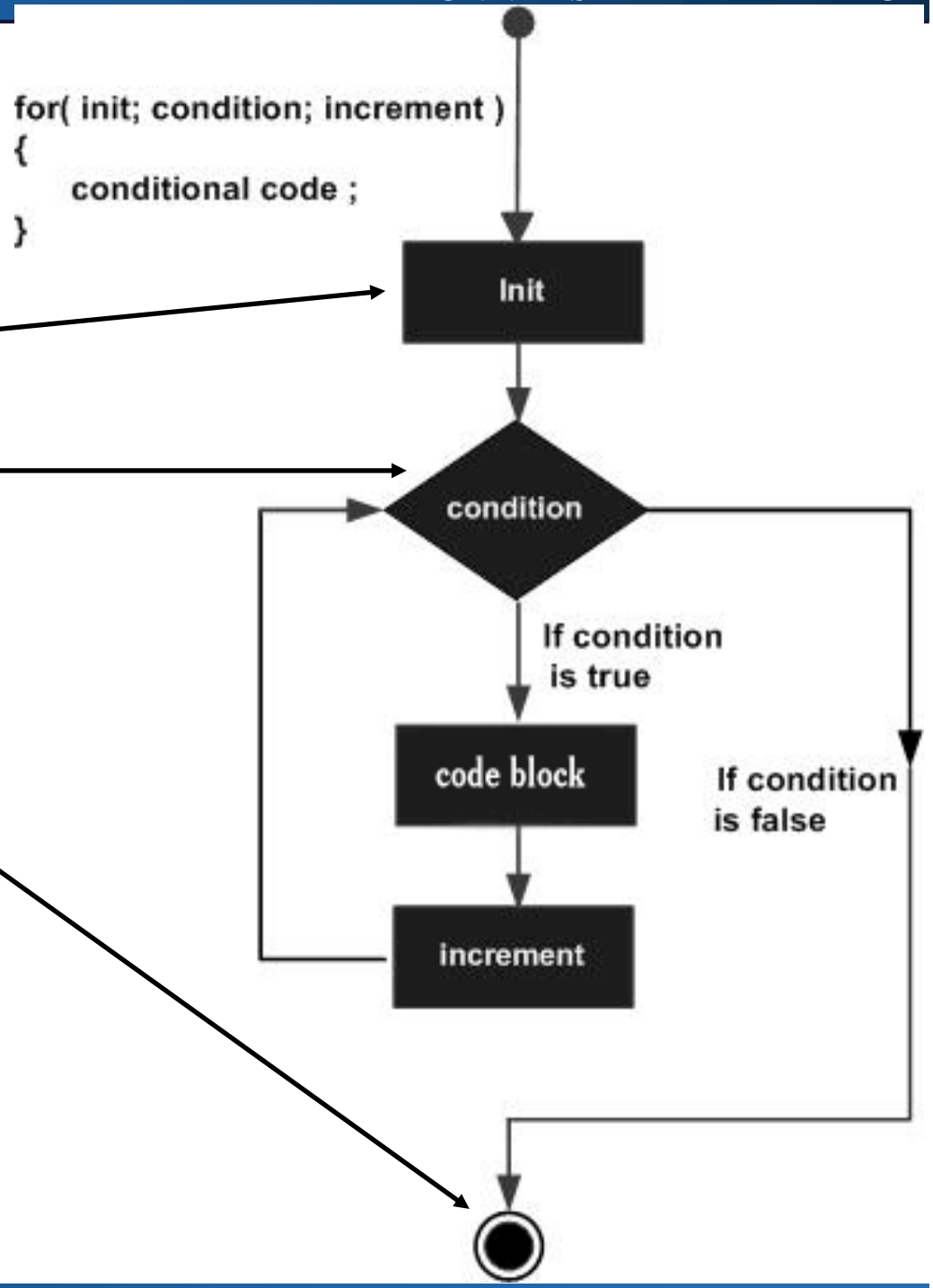
```
int fak( int n )
{
  int ergebnis = 1;

  for ( int zaehler=2; zaehler<=n; zaehler=zaehler+1 )
  {
    ergebnis = zaehler*ergebnis;
  }

  return ergebnis;
}
```

Bemerkungen

- Eine `For`-Schleife kann direkt in eine `While`-Schleife transformiert werden. Dabei wird die Laufvariable vor der Schleife definiert.
- Der Gültigkeitsbereich von `zaehler` erstreckt sich nur über die `for`-Schleife (ansonsten hätte man es wie `ergebnis` außerhalb der Schleife definieren müssen). Wir werden später sehen wie man den Gültigkeitsbereich gezielt mit neuen Umgebungen kontrollieren kann.
- Die Initialisierungsanweisung enthält Variablendefinition und Initialisierung.
- Wie beim Fakultätsprogramm mit `while` wird die Inkrementanweisung am Ende des Schleifendurchlaufes ausgeführt.

Ablaufdiagramm

- Operation
- Verzweigung
- Terminal



Wiederholung

Lokale Konstanten und Variablen:

```
int f( int x )
{
  const int a = 3; // Konstante, kann nicht geaendert werden

  int b = x;       // Variablendefinition mit Initialisierung

  int c;           // Variablendefinition ohne Initialisierung
}
```

Zuweisung:

```
c = 3 * a + f(b); // Variable wird Wert des Ausdrucks " zugewiesen "
```

Auswertung eines Ausdrucks erfolgt relativ zu einer Umgebung, die eine Bindungstabelle enthält.

Anweisungen und Anweisungsfolgen:

- Anweisungen werden immer mit einem Semikolon beendet.
- Anweisungsfolgen werden der Reihe nach bearbeitet und in { } gesetzt.
- Wo eine einzelne Anweisung stehen kann darf auch eine Folge von Anweisungen stehen.

`if`-Anweisung (Bedingte Anweisung, Selektion):

```
if ( a < 0 ) b = f(c); else b = f(3);
if ( a < 0 ) { b = f(c); c = 0; } else { b = f(3); c = 5; }
```

`while`-Schleife:

```
int i = 10; while ( i >= 0 ) i = i-1;
int i = 10; int b = 0; while ( i >= 0 ) { b = b+f(i); i = i-1; }
```

`for`-Schleife:

```
for ( int i=0; i>=0; i=i-1 ) ;
int b = 0; for ( int i=0; i>=0; i=i-1 ) b = b+f(i);
```

Beispiele

Wir benutzen nun die neuen Konstruktionselemente um die iterativen Prozesse zur Berechnung der Fibonaccizahlen und der Wurzelberechnung nochmal zu formulieren.

Programm: (Fibonacci mit For-Schleife [fibfor .cc])

```
int fib( int n )
{
  int a = 0;
  int b = 1;
  for ( int i=0; i<n; i=i+1 )
  {
    int t = a+b; a = b; b = t;
  }
  return a;
}
```

Programm: (Newton mit While-Schleife [newtonwhile.cc])

```
#include "fcpp.hh"

double wurzel( double a )
{
  double x = 1.0;

  while ( fabs(x*x - a) > 1e-12 )
    x = 0.5 * (x + a/x);
  return x;
}

int main()
{
  return print( wurzel(2.0) );
}
```

Goto

Neben den oben eingeführten Schleifen gibt es eine alternative Möglichkeit die Wiederholung zu formulieren. Wir betrachten nochmal die Berechnung der Fakultät mittels einer `while`-Schleife:

```
int t = 1;
int i = 2;

while ( i <= n )
{
  t = t*i;
  i = i+1;
}
```

Goto

Mit der goto-Anweisung kann man den Programmverlauf an einer anderen, vorher markierten Stelle fortsetzen:

```
int t = 1; int i = 2;
anfang: if ( i > n ) return t;
t = t*i;
i = i+1;
goto anfang;
```

- anfang nennt man eine Sprungmarke (engl.: label). Jede Anweisung kann mit einer Sprungmarke versehen werden.
- Der Sprung kann nur innerhalb einer Funktion erfolgen.
- `While`- und `For`-Schleife können mittels `goto` und Selektion realisiert werden.

In einem berühmten Letter to the Editor [*Go To Statement Considered Harmful, Communications of the ACM, Vol. II, Number 3, 1968*] hat Edsger W. Dikstra[13] dargelegt, dass `goto` zu sehr unübersichtlichen Programmen führt und nicht verwendet werden sollte.

Man kann zeigen, dass `goto` nicht notwendig ist und man mit den obigen Schleifenkonstrukten auskommen kann. Dies nennt man strukturierte Programmierung. Die Verwendung von `goto` in C/C++ Programmen gilt daher als verpönt und schlechter Programmierstil!

Eine abgemilderte Form des `goto` stellen die **break**- und **continue**-Anweisung dar. Diese erhöhen, mit Vorsicht eingesetzt, die Übersichtlichkeit von Programmen.

[13] Edsger Wybe Dijkstra, 1930–2002, niederländischer Informatiker.

Regeln guter Programmierung

1. Einrückung sollte verwendet werden um Schachtelung von Schleifen bzw. `if`-Anweisungen anzuzeigen:

```
if ( x >= 0 )
{
  if ( y <= x )
    b = x - y; // b ist groesser 0
  else
  {
    while ( y > x )
      y = y - 1;
    b = x + y;
  }
  i = f(b);
}
```

Regeln guter Programmierung

2. Verwende möglichst sprechende Variablennamen! Kurze Variablennamen wie i, j, k sollten nur innerhalb (kurzer) Schleifen verwendet werden (oder wenn sie der mathematischen Notation entsprechen).

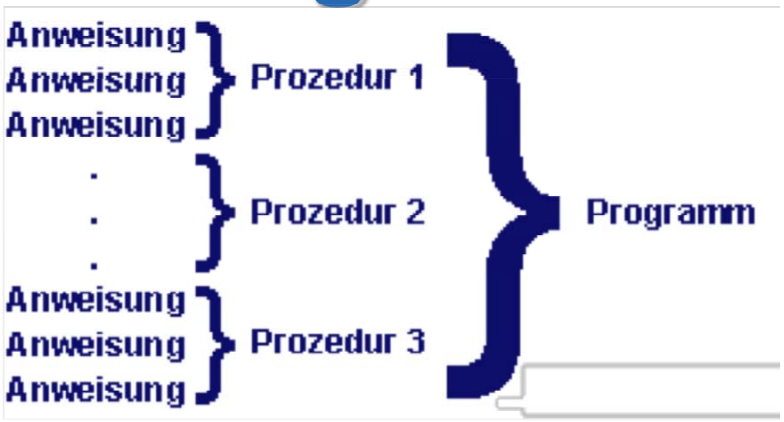
3. Beschränke die Gültigkeit von Konstanten und Variablen auf den kleinstem möglichen Bereich.

4. Nicht mit Kommentaren sparen! Wichtige Anweisungen oder Programmzweige sollten dokumentiert werden. Beim „Programmieren im Großen“ ist die Programmdokumentation natürlich ein wesentlicher Bestandteil der Programmierung.

5. Verletzung dieser Regeln werden wir in den Übungen ab sofort mit Punktabzug belegen!

6. To be continued . . .

Prozedurale Programmierung



- Lokale Variablen und die Zuweisung
- Syntax von Variablen-definition und Zuweisung
- Anweisungsfolgen (Sequenzen)
- Bedingte Anweisung (Selektion)
- Schleifen
- Formale Programmverifikation
- Prozeduren und Funktionen

Formale Programmverifikation

Das Verständnis selbst einfacher imperativer Programme bereitet einige Mühe. Übung und Erfahrung helfen hier zwar, aber trotzdem bleibt der Wunsch formal beweisen zu können, dass ein Programm „funktioniert“. Dies gilt insbesondere für sicherheitsrelevante Programme.

Eine solche „formale Programmverifikation“ erfordert folgende Schritte:

1. Eine formale Beschreibung dessen was das Programm leisten soll. Dies bezeichnet man als Spezifikation.

2. Einen Beweis dass das Programm die Spezifikation erfüllt.

3. Dies erfordert eine formale Definition der Semantik der Programmiersprache.

Formale Programmverifikation

Beginnen wir mit dem letzten Punkt. Hier haben sich unterschiedliche Vorgehensweisen herausgebildet, die wir kurz beschreiben wollen:

- Operationelle Semantik. Definiere eine Maschine, die direkt die Anweisungen der Programmiersprache verarbeitet.

- Denotationelle Semantik. Beschreibe Wirkung der Anweisungen der Programmiersprache als Zustandsänderung auf den Variablen:
  - $v_1, \ldots, v_m$ seien die Variablen im Programm. $W(v_i)$ der Wertebereich von v_i.
  - $Z = W(v_1) \times \ldots \times W(v_m)$ ist die Menge aller möglichen Zustände.
  - Sei a eine Anweisung, dann beschreibt $F_a : Z \to Z$ die Wirkung der Anweisung.

Formale Programmverifikation

- Axiomatische Semantik. Beschreibe Wirkung der Anweisungen mittels prädikatenlogischer Formeln. Man schreibt

$$P \ \{a\} \ Q$$

wobei P, Q Abbildungen in die Menge $\{\text{wahr}, \text{falsch}\}$, sog. Prädikate, und a eine Anweisung ist.

$P \ \{a\} \ Q$ bedeutet dann:

– Wenn P **vor** der Ausführung von a wahr ist, dann gilt Q **nach** der Ausführung von a (P impliziert Q).
– P heißt auch **Vorbedingung** und Q **Nachbedingung**.

Beispiel:

$$-1000 < x \leq 0 \ \{x = x - 1\} \ -1000 \leq x < 0$$

Der oben beschriebene Formalismus der axiomatischen Semantik heisst auch Hoare[14]-Kalkül. Für die gängigen Konstrukte imperativer Programmiersprachen, wie Zuweisung, Sequenz und Selektion, lassen sich Zusammenhänge zwischen Vor- und Nachbedingung herleiten.

Sei nun S ein Programmfragment oder gar das ganze Programm. Dann lassen sich mit dem Hoare-Kalkül entsprechende P und Q finden so dass

$$P \; \{S\} \; Q.$$

Schließlich lässt sich die Spezifikation des Programms ebenfalls mittels prädikatenlogischer Formeln ausdrücken. P_{SPEC} bezeichnet entsprechend die Vorbedingung (Bedingung an die Eingabe) unter der das Ergebnis Q_{SPEC} berechnet wird.

[14] Sir Charles Anthony Richard Hoare, geb. 1934, brit. Informatiker.

Für ein gegebenes Programm S, welches die Spezifikation implementieren soll, sei nun $P_{\text{PROG}}\ \{S\}\ Q_{\text{PROG}}$ gezeigt. Der formale Prozess der Programmverifikation besteht dann im folgenden Nachweis:

$$(P_{\text{SPEC}} \Rightarrow P_{\text{PROG}}) \wedge (P_{\text{PROG}}\ \{S\}\ Q_{\text{PROG}}) \wedge (Q_{\text{PROG}} \Rightarrow Q_{\text{SPEC}}). \tag{1}$$

Dabei kann z. B. P_{SPEC} eine stärkere Bedingung als P_{PROG} sein. Beispiel:

$$-1000 < x \leq 0 \ \Rightarrow\ x \leq 0.$$

Der Nachweis von (1) liefert erst die **partielle Korrektheit**. Getrennt davon ist zu zeigen, dass das Programm terminiert. Kann man dies nachweisen ist der Beweis der **totalen Korrektheit** erbracht.

Der Nachweis von $P_{\text{PROG}}\ \{S\}\ Q_{\text{PROG}}$ kann durch automatische Theorembeweiser unterstützt werden.

Korrektheit von Schleifen mittels Schleifeninvariante

Wir betrachten nun eine Variante des Hoare-Kalküls, mit der sich die Korrektheit von Schleifenkonstrukten nachweisen lässt. Dazu betrachten wir eine `while`-Schleife in der kanonischen Form

while (B(v)) { v = H(v); }

mit

- $v = (v_1, \ldots, v_m)$ dem Vektor von Variablen, die im Rumpf modifiziert werden,
- $B(v)$, der Schleifenbedingung und
- $H(v) = (H_1(v_1, \ldots, v_m), \ldots, H_m(v_1, \ldots, v_m))$ dem **Schleifentransformator**.

Korrektheit von Schleifen mittels Schleifeninvariante

Als Beispiel dient die Berechnung der Fakultät. Dort lautet die Schleife:

```
while ( i <= n ) { t = t*i; i = i+1; }
```

Also

$$v = (t, i) \qquad B(v) \equiv i \leq n \qquad H(v) = (t * i, i + 1).$$

Zusätzlich definieren wir die Abkürzung

$$H^j(v) = \underbrace{H(H(\ldots H(}_{j \text{ mal}} v) \ldots)).$$

Definition: (Schleifeninvariante)
Sei $v^j = H^j(v^0)$, $j \in \mathbb{N}_0$, die Belegung der Variablen nach j-maligem Durchlaufen der Schleife. Eine Schleifeninvariante INV(v) erfüllt:

1. INV(v^0) ist wahr.

2. INV$(v^j) \wedge B(v^j) \Rightarrow$ INV(v^{j+1}).

Gilt die Invariante vor Ausführung der Schleife und ist die Schleifenbedingung erfüllt, dann gilt die Invariante nach Ausführung des Schleifenrumpfes.

Angenommen, die Schleife wird nach k-maligem Durchlaufen verlassen, d. h. es gilt $\neg B(v^k)$. Ziel ist es nun zu zeigen, dass

$$\text{INV}(v^k) \wedge \neg B(v^k) \Rightarrow Q(v^k)$$

wobei $Q(v^k)$ die geforderte Nachbedingung ist.

Beispiel: Betrachte das Programm zur Berechnung der Fakultät von n:

```
t = 1; i = 2;
while ( i <= n ) { t = t*i; i = i+1; }
```

Behauptung: Sei $n \geq 1$, dann lautet die Schleifeninvariante:

$$\text{INV}(t, i) \equiv t = (i-1)! \wedge i - 1 \leq n.$$

1. Für $v^0 = (t^0, i^0) = (1, 2)$ gilt $\text{INV}(1, 2) \equiv 1 = (2-1)! \wedge (2-1) \leq n$. Wegen $n \geq 1$ ist das immer wahr.

2. Es gelte nun $\text{INV}(v^j) \equiv t^j = (i^j - 1)! \wedge i^j - 1 \leq n$ sowie $B(v^j) = i^j \leq n$. (Vorsicht v^j bedeutet **nicht** v hoch j!). Dann gilt

   - $t^{j+1} = t^j \cdot i^j = (i^j - 1)! \cdot i^j = i^j!$
   - $i^{j+1} = i^j + 1$, somit gilt wegen $i^j = i^{j+1} - 1$ auch $t^{j+1} = (i^{j+1} - 1)!$.
   - Schließlich folgt aus $B(i^j, t^j) \equiv i^j = i^{j+1} - 1 \leq n$ dass $\text{INV}(i^{j+1}, t^{j+1})$ wahr.

3. Am Schleifenende gilt $\neg(i \leq n)$, also $i > n$, also $i = n + 1$ da i immer um 1 erhöht wird. Damit gilt dann also

$$\begin{aligned} & \mathsf{INV}(i,t) \wedge \neg B(i,t) \\ \Leftrightarrow \quad & t = (i-1)! \wedge i - 1 \leq n \wedge i = n + 1 \\ \Leftrightarrow \quad & t = (i-1)! \wedge i - 1 = n \\ \Rightarrow \quad & t = n! \equiv Q(n) \end{aligned}$$

Für den Fall $n \geq 0$ muss man den Fall $0! = 1$ als Sonderfall hinzunehmen. Das Programm ist auch für diesen Fall korrekt und die Schleifeninvariante lautet $\mathsf{INV}(i,t) \equiv (t = (i-1)! \wedge i - 1 \leq n) \vee (n = 0 \wedge t = 1 \wedge i = 2)$.

Prozeduren und Funktionen

In der Mathematik ist eine Funktion eine Abbildung $f : X \to Y$. C++ erlaubt entsprechend die Definition n-stelliger Funktionen

```
int f( int x1, int x2 ) { return x1*x1 + x2; }
...
int y = f( 2, 3 );
```

In der Funktionalen Programmierung ist das einzig interessante an einer Funktion ihr Rückgabewert. Seiteneffekte spielen keine Rolle. In der Praxis ist das jedoch anders. Betrachte

```
void drucke( int x )
{
  int i = print( x ); // print aus fcpp.hh
}
...
drucke( 3 );
...
```

Prozeduren und Funktionen

Es macht durchaus Sinn eine Funktion definieren zu können, deren einziger Zweck das Ausdrucken des Arguments ist. So eine Funktion hat keinen sinnvollen Rückgabewert, ihr einziger Zweck ist der Seiteneffekt.

C++ erlaubt dafür den Rückgabetyp **void** (nichts). Der Funktionsrumpf darf dann keine **return**-Anweisung enthalten, welche einen Wert zurückgibt.

Der Funktionsaufruf ist eine gültige Anweisung, allerdings ist dann keine Zuweisung des Rückgabewerts erlaubt (die Funktion gibt keinen Wert zurück).

Funktionen, die keine Werte zurückliefern heißen Prozeduren. In C++ werden Prozeduren durch den Rückgabetyp **void** gekennzeichnet.

C++ erlaubt auch die Verwendung von Funktionen als Prozeduren, d. h. man verwendet einfach den Rückgabewert **nicht**.