Ergänzendes Skript zur Vorlesung Prog 1 & 2 an der FH-Bingen

C-Progammierung

Bei Prof. Dr. M. Mengel

Fachbereich 2

Studiengänge :

Angewandte Informatik

Ingenieur-Informatik

Elektrotechnik

Inhaltsverzeichnis

Programmieren 1
-
Ziele

- Allgemeines Verständnis von Algorithmen bzw. deren Formulierung

- Entwicklung von Algorithmen
  Motto: vom Problem zur Problemlösung

- Erlernen der Programmiersprache C
  (Sprachelemente von C findet man auch in anderen Programmiersprachen – C++, Java, JavaScript, PHP, ...)

- Beschreibung von Algorithmen in C

- Strukturierter Entwurf und strukturiertes Programmieren

Vom Problem zur Problemlösung

Aufgabe:

Geben Sie die einzelnen Schritte an, die Sie durchführen müssen, wenn Sie an einem Münzautomaten telefonieren.

==> Vorschlag für Vorgehensweise:

Beschreibung schrittweise entwickeln und immer weiter verfeinern

Aufgabe zerlegen und strukturieren

Problem: wie genau? was ist die kleinste Einheit?

Vorbedingungen und Nachbedingungen formulieren

Für wen wurde die Beschreibung gemacht?

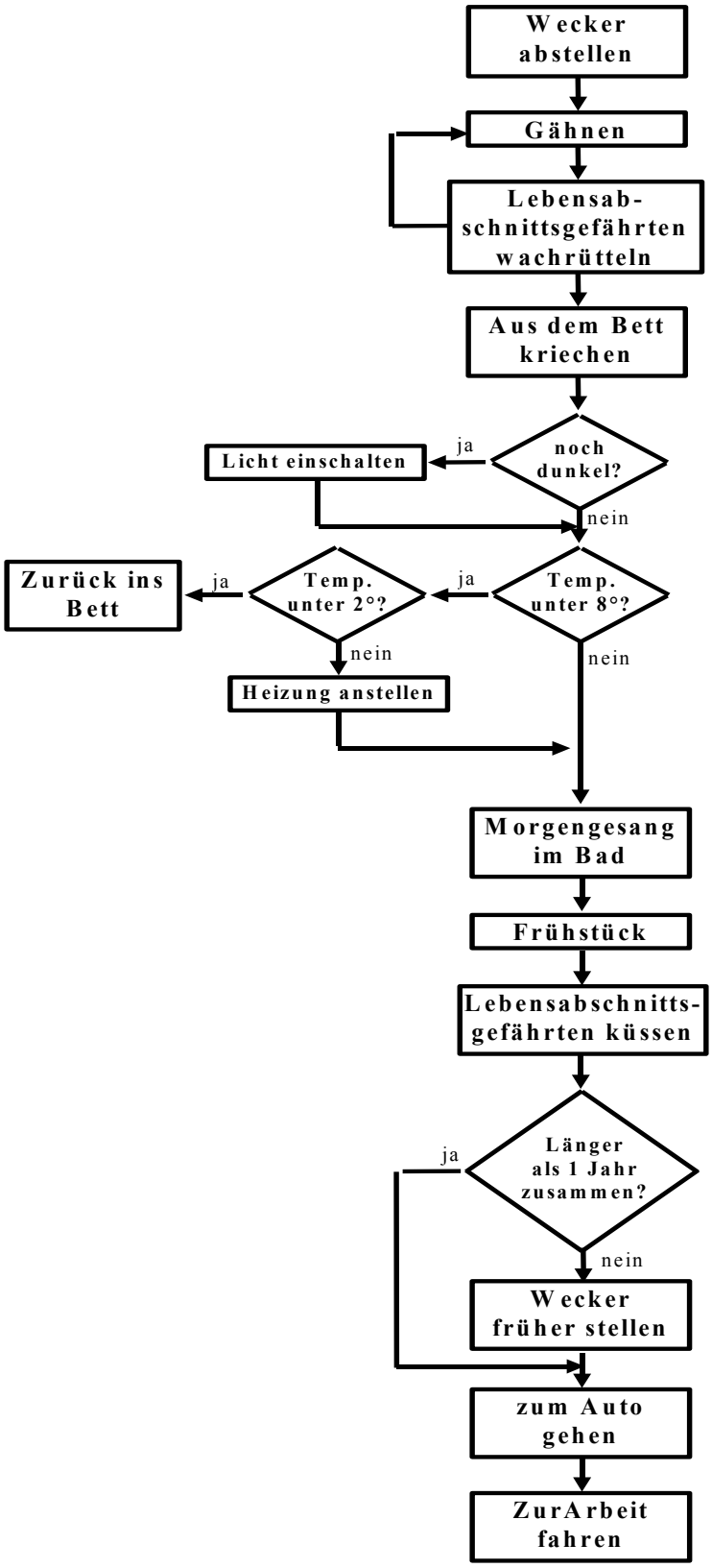
Die letzte Fragestellung führt uns zu unserem **Bezugssystem** oder einer **formalen Verfahrensbeschreibung**

- Ablaufplan
- Struktogramm
- Programmiersprache

Wir werden Verfahren/Beschreibungen betrachten, die von einer Maschine (=Computer) ausgeführt werden können.

Aufgabe

- Geben Sie den Algorithmus, der Ihren morgentlichen Aufbruch zur FH beschreibt
- Tic-Tac-Toe (3 Gewinnt) - Beschreiben Sie das Spiel Tic-Tac-Toe, so dass es anschließend implementiert werden könnte.


Wecker abstellen
Gähnen
Lebensab-schnittsgefährten wachrütteln
Aus dem Bett kriechen
noch dunkel?
ja
Licht einschalten
nein
Temp. unter 8°?
ja
Temp. unter 2°?
ja
Zurück ins Bett
nein
Heizung anstellen
nein
Morgengesang im Bad
Frühstück
Lebensabschnitts-gefährten küssen
Länger als 1 Jahr zusammen?
ja
nein
Wecker früher stellen
zum Auto gehen
ZurArbeit fahren

Algorithmusbegriff

Ein Algorithmus ist - intuitiv gesehen - eine Anleitung zur Lösung eines Problems, wie auch ein Kochrezept, ein Strickmuster oder eine Reparatur- oder Montageanweisung.

Was einen Algorithmus jedoch von einem Kochrezept abhebt, ist die Tatsache, dass er von einer Maschine ausgeführt werden muß. Dies bedeutet, dass ein Programm so geschrieben sein muß, dass der Computer es ausführen kann, ohne es inhaltlich zu verstehen. Wie bei jeder anderen Sprache müssen hier die Regeln der **Syntax** (Einhaltung der formalen Regeln) und die der **Semantik** (Bedeutung der Konstrukte untereinander) eingehalten werden.

Er bildet eine Einheit aus den dafür festgelegten Datenstrukturen und Operationen.

Eigenschaften eines Algorithmus

Algorithmen die wir betrachten müssen folgende grundsätzlichen Eigenschaften erfüllen:

Allgemeinheit

Ein Algorithmus löst im allgemeinen eine Klasse von Problemen. Die Auswahl des Einzelfalles erfolgt meist über Parameter.

Ein Algorithmus entsteht oft dadurch, dass ein bestimmter Fall gelöst wird und andere Situationen darauf zurück geführt werden.

Determiniertheit

Algorithmen sind in der Regel determiniert, d.h. bei gleichen Eingabewerten und Startbedingungen erfolgt stets dasselbe Ergebnis. Beispiele für nicht determinierte Algorithmen sind z.B. stochastische Simulationen mit Hilfe von Zufallszahlen.

Determinismus

Ein Algorithmus heißt deterministisch, wenn zu jedem Zeitpunkt seiner Bearbeitung höchstens eine Möglichkeit der Fortsetzung besteht.

Terminierung

In der Regel sind nur solche Algorithmen von Interesse, die für jede Eingabe nach endlichen vielen Schritten terminieren, d.h. anhalten. Eine Ausnahme sind hier Betriebssystem-Funktionen und Prozeß-Steuerungen.

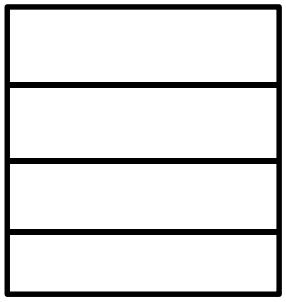
Beschreibung von Algorithmen

Es gibt mehrere Möglichkeiten, einen Algorithmus anzugeben:

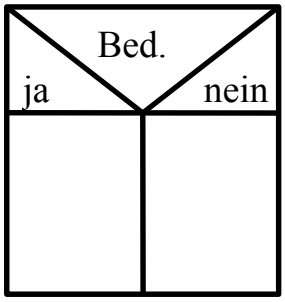
- umgangssprachlich
- Pseudocode
- Ablaufplan, **Struktogramm**
- Mathematische Beschreibung
- Programmiersprache

Ein Struktogramm ist ein Hilfsmittel mit dem Algorithmen in formalisierter Form dargestellt werden können. Die graphische Darstellungsmethode zeichnet sich insbesondere durch die überschaubare Anzahl von elementaren Strukturelementen aus, die bel. miteinander kombiniert werden können und durch ihre Übersichtlichkeit.

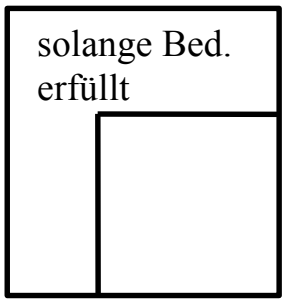
Ein Strukturblock ist entweder vollständig in einem anderen enthalten oder er steht außerhalb von diesem:



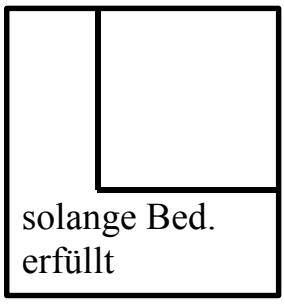
Sequenz
(Folge von
Anweisungen):
-Zuweisung
-Eingabe
-Ausgabe



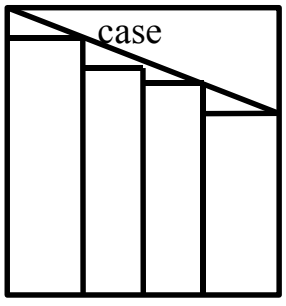
Bedingung
(Verzweigung)



Abweisende
Schleife



nicht abweisende
Schleife



1-aus-n
Auswahl

Vom Algorithmus zum Programm

Wenn man einen Algorithmus auf einem Computer ausführen möchte, muss dieser in ein Programm überführt werden. Neben der Beschreibung des Algorithmus in einer Programmiersprache bedarf es verschiedener Werkzeuge um ein lauffähiges Programm zu erhalten.

Je nachdem ob eine Compiler- oder eine Interpreter-Sprache zur Formulierung eines Programms verwendet wird, gibt es bei der Programmentwicklung einen Entwicklungszyklus:

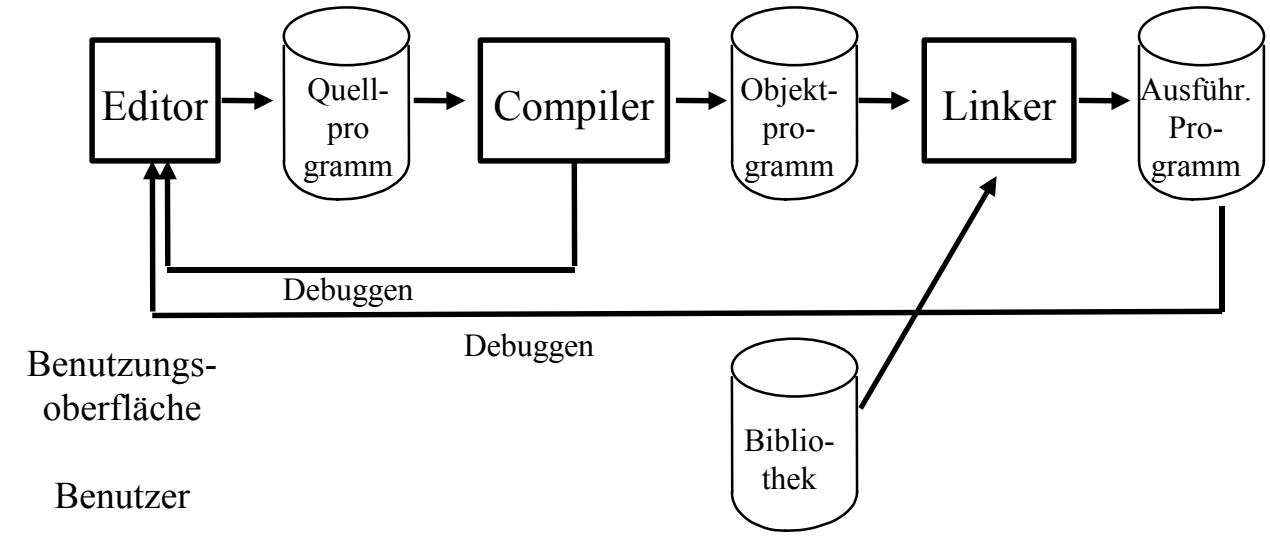


Abbildung: Programmentwicklung mit einer Compiler-Sprache

Bei Programmen in einer Compilersprache muß der Code von einem Compiler zuerst vollständig in Maschinensprache übersetzt werden, bevor er ausgeführt werden kann, während bei Interpretersprachen jede Anweisung bei der Ausführung interpretiert wird.

In jedem Fall muß zuerst der Programmtext (Quelltext - *.pas-, *.c- oder *.cpp-Dateien) mit einem Texteditor in eine Datei geschrieben werden. Im Falle einer Compilersprache wird dieser Quelltext vom Übersetzer (Compiler) gelesen, der daraus eine Objektdatei (*.obj-Dateien) macht, die aus Maschinenbefehlen besteht, aber noch „verschiebbar“ ist, d.h. noch nicht auf endgültige Adressen im Hauptspeicher festgelegt ist.

Der Binder (Linker) bindet mehrere Objektdateien zu einem ausführbaren Programm zusammen. Dazu können auch Objektdateien aus Bibliotheken. Zuletzt wird das Lauffähige Programm (executable) gestartet, das heißt, es wird in den Hauptspeicher gebracht und bekommt den Prozessor zugeteilt.

Kodieren bedeutet aber immer auch, dass der implementierte Code verbessert werden muß. Es müssen nach und nach syntaktische und unter Umständen nach Tests semantische Fehler verbessert werden. Im schlimmsten Fall muß der Algorithmus komplett neu kodiert werden. Dann hat man allerdings vermutlich zu Beginn der Programmentwicklung einen grundsätzlichen Fehler begangen.

Schritte zur algorithmischen Lösung von Problemen

Die algorithmische Lösung eines Problems besteht nicht in einem einzigen Schritt, nämlich dem Schreiben eines C-Programms, sondern es lassen sich mehrere Teilschritte identifizieren, die beim Entwurf von Programmen auftreten:

1. Formulierung des Problems
2. Die formale, abstrakte Beschreibung des Problems (z.B. mit Hilfe der Prädikatenlogik als Vor- und Nachbedingung)
3. Entwurf eines Lösungsalgorithmus
4. Kontrolle und Nachweis der Korrektheit des Lösungsalgorithmus
5. Die Übertragung des Lösungsalgorithmus in eine Programmiersprache (sogenannte Kodierung)
6. Fehlerbeseitigung und Tests
7. Die Effizienzuntersuchung
8. Die Dokumentation

Allgemein und insbesondere bei C ist es wichtig, dass man beim Einstieg in die Programmiersprache besonders sorgfältig vorgeht und dadurch langwierige und mühsame Phasen der Fehlersuche und -verbesserung vermeidet.

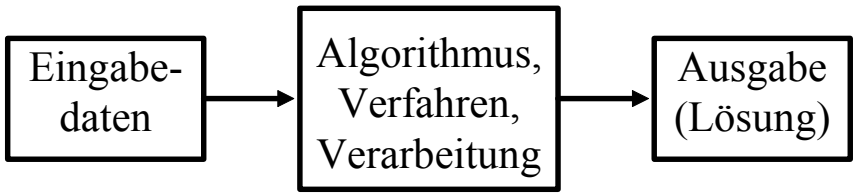


Abbildung: Schematische Darstellung eines Algorithmus (sehr grob):

Ziel höherer Programmiersprachen wie FORTRAN, PASCAL, C, C++, Java:

Der Programmiervorgang ist ein iterativer Vorgang bei dem man eine Vorgehensweise nach formalen Aspekten aufschreiben muß. Der Entwickler kann also umso besser Programme entwickeln je komfortabler die Programmiersprache und die Programmierwerkzeug sind. Die höheren Programmiersprachen zielen insbesondere auf die folgenden Aspekte:

- Hoher Bedienkomfort und Sicherheit bei der Entwicklung von Programmen
- Verwendung von symbolischen Namen (Variablen) - der Mensch kann sich Namen leichter merken als Speicheradressen
- Entlastung des Programmierers von systemspezifischen, prozessorabhängigen (spez. Befehle, spez. Register, ...) Dingen durch abstrakte, mathematische Beschreibung
- Verwendung von Konstrukten zur Entwicklung komplexer, wiederverwendbarer Programme (Unterprogramme) - wiederverwendbar hat in diesem Zusammenhang mindestens zwei Bedeutungen:
  - das entwickelte Programm soll auf unterschiedlichen Maschinen/Rechner ablauffähig sein und
  - das entwickelte Programmteil sollte von der Konzeption und Kodierung so angelegt sein, dass es dazu geeignet ist, ähnliche Problemstellungen wie die ursprüngliche zu lösen oder aber einfach auf eine ähnliche Problemstellung anpassbar sein

- Konzepte zur sicheren und schnellen Programmentwicklung sollen bereits Bestandteil der Programmiersprache sein (zu mindest teilweise) - Beispiele: vordefinierte Datentypen, Mechanismen zur Beschreibung eigener anwendungsspezifischer Datenstrukturen, Kontrollkonstrukte (möglichst KEINE Sprunganweisungen), Variablennamen müssen vor der Verwendung festgelegt werden und müssen eindeutig sein.
- Es existieren bereits vorgefertigte, komplexe Funktionen, die häufig vorkommende problemorientierte Aufgaben lösen (Programmbibliotheken, Modulbibliotheken).

Programmiersprachen sind streng nach vorgegebenen Regeln (Grammatiken) aufgebaute formale Sprachen, in denen der Programmierer seine Rechenvorschriften ablegt.

Das Programm wird somit in einer maschinenunabhängigen Programmiersprache geschrieben. Der entsprechende Compiler übersetzt das Programm aus einer höheren Programmiersprache in die Assembler- oder Maschinensprache.

Bei einer Interpretersprache werden die Anweisungen bei der Ausführung in die maschinenspezifischen Befehle umgestzt und ausgeführt. Wird eine Anweisung wiederholt ausgeführt, so muß sie auch wiederholt umgesetzt werden.

Warum C?

- Hohe Praxisrelevanz (Unix, ...)
- Es existieren sehr viele moderne Entwicklungen in C
- gute Ausgangsbasis für alle anderen aktuellen Entwicklungen im Bereich der Programmiersprachentwicklungen (C++, Java, C#, ...)
  - sie besitzen ähnliche Konstrukte
- Bei dem Erlernen der Programmiersprache soll der Blick geschärft werden für die Konzepte von Programmiersprachen (Generalisierung, Abstraktion, ...)
- C gilt als einfache (= mit wenigen Programmkonstrukten) und flexible Programmiersprache –
  - VORSICHT:
    Die syntaktische und semantische Korrektheit eines Programmes muß der Programmierer noch mehr als bei Pascal sicherstellen!
- Denken Sie daran, dass ein Programm einfach zu lesen und pflegen sein soll (erweiterbar, wiederverwendbar, wartbar, robust) und daher nicht jede Möglichkeit der Programmiersprache im Sinne eines guten Programmierstils zur Übersichtlichkeit oder Verständlichkeit des Programmcodes beiträgt!

Formalismen zur Beschreibung von Programmiersprachen

Um eine Programmiersprache wie C vollständig und präzise beschreiben zu können, bedient man sich in der Literatur häufig sogenannter Syntaxdiagramme („Eisenbahndiagramme“) oder der Backus-Naur-Form (BNF) bzw. der Erweiterten BNF (EBNF).

Syntaxdiagramme

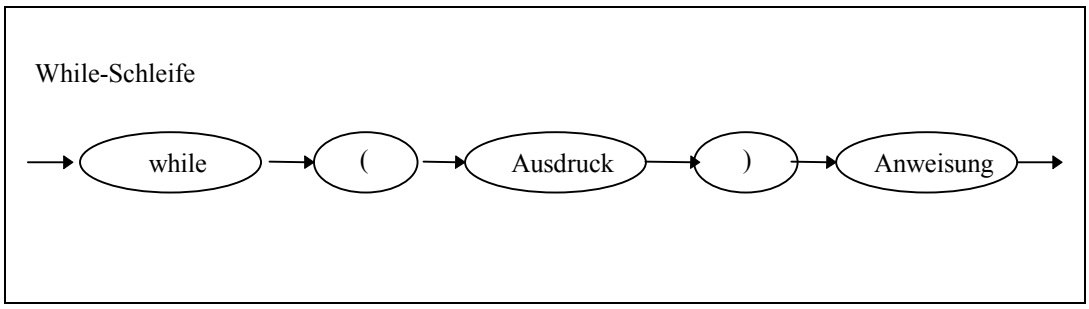
Syntaxdiagramme dienen der Beschreibung einer (formalen) Programmiersprache. Es ist ein Ausdrucksmittel, mit dem man gleichzeitig eindeutig die erlaubte Reihenfolge als auch alle möglichen Reihenfolgen der verschiedenen Elemente einer Programmiersprache beschreibt.

In den sog. Ableitungsregeln werden Sequenzen (Folgen) und Alternativen (Auswahl) beschrieben. Außerdem ist es begrenzt möglich, anzugeben, wie oft Elemente auftreten; man kann beschreiben, ob ein Element an einer bestimmten Stelle (zwingend) vorkommen muß, es optional ist (vorkommen kann aber nicht vorkommen muß) oder beliebig oft vorkommt. Diese Möglichkeiten können beliebig miteinander kombiniert werden.

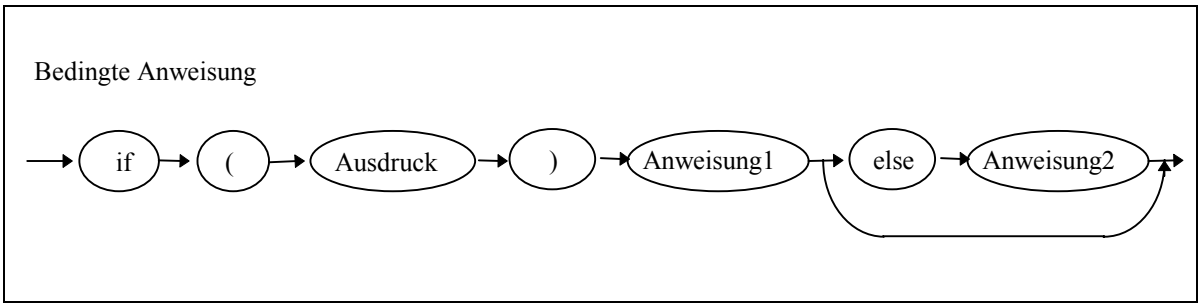
Dabei bezeichnet man die kleinsten (auf eine bestimmte Art nicht weiter teilbare) Teile einer Programmiersprache als Token oder Terminale. Beispiele für die Token sind alle vordefinierten Schlüsselworte in einer Programmiersprache (z.B. while, for, int, if, =, &&, {, }, [,], ...) oder die Variablen. Sie können nicht weiter zerlegt oder abgeleitet werden.

Anders ist es bei den Nonterminalen. Damit wird alles andere bezeichnet, was als komplexes Gebilde benannt werden kann aber nach Regeln (der Syntax der Programmierspache) weiter zerlegt werden kann (z.B. eine Anweisung, alle Teile einer while-Schleife, Unterprogramme, ...). Nonterminale stehen damit immer am Anfang einer Syntaxregel.

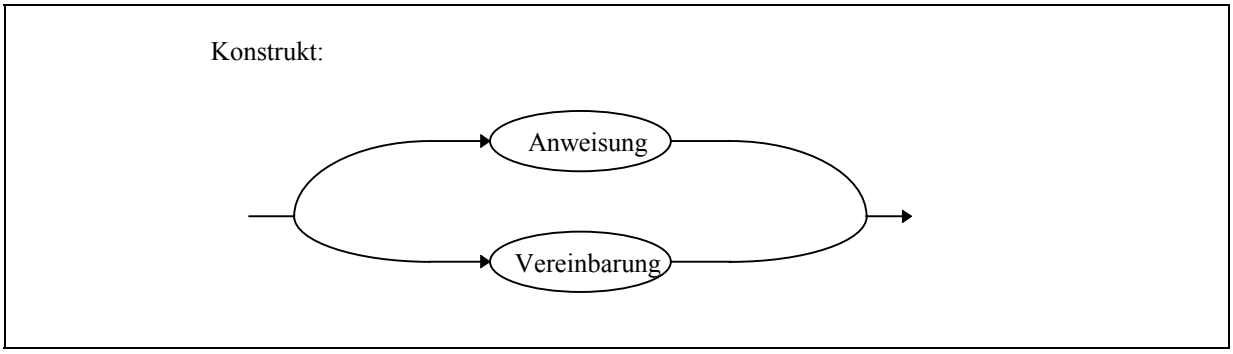
Sequenz:



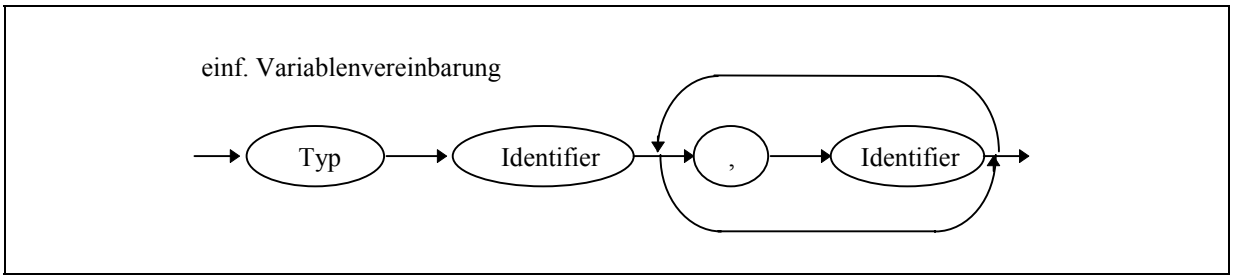
Optionale Elemente:



Alternativen:



Beliebige Wiederholung



Backus-Naur-Form (BNF)

Mit Hilfe dieser Metasprache kann man die Syntax einer Programmiersprache als Ableitungsregeln beschreiben. Im folgenden werden die Elemente vorgestellt:

< >

Innerhalb dieser Klammersymbole steht der Name einer Gruppe von Sprachelementen. Dieser Name taucht dann in einer anderen Ableitungsregel auf der linken Seite auf und wird weiter aufgelöst (Bsp.: <Anweisung>). Falls in einer Syntaxbeschreibung mehrmals das gleiche Sprachelement vorkommt und unterschieden werden muß, wird der Name zusätzlich mit einem Namen versehen (Bsp.: <Anweisung1>, <Anweisung2>).

::=

Das links von diesem Definitionszeichen stehende Sprachelement wird durch den rechts davon stehenden syntaktischen Ausdruck definiert.

|

Die vor und nach diesem Zeichen stehenden Sprachelemente schließen sich gegenseitig aus. Es werden entweder die vor dem Zeichen stehenden Elemente oder nach diesem Zeichen stehenden Elemente verwendet.

Ø

Das Leere Zeichen

Alle übrigen Zeichen und Worte, die außerhalb der spitzen Klammern stehen sind Schlüsselworte (Endsymbole) und müssen stets angegeben werden.

In der erweiterten BNF können zusätzlich weitere Elemente vorkommen:

[]

In eckige Klammern eingeschlossene Sprachelemente sind optional.

{ }

In geschweifte Klammern eingeschlossene Elemente können wiederholt vorkommen.

*****

Mit * versehene geschweifte Klammern können beliebig oft vorkommen (auch keinmal).

+

Mit + versehene geschweifte Klammern müssen mindestens einmal vorkommen.

 Aufgabe:

Geben Sie die Regel an, wie man syntaktisch korrekt eine beliebige Dezimalzahl mit diesem Formalismus beschreibt.

Daten und Variablen

Konzept der Variablen

Eine der grundlegendsten Konzepte höherer Programmiersprachen sind Variablen. Variablen haben:

- eine eindeutige Bezeichnung (Identifier),
- einen eindeutigen Typ (Wertebereich) und
- zur Laufzeit einen Ort im Speicher.

Im Speicher steht der Wert der Variable, der hoffentlich definiert ist.

Dies wollen wir uns noch einmal verdeutlichen:

Der Ort ist im Rechner eine adressierbare Speicherzelle, der Wert wird durch die darin enthaltene Bitfolge repräsentiert (Die Bedeutung der Bitfolge ergibt sich durch die Festlegung des Typs der Variable!). Die Kombination aus symbolischem Namen und dem Paar Adresse und Inhalt - also Ort und Wert - wird als Variable bezeichnet.

Beispiel:

Die Zuweisung:

```
meineZahl = 5;
```

bedeutet, dass im Speicher an dem der Variablen „meineZahl“ zugeordneten Speicherplatz, der beispielsweise die Adresse 45110, besitzen könnte, der Wert 5 als Bitmuster 00...0101 eingetragen wird.

Dieser Zugriff gilt sowohl für die einfachen Grundtypen **int, float** und **char** als auch für die strukturierten Typen (etwa **Felder - eine Zusammenfassung von Datenobjekten des gleichen Typs -**).

Einzelne Elemente einer entsprechenden Feld-Variablen können über einen Index oder mehrere Indices angesprochen werden.

Jede Variable muß vor ihrem Gebrauch deklariert (vereinbart) werden, und sie sollte initialisiert (mit einem Startwert versehen) werden. Die Variablendeklaration prüft der Compiler und gibt, falls eine Variable nicht deklariert ist, dies als Fehler an.

Möglicherweise folgenschwerer für die Programmtests sind versäumte Initialisierungen von Variablen, die der Compiler nicht erkennt. Die Variablen sind (je nach Compiler) in einem undefinierten Zustand!.

Grunddatentypen in C

In C kennt man drei Grunddatentypen:

- int (ganze Zahlen),
- float (Gleitpunktzahlen)
- char (Zeichen).

Beispiel:

```
int i, j, k;
float f;
char s;
i = 1;
f = 1.0;
char = '1';
```

Bei den Zahlen gibt es zusätzlich folgende Varianten bezüglich der Rechengenauigkeit:

short [int]	ganze Zahlen, im Allg. 16 Bit lang
long [int]	ganze Zahlen, im Allg. 32 Bit lang
unsigned [int]	das Vorzeichen Bit gehört zum Wert,
unsigned short	d.h. es gibt nur positive Zahlen
unsigned long	
double	rationale Zahlen
	(bei double: doppelte Genauigkeit gegenüber float)

Weitere Datentypen

Neben den elementaren Typen in C gibt es zusätzlich die Möglichkeit, abstrakte (= anwendungsspezifische) Typen zu deklarieren (z.B. Temperatur, Luftdruck, Entfernung, ... Position, ...).

Sie stehen im Deklarationsteil nach den Konstanten- und vor den Variablendeklarationen und beginnt mit dem Schlüsselwort **typedef**.

Variablen mit konstantem Wert

Im Deklarationsteil in C können konstante Zeichenfolgen oder Zahlen durch eine Variable mit konstantem Wert (constant identifier) bezeichnet werden. Wie bei einer „normalen“ Variable muss der Name eindeutig sein. Im Gegensatz zu Variablen darf bzw. kann der Wert einer Konstanten nicht verändert werden.

Konstanten

Konstanten und logische Datentypen gibt es zunächst nicht. Sie können allerdings sehr einfach festgelegt werden.

Mit Hilfe der Präcompileranweisung **#define** können beliebige Literale (Werte) mit einem Symbol bezeichnet werden.

C könnte man somit folgendermaßen um einen logischen Datentyp erweitern:

```
#define FALSE      0
#define TRUE       !FALSE

int logical, a, b;
...
logical = TRUE;
a = 5;
b = 3;
...
if (logical && a == b)
   printf (“Aussage ist wahr“);
```

Bemerkung: In C gilt der Zahlenwert **0 als logisch falsch**, jeder Wert **ungleich 0 als logisch wahr**.

Die Negation (!) entspricht dem Test des Ausdruck auf 0
(== 0)

Feld-/Array-Definitionen

Neben einzelnen Variablen können auch Felder (Arrays) von einzelnen Elementen des gleichen Typs definiert werden.

Die Länge und Anzahl der Dimensionen wird bei der Deklaration der Variablen in eckigen Klammern angegeben und an den Bezeichner (Identifier) gehängt.

Für jede Dimension gibt es ein eigenes Klammerpaar. Die Dimensionslänge darf nur als Konstante oder konstanter Ausdruck angegeben werden, der zum

Zeitpunkt des Übersetzens ausgewertet werden kann. Bei einem Zugriff auf eine Feldvariable muß deren Dimension bekannt sein!

```
char farbe[10], brett[8][8];
```

Typfestlegung und Reservierung von Speicherbereich, dessen Inhalt **nicht** definiert ist!

Beispiel farbe[10]:

Index	0	1	2	3	4	5	6	7	8	9
Wert	?	?	?	?	?	?	?	?	?	?

Wurde ein Feld der Größe n angelegt, so werden die einzelnen Elemente unter Verwendung eines Index in einer eckigen Klammer angesprochen.

Beim Zugriff sind als Indexangabe beliebige arithmetische Ausdrücke erlaubt.

In C kann man den Index auch als Distanzangabe eines Elementes zum Beginn des Arrays auffassen.

Besonders häufig werden eindimensionale Zeichenfelder (Strings) benötigt.

Operatoren in C

Operator	Bedeutung	Verarbeitungsreihenfolge

`*`
`/`
`%`
	Multiplikation
	Division
	Rest bei Ganzzahldivision
		Links -> Rechts

`+` `-`	Addition Subtraktion	L -> R
`<<` `>>`	Linksshift Rechtsshift	L -> R

`<`
`<=`
`>`
`>=`
	kleiner
	kleiner oder gleich
	größer
	größer oder gleich
		L -> R

`==` `!=`	gleich ungleich	L -> R
`&&`	logische UND	L -> R
`||`	logische ODER	L -> R
`!`	Logische Negation	L -> R
`=, +=, ...`	Zuweisungen	R -> L
`? :`	Bedingter Ausdruck	R -> L

Typumwandlung (allgemein):

- short, char, Aufzähltypen können immer an Stelle von int benutzt werden
- es wird auf den nächst größeren Datentyp ausgeweitet:
  - float -> double -> long double
  - signed -> unsigned, int -> long
- abschließend erfolgt eine Typanpassung an den Typ auf der linken Seite

Welchen Wert hat b nach der Zuweisung?

```
int b;
b = 2 / 3 +  0.333;
```

Typumwandlung (explizit)

Die Datentypumwandlung oder -konvertierung (casting) bzw. der cast-Operator enthält in runden Klammern eine skalare Typbezeichnung ohne Speicherklasse und geht dem Operanden unmittelbar voraus.

Beispiel:

```
int n = 5;
double y;

y = sqrt ( (double)n );
```

n ist eine integer-Zahl. Die Funktion sqrt( ) erwartet jedoch als Argument eine Gleitpunktzahl doppelter Genauigkeit. Die Typumwandlung von n wird durch den cast-Operator **(double)** bewirkt.

Die Typumwandlung im Zusammenhang mit dem cast-Operator hat in der Folge für die Verwendung der Variable keine Bedeutung.

Es liegt in der Sorgfaltspflicht des Programmierers sicherzustellen, dass die Argumente bei Funktionen und Prozeduren vom richtigen Typ sind (Welche Möglichkeiten hat er oder sie?).Nicht alle Compiler geben Hinweise auf falsche Typen in Form von Warnung an.

Anweisungen

Wie bereits bei den Struktogrammen gesehen, gibt es prinzipiell nur eine Handvoll Anweisungen, die allerdings in einer Programmiersprache beliebig kombiniert werden können:

- Zuweisungen/Inkremente/Ein-Ausgabe-Anweisungen
- Bedingte Zuweisungen
- Schleife (for-, abweisende, nicht abweisende Schleife)
- Fallunterscheidungen
- Unterprogramme zu Strukturierung

Wir werden uns nun die Anweisungen im einzelnen genauer ansehen und dabei immer darauf achten, daß wir beabsichtigen, strukturierte Programme zu erstellen und somit zumindest nicht am Anfang jeden Trick und Kniff, die in C möglich sind und die es in C zahlreich gibt, ausprobieren und vorstellen wollen.

Zuweisung (Assignment):

```
<variable> = <expr>;
```

Die Variable erhält den Wert des Ausdruckes expr

Folge von Anweisungen (Sequenz):

```
{
[<definitions>]
<statement1>
<statement2>
<statement3>
...
<statement n>
}
```

Bedingung:

```
if ( <condition> )
  <statement1>
[ else <statement2> ]
```

Bedingte Ausführung von Anweisungen bei Unterscheidung mehrerer Fälle:

```
if ( <condition1> )
  <statement1>
else if ( <condition2> )
  <statement2>
...
else if ( <condition n> )
  <statement n>
[ else <statement n+1> ]
```

Fallunterscheidung

```
switch (<expr>)
{
  case <const term1> : <statement1> [break;]
  case <const term2> : <statement2> [break;]
...
  case <const term n> : <statement n> [break;]
  [default          <statement def> [break;] ]
}
```

Leere Anweisung

```
;
```

In C gehört das Semikolon zur Anweisung!

Schleifen

1. while Schleife

Test der Bedingung vor Schleifendurchlauf

```
while ( <expr> )
  <statement>
```

2. do-while Schleife

Test der Bedingung nach Schleifendurchlauf

```
do
  <statement>
while ( <expr> );
```

3. for Schleife

Test der Bedingung vor Schleifendurchlauf

```
for ( [<expr1>]; [<expr2>];[<expr3>])
<statement>
```

expr1 wird einmal vor dem 1. Schleifendurchlauf ausgewertet (= Initialisierung),

expr2 wird zu Beginn jedes neuen Schleifendurchlaufes ausgewertet,

expr3 wird an Ende eines jeden Schleifendurchlaufes ausgewertet.

Programmein- und Ausgabe: Dateien, Dateihandling

Alle Daten und Programme werden im Rechner auf sogenannten Dateien gespeichert. In einem Verzeichnis (directory) werden Informationen über die in ihm enthaltenen Dateien geführt: Name, Länge, Erzeugungsdatum sowie Datum der letzten Änderung (und ggf. Besitzer). Dateiverzeichnisse können hierarchisch angeordnet sein. Dann gibt es immer ein sogenanntes Wurzelverzeichnis. In jedem Verzeichnis können Dateien und weitere Verzeichnisse angelegt sein.

Über Ein- und Ausgabemöglichkeiten kann ein Programm lesend oder schreibend auf solche Dateien bzw. auf die darin enthaltenen Daten zugreifen.

Dadurch können

- umfangreichere Eingaben an Programme vorbereitet,
- Berechnungsergebnisse dokumentiert,
- Ausgaben zum Zwischenspeichern (d.h.) späteren Wiedereinlesen erzeugt,
- Protokolle (Zwischenschritte) abgelegt,
- Ergebnisse an andere Programme (Austausch von Daten) weitergegeben werden oder
- Programme in gewissem Rahmen konfigurierbar gehalten werden..

Die Art und Weise, wie die Dateien angelegt und die Daten geschrieben werden (d.h. das Format), hängt in aller Regel davon ab, wie die Daten weiter genutzt werden sollen.

Prinzipiell ist beim Umgang mit Dateien folgender Vorgehensweise einzuhalten:

- Dateien können zum Lesen **oder** Schreiben geöffnet werden.
  Es ist auch möglich, gleichzeitig eine Datei zum Lesen und Schreiben zu öffnen; dies sollten wir allerdings am Anfang vermeiden
  Datei variable sind vom Typ FILE (Bsp: FILE *eingabe).
  Dazu muß eine Verbindung zwischen dem Programm und der Datei hergestellt werden (Befehl: fopen). Der Schreib- bzw. Lesekopf wird positioniert.
- Mit jeder Lese- bzw. Schreibanweisung (fscanf, fprintf) wird angegeben, auf welche Datei zugegriffen werden soll und welche Daten (Art und Reihenfolge) gelesen oder geschrieben werden sollen. In eine Art Schablone werden die Werte der Variablen eingesetzt bzw. von dort in den Variablen abgelegt. Der Lese- bzw. Schreibkopf wird unmittelbar danach weiter geschoben. Der Zugriff auf die Datei erfolgt immer an der aktuellen Position des Lese-/Schreibkopfes.

- Am Ende der Eingabe/Ausgabe sollte die Verbindung zwischen Datei und Programm wieder beendet werden und die Datei geschlossen werden (fclose).

Zugriff auf Datei

Für die Kommunikation mit Dateien gibt es in C den Typ **FILE.** Prinzipiell hat die Kommunikation mit einer Datei folgenden Aufbau:

```
FILE *fp;    /* Vereinbarung einer Variable,
                die auf eine Datei zeigt */
fp = fopen (name, mode);
        /* Zuweisung erfolgt an Variable fp
        Datei mit dem Namen 'name'; diese
Datei kann
gelesen werden,        wenn mode = "r",
geschrieben werden,    wenn mode = "w" oder
erweitert werden,      wenn mode = "a";
        u.U. wird noch zwischen Text- und
        Binärdateien unterschieden. */
...
/* Formatierte Eingabe */
int fscanf (FILE *fp, char *format, ...);
/* Formatierte Ausgabe */
int fprintf (FILE *fp, char *format, ...);

feof ( FILE *fp)
/*liefert Wert != 0, wenn das Dateiende
erreicht ist*/
...
fclose (fp); /* Datei schließen */
/* Zusätzlich lesen von einem String:*/
int sscanf (char *str, char *format, ... );
```

Formatierte Ausgabe

```
#include <stdio.h>
...
int printf  (char *format, arg1, arg2, ...);
```

printf gibt nach standard-Ausgabe aus und formatiert die angegebenen Argumente gemäß **format**.

In dem format-Steuerstring ist '%' das Steuerzeichen; ein weiteres Kennzeichen folgt, das den Typ des Argumentes angibt bzw. Angabe über die Formatierung macht.

Es kann zusätzlich mit Angaben über die zu reservierenden Stellen und die Ausrichtung kombiniert werden kann:

printf (format, "hello world");

mit format	liefert
		123456789012345	
"%10s"	|hello world|
"%.10s"	|hello worl|
"%-10s"	|hello world|
"%.15s"	|hello world____|
"%15.10s"	|____hello worl_|
"%-15.10s"	|hello worl_____|

Formatelemente

Für die unterschiedlichen Argumenttypen gibt es unterschiedliche Umsetzungszeichen:

Zeichen	Argument
d,i	int; alsDezimalzahl
o	int; als Oktalzahl
x,X	int; als hexadezimale Zahl
u	int; als vorzeichenlose Dezimalzahl
c	character, int; als einzelnes Zeichen
s	char *; als Zeichenkette
f	double; als Gleitpunktzahl in der Art: [-]m.dddddd
e, E	double; [-]m.dddddde[-/+]xx
g,G	double; Ausgabe wie bei f oder e in Abhängigkeit des Zahlenwertes
%	es wird ein '%' ausgegeben

Zusätzlich können zwischen '%' und dem Kennbuchstaben noch weitere Werte angegeben werden:

- Ein Minuszeichen für Linksbündigkeit
- ein Pluszeichen für Vorzeichenausgabe, Leerzeichen bedeutet Leerzeichen bei positiven Werten
- eine Zahl für minimale Feldlänge und bei s, f, g oder G ein Trennpunkt und ggf. die Anzahl der anzuzeigenden Nachkommastellen

Die Angabe von '\n' bewirkt einen Zeilenumbruch.

Formatierte Eingabe

fscanf ist das Gegenteil zu fprintf. Die Funktion bewirkt eine formatierte Eingabe von der angegebenen Datei (wenn sie zum Lesen geöffnet wurde).

Der zweite Parameter ist wieder ein Steuerstring. Er wird von links nach rechts bearbeitet. Blanks, Tabulatoren- und Zeilenvorschubsteuerung werden genauso wie in der Eingabe ignoriert.

Die Anzahl und der Typ der nachfolgenden Parameter muß genau der Anzahl der %-Zeichen mit Kennzeichen entsprechen.

Von fscanf werden keine Konvertierungen vorgenommen.

Wichtig:

> Das Verständnis in C ist, daß beim Einlesen die Adresse angeben wird, wo der Wert abgelegt wird!

fprintf, fscanf, sscanf, printf und scanf liefern einen integer Wert zurück. Mit Hilfe dieses Rückgabewertes kann vom Anwendungsprogramm aus kontrolliert werden, wieviele Parameter korrekt geschrieben bzw. gelesen wurden.

Das Anwendungsprogramm sollte diese Möglichkeit immer nutzen.

Zeichen und Strings in C

Wie bereits beschrieben kennt C den Grunddatentyp char (= einzelnes Zeichen). Die Angabe eines char in einem vergleichenden Ausdruck oder in einer Zuweisung erfolgt zwischen einfachen Hochkommata.

Beispiel:

```
char c;
c = '*';
```

Eine besondere Rolle spielen wegen Ihrer häufigen Verwendung und wegen des reichen auf sie anwendbaren Funktionssatzes die char-Arrays oder Strings (Zeichenketten). Im engeren Sinn versteht man in C unter einem String eine Zeichenkette, die von dem Grenzzeichen '\0' abgeschlossen wird (wichtig: '\0' benötigt zusätzlichen Speicherplatz der Größe eines Zeichens, '\0' wird allerdings nicht bei den entsprechenden Operationen mitgezählt, z.B. strlen - einer Standardfunktion zur Berechnung der Stringlänge).

Beispiele für Strings:

```
char str[20], *name = "Fachhochschule Bingen";
strcpy (str, "Elektrotechnik");
       /* Stringzuweisung */
```

Operationen auf Zeichenfolgen/Strings

s hat den Wert "Fachhochschule ", t hat den Wert " Bingen":

strcat (s, t) — hängt t an das Ende von s an:
=> „Fachhochschule Bingen"

strncat(s,t,n) — hängt n Zeichen von t an s an (für n = 1):
=> „Fachhochschule B"

strcmp(s,t) — liefert
- negativ, wenn s < t ist
- 0, wenn s == t ist und
- positiv, wenn s > t

strncmp(s,t,n) — wie strcmp, aber nur für die ersten n

	Zeichen
strcpy(s,t)	kopiert t nach s
strncpy(s,t,n)	wie strcpy aber höchstens n Zeichen
strlen(s)	liefert Länge von s ohne ‘\0’ am Schluß
strchr(s,c)	liefert Zeiger (char *) auf erstes c in s oder NULL, wenn c nicht in s
strrchr(s,c)	liefert Zeiger (char *) auf letztes c in s oder NULL, wenn c nicht in s

Vergessen Sie nicht die Datei ‘string.h’ zu inkludieren.

Schrittweise Programmentwicklung

Mit zunehmender Komplexität (Umfang und Schwierigkeit) der gestellten Aufgabe ist es wichtig, dass wir noch einmal zusammenfassen, was wir bereits jetzt in den Aufgaben immer wieder festgestellt haben:

- Läßt sich eine Aufgabe durch einen Algorithmus lösen, so gibt es im allgemeinen mehr als einen Lösungsalgorithmus.
- Ein Algorithmus kann auf verschiedene Weisen beschrieben werden (umgangssprachlich, Pseudocode, mathematischer Formalismus, Struktogramme oder in einer Programmiersprache)
- Die verschiedenen Beschreibungen eines Algorithmus unterscheiden sich in den Voraussetzungen, die derjenige erfüllen muß, der eine solche Beschreibung verstehen und danach verfahren soll
- Bei der Beschreibung eines Algorithmus können wir daran denken, das Problem bzw. die Problemlösung so zu beschreiben, dass wir die Beschreibung entwickeln und immer weiter (d.h. schrittweise) verfeinern
- Dabei ist ganz besonders wichtig, dass, wenn wir einen Algorithmus beschrieben haben, wir diesen Algorithmus auf seine Richtigkeit kontrollieren (**verifizieren**) und mit Hilfe von Tests feststellen, ob er in allen Fällen genau das leistet, was vorgegeben wurde; dies geschieht, nachdem der Algorithmus als Programm implementiert worden ist

Checkliste für Algorithmus-/Programmentwicklung

Bemerkung: Die nachfolgende Check-Liste ist kein starres unabänderliches Schema. Sie soll nur das Finden eines Lösungsansatzes unterstützen. Insbesondere ist die Anzahl der Verfeinerungen nicht festgelegt. Auch ist durchaus damit zu rechnen, dass eine Aufgabenstellung modifiziert wird oder in einem anderen Zusammenhang wiederverwendet wird.

Problemstellung erkannt?

- Haben Sie das Problem erkannt oder müssen weitere Informationen über das Problem besorgt werden?
- Kann das Problem als Aufgabe präzise formuliert werden oder muß die Aufgabenstellung konkretisiert werden, z.B. durch Definition von Einschränkungen?
- Beschreiben Sie, was bekannt und was gesucht ist. Wie sieht die Eingabe aus? Welche Ausgabe wird erwartet?
- Legen Sie die Bedingungen fest, unter denen das Verfahren korrekt arbeitet oder die erfüllt sein müssen.

Formulierung der Spezifikation

- Wie muß die Ausgabe beschrieben werden, wieviel muß gesagt werden, damit die Aufgabe unmißverständlich erfasst werden kann? Welche Details gehören nicht zur Aufgaben-stellung und würden die Entwicklung nur stören?
- Ist die Spezifikation vollständig - deckt sie alle Fälle ab -? Ist sie ohne Widersprüche?

Entwurf des Algorithmus

- Ist dieselbe oder eine ähnliche/vergleichbare Spezifikation bereits formuliert und ihre Lösung durch einen Algorithmus beschrieben worden? Wenn ja, dann informiere man sich über dieses Verfahren
- Ist eine allgemeinere Spezifikation bekannt? Wenn ja, dann sind Informationen über diesen Algorithmus bzw. diese Lösung der Spezifikation zu besorgen. Es ist zu prüfen, ob die Aufgabe als Sonderfall des allgemeinen Falls behandelt werden kann. Ist dies möglich, dann kann der allgemeine Algorithmus angewendet werden (**man muß das Rad nicht immer wieder neuerfinden!**)

- Läßt sich die Aufgabe in einfachere Teilaufgaben aufteilen?
  Wenn dies möglich ist, dann gehe man nach diesem Schema auch bei den Teilaufgaben vor. Der gesamte Algorithmus kann so nach und nach entwickelt werden.
- Man stelle einen in Einzelschritten gegliederten Algorithmus auf. Sind Teilaufgaben zu lösen, dann kann man für jede Teilaufgabe einen Schritt nehmen.

Korrektheit des Algorithmus (Erfüllt der Algorithmus die Spezifikation?)

- Man übezeuge sich von der Korrektheit des Algorithmus und seiner Beschreibung durch
  - theoretische Überlegungen und experimentelle Tests
    insbesondere von Bedingungen, Schleifenabbruchkriterien, Zugriff auf Feldelemente, Übergabe von Werten, ...
- Ist der Algorithmus nicht korrekt, dann ist er solange zu verbessern, bis er korrekt ist. U.U. muß man bis zu einer

  Neuformulierung der Problemstellung gehen. Dies zeigt allerdings auch, dass die ursprüngliche Formulierung nicht ausreichend war.

Ist eine Verbesserung des Verfahrens möglich?

- Führen Sie Effizienzbetrachtungn durch. Können Sie das Verfahren so verbessern, dass das Problem unverändert ge-löst wird aber möglicherweise in weniger (Lauf-)Zeit (also in weniger Programmschritten) oder mit weniger Speicherplatz. In aller Regel ist abzuwägen, ob man Speicherverhalten oder Laufzeit optimieren möchte.

Kodierung und Test

- Testen Sie nach der Kodierung das Verfahren mit Datensätzen, die die unterschiedlichen (möglichst alle) Programmsituationen durchlaufen.

Strukturiertes Programmieren - Ziele

- Wartbarkeit und Wiederverwendbarkeit

- Einfache Anpassung an sich verändernde Randbedingungen (Konfigurieren statt Neuprogrammieren) - Entlastung des Programmierers und Beschleunigung der Tests und der Entwicklung

- Portierbarkeit - Programm läuft auch auf Rechnern, auf denen das Programm nicht entwickelt und getestet wurde

- Skalierbarkeit - durch einfache Modifikationen (z.B. Veränderung der Konstantenwerte) kann der Algorithmus auch auf andere/ähnliche Probleme erweitert werden

- Integrationsfähigkeit mit anderen Komponenten
  Das Programm kann einfach mit anderen Programmteilen, die separat entwickelt wurden, kombiniert werden.

Methoden, Techniken

- Schrittweises Verfeinern (Top-Down, Bottom-Up)
- Programmiersprachenunabhängige Beschreibung des Verfahrens (umgangssprachlich, Pseudocode, Struktogramme, mit Hilfe von mathematischen Formeln)
- Verwendung von Kontrollkonstrukte (Schleifen, Bedingungen, Fallunterscheidungen)
- Vermeidung von Literalen (Zahlen- und Zeichenwerten) im Programm soweit möglich
- Einrückung des Programms (Quellcodes) gemäß der Kontrollkonstrukte (Verbesserung der Lesbarkeit und Prüfbarkeit)
- Verwendung von Kommentaren zur Erläuterung der wesentlichen Schritte
- Formulierung von Vor- und Nachbedingungen an Startwerte und Ergebnisse, die weitergereicht werden.
- Schnittstellen Spezifikation - Wie soll sich ein Algorithmus nach aus verhalten?
- Information Hiding-Verbergen von Implementierungsdetails
- Systemunabhängigkeit - Verwendung der **definierten** Sprachkonstrukte
- Modularisierung
  lokale Änderungen haben lokale Auswirkungen

Unterprogramme

Eine Befehlsfolge in einem Programm kann in der gleichen Form an verschiedenen Stellen vorkommen, wobei sich nur die Anfangsdaten ändern. Man kann nun diese Befehlsfolge aus dem Programm ausgliedern und einmal

separat beschreiben. Dadurch wird Speicherplatz gespart und die Übersichtlichkeit des Programms verbessert.

Neben der Einbindung in den Programmablauf muß die Übergabe der Eingangsdaten (**Parameter**), die im Unterprogramm benötigt werden, und die Rückgabe des Ergebnisses.

Den Einsatz von Unterprogrammen kann man unter zwei Gesichtspunkten betrachten:

- Arbeitserleichterung
- Strukturierung

Zunächst liefert es Erleichterung bei der Kodierung: häufig benötigte Sequenzen werden einmal implementiert und bel. oft verwendet (reduziert Fehlerquellen und Implementierungs-aufwand). Sie erhöhen die Wiederverwendung von Programmteilen.

Der zweite Aspekt ist allerdings nicht minder wichtig und kann gerade bei komplexen Aufgabe nicht hoch genug eingeschätzt werden:

Die Verwendung von Unterprogrammen ist neben der Beschreibung und Verwendung von abstrakten Datenstrukturen eine wesentliche Komponente der

modularen Programmentwicklung. Mit den Datenobjekten zusammen bilden die Unterprogramme die konkrete Realisierung in einer gegebenen Programmiersprache.

Sie bilden nach außen zu den anderen Programmteile die Schnittstelle zu der Problemlösung und machen so das Verfahren (in unterschiedlichen Anwendungsfällen) verfügbar.

Ein Modul hat den Vorteil, dass es

- separat entwickelt,
- separat getestet und
- bei Bedarf ausgetauscht werden kann.

Funktionen und **Prozeduren** unterscheiden sich dadurch, dass Funktionen einen Rückgabewert liefern; Prozeduren liefern keinen expliziten Rückgabewert. Insofern können Funktionsaufrufe auch in Ausdrücken (z.B. Zuweisungen oder Schleifenbedingungen) stehen - hierbei besteht allerdings die Gefahr, dass es

Seiteneffekte gibt, d.h. neben der Ermittlung des Rückgabewerte werden u.U. noch andere Variable verändert oder Aktionen ausgeführt, die den Programmzustand verändern.

Prozeduren werden beim Aufruf wie separate Anweisungen behandelt.

Sowohl Funktionen als auch Prozeduren müssen in aller Regel beim Aufruf mit Werten versorgt werden.

Beim Aufruf müssen also die **Anzahl**, die **Typen** und auch die **Reihenfolge** der aktuellen Parameter **mit** denen in der **Deklaration übereinstimmen**. Die Übergabe per Parameter ist „einfach“ und daher einfach nachzuvollziehen und zu kontrollieren (für den Compiler). Sie ist der Übergabe über globale/externe Variable vorzuziehen.

Die Übergabe durch globale Variable sollte nur dort eingesetzt werden, wo mehrere Funktionen die gleichen Datenobjekte verändern und deren Verwendung generell bekannt ist.

Parameter werden durch Komma getrennt und in runde Klammern eingeschlossen an den Funktionsnamen angehängt. Beim Funktionsaufruf gibt man also die aktuellen Parameter an.

Eine Funktions- bzw. Prozedurdeklaration erfolgt in C nach den Konstanten-, Typ- und Variablendeklarationen und fügt sich so in den Gesamtaufbau ein:

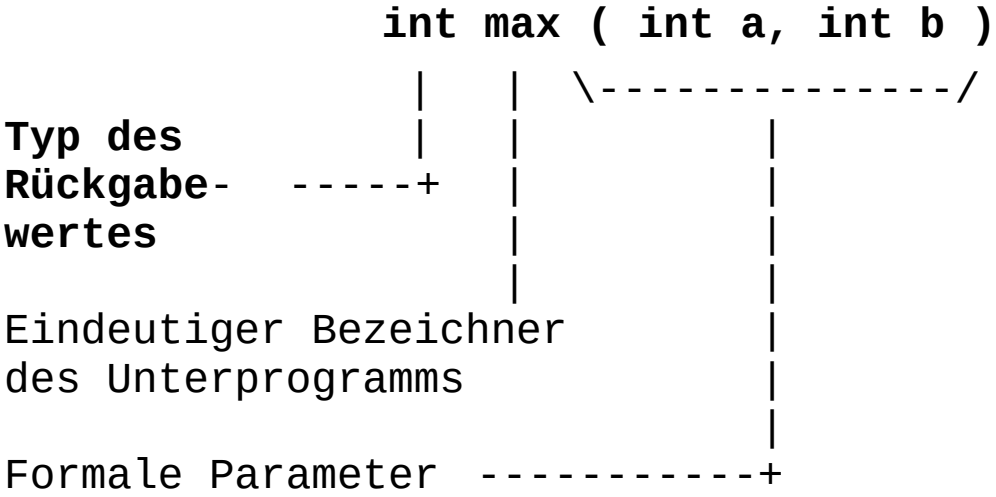
Unterprogramme - Definition und Aufruf

Wir verwenden das zugegebenermaßen einfache Beispiel zur Bestimmung des Maximums von zwei Zahlen, um Funktionen einzuführen:

```
int max ( int a, int b )
  {
  int hilf;
/* Vorbedingung: a und b sind integer-Zahlen */
  if (a < b)
      {
        hilf = a;
        a    = b;
        b    = hilf;
      };
  return a;
/* Nachbedingung: max ist das Maximum von a
   und b; a und b sind ggf. vertauscht, was
   keine Auswirkung im aufrufenden
   Programmstück hat !*/
  }
```

Zur weiteren Verständigung und zur eindeutigen Bezeichnung der einzelnen Komponenten bei Funktionsdefinitionen und -aufrufen werden wir einige Begriffe festlegen und an dem Beispiel noch einmal aufzeigen:

Das erste Schlüsselwort gibt an, um welche Art von Unterprogramm es sich handelt: Funktionen haben einen Rückgabewert (int, float, char, ...) Prozeduren besitzen keinen Rückgabewert (void).



Die formalen Parameter werden in der Funktions-/Prozedurdeklaration verwendet, um zu beschreiben, wie die entsprechenden Variablen bei einem Aufruf gemäß der Rechenvorschrift verwendet werden.

Der Typ des Rückgabewertes ist einer der Grundtypen (int, float, char, ...) oder ein anwendungsspezifischer Typ (wie wir später noch sehen werden).

Werden innerhalb des Unterprogrammblockes Variable (, Konstanten und Typen) definiert, so werden wir diese als lokale Variable bezeichnen. Im Gegensatz dazu werden die Variablen, die im Hauptprogramm oder allgemeiner im aufrufenden Programmteil definiert sind, globale Variablen genannt.

Die Funktion wird wie folgt aufgerufen:

```
c = max ( x, y);
      \-----/
         |
Aktuelle Parameter
```

Die aktuellen Parameter müssen in Anzahl, Typ und Reihenfolge mit den formalen Parameter übereinstimmen.

Zum Zeitpunkt der Deklaration werden die formalen Parameter also benötigt, um allgemein zu beschreiben, wie die angegebenen Variablen bzw. deren Werte zum Zeitpunkt der Ausführung in die Rechenvorschrift eingehen.

Dabei gilt in C die folgende Regel:

Wird in der Parameterliste ein Wert (Konstante) oder eine Variable angegeben, so werden zum Zeitpunkt des Funktions- bzw. Prozeduraufrufs die Werte des aktuellen Ausdrucks übergeben und Veränderungen zur Ausführung des Unterprogrammes haben **keine** Auswirkung auf die Variablen im Aufruf. Insbesondere können dann auch Ausdrücke angegeben werden.

Wenn in der formalen Parameterliste zusätzlich der Inhaltsoperator (*) angegeben wird, dann haben die Veränderungen an der entsprechenden Variable auch nach Ausführung des Unterprogramms ihre Gültigkeit. Auf diese Art und Weise können mehrere Wert zurückgeliefert werden und auch komplexe Datentypen modifiziert werden. Dabei ist darauf zu achten, dass beim Aufruf nur eine Variable mit dem Adressoperator (&) zusammen angegeben wird, nicht aber ein Ausdruck verwendet werden kann. Diese Art der Parameterübergabe kann man sich auch so vorstellen als ob die Adresse übergeben wird, wo der Wert steht.

Besonders sorgfältig muß man Unterprogramme entwerfen und ihr Verhalten auf die Umgebung in dem aufrufenden Programmstück bedenken. Ob Aufrufparameter modifiziert zurückgegeben werden, kann man dem Aufruf in C ansehen.

Parameterübergabeverfahren

Beim Aufruf einer Funktion/Prozedur erfolgt eine Zuordnung von Variablen, die als Argument verwendet werden und als formale Parameter im Funktions-/Prozedurkörper angesprochen werden.

Wie diese Zuordnung erfolgt ist bei verschiedenen Programmiersprachen teilweise unterschiedlich. In einigen Programmiersprachen (so auch in C) kann die Programmiererin/der Programmierer wählen, welches Verfahren der Zuordnung - Parameterübergabeverfahren - für einen bestimmten Parameter angewendet werden soll.

Es werden hier die beiden Möglichkeiten vorgestellt, die in der Programmiersprache C existieren, um Parameter an ein Unterprogram zu übergeben:

- call-by-value
- call-by-reference

Call-by-Value

Die Zuordnung zwischen einer aktuellen Variable erschöpft sich darin, dass **vor** Beginn der Ausführung des Funktions-/Prozedurkörpers der Wert der aktuellen Variable dem entsprechenden formalen Parameter zugewiesen und damit initialisiert wird.

Veränderungen des Wertes des formalen Parameters während der Ausführung der Funktion/Prozedur zeigen keinerlei Wirkung auf die aktuelle Variable, mit der die Funktion/ Prozedur aufgerufen wurde.

In C wird dieses Verfahren ohne besondere Erklärung der Programmiererin/des Programmierers eingesetzt.

Funktions-/Prozeduraufruf:

Man beachte, dass die Berechnung des Wertes, mit dem ein Parameter versorgt wird, vor Beginn der Ausführung erfolgt. Handelt es sich bei dem Argument um eine Konstante oder Variable, so wird der gespeicherte Wert übertragen, handelt es sich um ein programmiersprachliches Konstrukt zur Berechnung des Wertes (z.B. eines arithmetischen Aus-drucks), so wird dieser Ausdruck ausgewertet und der ermittelte Wert übertragen.

Wirkung nach außen (zum aufrufenden Programmteil):

Bei einem Parameter, der nach dem call-by-value Verfahren übergeben wurde, ist es **nicht** möglich eine Wirkung der Ausführung des Funktions-

/Prozedurkörpers (also eine Wertänderung von der aufgerufenen Variable oder des Ausdruckes) nach außen zu geben.

Vorteile:

- der Parameterwert wird vor Funktions-/Prozedurausführung bestimmt
- das Übergabeverfahren kann durch einige wenige Maschinenbefehle erledigt werden
- Veränderungen des Wertes des Parameter haben keine Auswirkung über die Ausführungszeit der Funktion/Prozedur
- im Funktionskörper kann der formale Parameter wie eine lokale Variable benutzt werden

Nachteile:

- der formale Parameter belegt separaten Speicherplatz; dies ist relevant, wenn der Speicherbedarf groß ist
- der Wert des Parameters wird auf jeden Fall ermittelt, auch wenn er in einem bestimmten Ausführungsfall wegen einer gegebenen Datensituation nicht benötigt wird
- das Kopieren des Wertes kann aufwendig sein
- Resultate können nicht nach außen gelangen

Call-by-Reference

Wenn der Programmierer in C dieses Verfahren anwenden möchte, so muß er dies für den entsprechenden Parameter dadurch kenntlich machen, dass er/sie die Adresse übergibt und vor dem entsprechenden formalen Parameter angibt.

Funktions-/Prozeduraufruf:

Die aktuellen Parameter verschmelzen förmlich mit den formalen Parameter und werden statt ihrer referenziert. Dies ist nur möglich, wenn es sich beim Aufruf um eine Variable handelt.

Wirkung nach außen (zum aufrufenden Programmteil):

Bei einem Parameter, der nach dem call-by-reference Verfahren übergeben wurde, werden sämtliche Veränderungen im aufrufenden Programm nach Ausführung des Unterprogramms sichtbar bleiben.

Vorteile:

- Parameterübergabe erfordert keine zusätzliche Maschinenoperationen

- es wird kein zusätzlicher Speicherplatz benötigt
- Veränderungen können nach außen gelangen
- Wert des aktuellen Ausdrucks für den Parameter wird nur ermittelt, wenn er benötigt wird

Nachteile:

- Zugriff auf das formale Objekt ist u.U. schwieriger, da es keine lokale Variable der Funktion/Prozedur darstellt
- Veränderungen während der Ausführung des Funktions-/Prozedurkörpers wirkt sich nach außen aus, d.h. aktuelles Objekt hat nach der Ausführung möglicherweise einen anderen Wert als vorher; es ist vom Anwendungsfall abhängig, ob solche **Seiteneffekte** erwünscht sind
- Wert des aktuellen Parameters wird ggf. mehrfach ermittelt

Funktionen und Prozeduren

Funktionen sind in C genauso wie in vielen anderen Programmiersprachen ein wichtiges Mittel zur Modularisierung und Vereinfachung/Strukturierung von Programmen.

In C bleibt es allerdings dem aufrufenden Programm überlassen, ob es den Rückgabewert der Funktion auswertet oder andere Effekte eines Funktionsaufrufes ausnutzt. Somit werden mit einem einzigen Konzept in C Prozeduren und Funktionen abdeckt.

Eine Funktion hat folgenden Aufbau:

```
[<Speicherklasse>][<Typ>] <Name>
( [<Formale Parameterliste >] )
{
     [<Definition der lokalen Variablen>]

   <Anweisungen>

}
```

Für Speicherklasse kann „**static**“ stehen, d.h. solche Funktionen können nur lokal von anderen Funktionen **in der gleichen Datei** benutzt werden.

Typ ist der Funktionswert (Rückgabewert) und kann entfallen, wenn es sich um int handelt. Als Typen sind die Grundtypen und Pointer- und Strukturtypen - behandeln wir später - zugelassen.

Wenn der Typ „**void**“ (= leer, frei, unbesetzt) angegeben wird, dann besagt dies, daß kein Wert zurück geliefert wird. Dann sprechen wir von einer Prozedur.

„**Name**“ bezeichnet den Namen der Funktion.

Genauso wie Variable werden auch Funktionen deklariert.

In diesem Fall spricht man von **Prototyping**:

> Bei der Deklaration wird im Gegensatz zur Definition nur die Schnittstelle (Name und Parameter) zu dem Unterprogramm festgelegt/bekannt gegeben ohne die konkrete Implementierung anzugeben. Die Funktionsdeklaration schließt mit einem Semikolon ab.

In C gibt es eine einfache Blockstruktur (keine geschachtelte Funktionen). Alle Funktionen befinden sich auf dem gleichen Niveau und können von allen anderen Unterprogrammen aufgerufen werden.

Sichtbarkeit, Lebensdauer, Gültigkeit von Variablen

Wir haben die folgenden Möglichkeiten kennen gelernt, Variable, Konstante, und Typen in einem Unterprogramm zu verwenden:

- lokale und globale Variable
  lokale Variable überdecken globale Variable mit dem gleichen Namen. Variable aus aufrufenden Programmteilen können benutzt werden und
- formale Parameter

Die Syntax zur Bildung von Programmen und zur Deklaration von Funktionen und Prozeduren erlaubt beliebig viele Schachtelungen von Blöcken (**nicht:** Unterprogramme in Unterprogrammen!). Allerdings sind nur solche Blockschachtelungen sinnvoll und zulässig, bei denen sämtliche verwendeten Namen definiert sind und eine eindeutige Bedeutung haben.

Die innerhalb eines Blockes vereinbarten Namen für Konstanten, Typen und Variablen haben nur innerhalb dieses Blockes Gültigkeit. Der **Gültigkeitsbereich** von Vereinbarungen ergibt sich eindeutig aus der **statischen Blockstruktur**, d.h. aus der Art, wie die Blöcke beim Aufschreiben des Programms ineinander geschachtelt werden.

Dabei gelten in C die folgenden Regeln:

- das in der Aufrufsequenz tiefer gelegene Programmstück überdeckt darunter liegenden Programmteile (bzw. deren Daten). Das Hauptprogramm hat die Aufruftiefe 0.
- Jede Vereinbarung eines Namens hat nur in dem Block Gültigkeit, in dem sie vorgenommen wird, mit der Ausnahme all der inneren Blöcke, die eine Vereinbarung des selben Namens enthalten.
- Standardnamen gelten als in einem den äußersten Block umfassenden fiktiven Block. Sie haben Gültigkeit in dem ganzen Programm, wenn sich nicht in dem Programm selbst oder in einem untergeordneten Block neu vereinbart werden (dies ist möglich, sicher aber kein guter Programmierstil).

Die **Lebensdauer** von (lokalen) Variablen ist auf die Dauer der Ausführung des Unterprogramms beschränkt. Man darf also nicht erwarten, dass die Werte der Variable beim Wiedereintritt in das Unterprogramm noch zur Verfügung stehen.

Verbunddatentypen

Ein Verbund oder eine Struktur in C ist eine Zusammenfassung mehrerer Teile, die unterschiedlichen Typs sein können und in einem bestimmten anwendungsbezogenen Zusammenhang stehen. Ihre Deklaration wird durch das Schlüsselwort "**struct**" eingeleitet.

Gefolgt von dem Schlüsselwort "struct" steht der Name für den in geschweiften Klammern angegebenen Strukturtyp.

Beispiel:

```
struct datum {
              int tag;
              char mon_name[4];
              int jahr;
             }
```

Damit ist ein neuer Typ festgelegt. Variable von diesem Typ werden anschließend verienbart und angelegt:

```
struct datum hochzeit, geb_tag;
```

Der Zugriff erfolgt im Rahmen von Zuweisungen und Bedingungen: <Strukturname>.<Elementname>:

```
geb_tag.jahr = hochzeit.jahr;
```

Iterative und Rekursive Algorithmen

Bereits die bisherigen Probleme haben gezeigt, daß bei der Ausführung eines Algorithmus möglicherweise eine Teilaufgabe gestellt und gelöst wird.

Bei der Beschreibung eines Algorithmus kann es daher vorkommen, dass zur Lösung einer Teilaufgabe ein separater Algorithmus mit einer separaten Spezifikation entwickelt und implementiert wird.

Die Formulierung von Teilaufgaben hilft, das Problem zu zerteilen und gibt damit eine Strukturierung vor.

Anders ausgedrückt:

> Ein Algorithmus kann einen anderen Algorithmus enthalten.

Man kann zwei Arten von Algorithmen unterscheiden:

- Erfordert jede Teilaufgabe die vorhergegangene Lösung aller vorher gestellten Teilaufgaben, so sprechen wir von einem **iterativen Algorithmus**.
- Stimmt eine Teilaufgabe mit der gestellten Aufgabe selbst überein, die der Algorithmus selbst lösen kann (wegen der Forderung nach Terminierung notwendigerweise mit unterschiedlichen Eingabewerten), dann nennt man diesen **rekursiven Algorithmus**.

Bem.:

> In der Mathematik kennt man ein Verfahren, dass man zur Beweisführung einsetzt und von dem Fall n-1 auf n schließt unter dem Namen vollständige Induktion.

Beispiel:
Größter Gemeinsamer Teiler:

Deklaration:

```
int ggt (int  a, int b)
```

Es gelten folgende Bedingungen:

ggt (a, b) = ggt (b, a)

ggt (a, 0) = ggt (0, a) = a

ggt (a, b) = ggt (|a|, |b|)

ggt (a, b) = a, wenn a = b

Lösungsvorschläge:

Primzahlzerlegung → sehr aufwendig!

fortlaufende Division → schwierig zu beschreiben

fortlaufende Subtraktion der jeweils kleineren Zahl

bis beide Zahlen gleich groß sind. Beispiel:

ggt (6, 21)

a	b
6	21
6	15
6	9
6	3
3	3

Führen Sie die Schritte auch mit folgenden Zahlen durch:

ggt (4, 17) , ggt (11, 66), ggt (12, 66), ggt (12, 30)

Die Funktion kann wie folgt implementiert werden:

```
program hallo (input, output);
{
function ggt (a, b : integer) : integer;
begin
 while a <> b do
   begin
     if a > b then a := a - b
     else  b := b - a;
   end;
 ggt := a;
end;
}
/* oder auch:    */
function ggt (a, b : integer) : integer;
begin
 if a = b then ggt := a
 else
    if a > b then ggt := ggt ( a-b, b )
    else ggt := ggt (a, b-a);
end;
begin

writeln (ggt von  4 und 17 ist ', ggt(4, 17):3 );
writeln ('ggt von 12 und 30 ist ', ggt(12, 30):3);
writeln ('ggt von 11 und 66 ist ', ggt(11, 66):3);
writeln ('ggt von 12 und 66 ist ', ggt(12, 66):3);
writeln ('ggt von  6 und 21 ist ', ggt(6, 21):3);

end.
```

Beispiel: Rekursive Algorithmen

Fac (n): Fac (n) = 1 * 2 * 3 * ... (n-1) * n

Fac (0) = 1; /* so festgelegt */

Iterative Lösung:

```
/* da wir Fak als Funktion implementieren und uns auf
die Lösung des Algorithmus zu Berechnung der Fakultät
konzentrieren wollen, geben wir auch 1 zurück, wenn
die Eingabe kleiner 0 ist; alternativ könnte man
darauf auch bei dem Unterprogramm achten */

int fac ( int n )
{
 int i;
 f = 1;
 for (i = 2; i < n; i++) f = f * i;
 return f;
}

Aufruf: i = fac(4);
```

Rekursive Lösung:

Fac (n): Fac (0) = 1; /* so festgelegt */

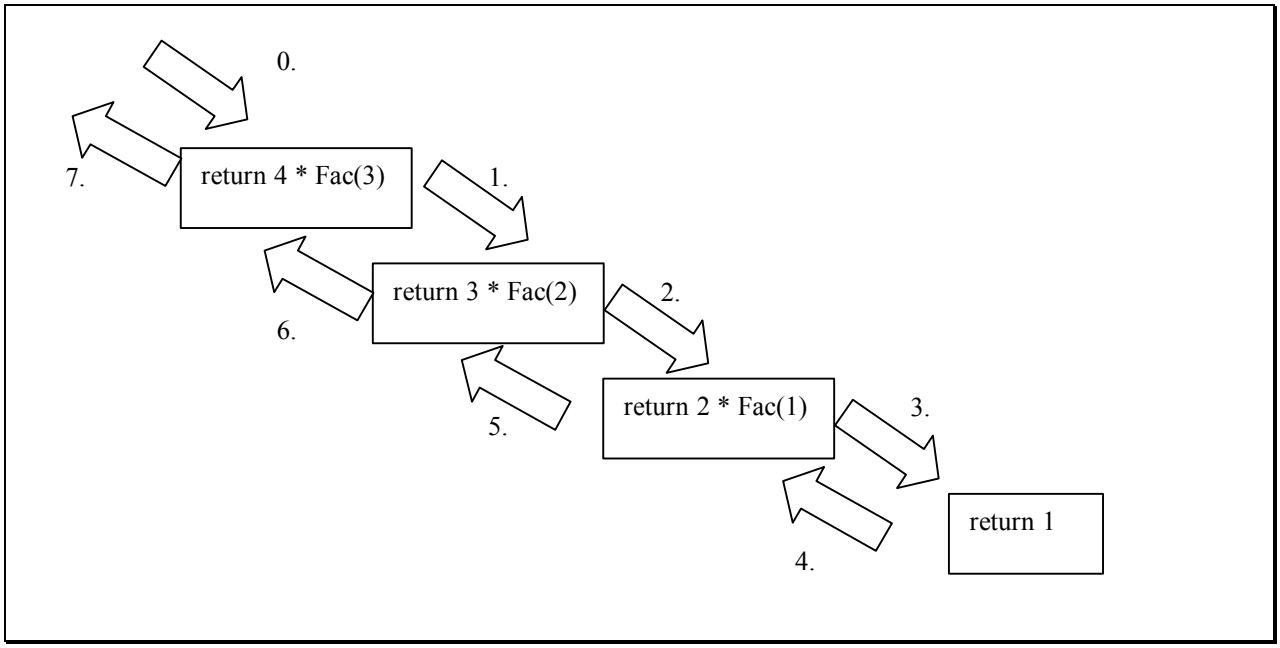
Fac (n) = n * Fac (n-1)

```
int fac ( int n )
{
 if ( n < 2 ) return 1;  /* Terminierung */
 return n * fac ( n-1 );    /* Rekursion */
}

Aufruf: i = fac(4);
```

Der rekursive Ansatz bietet oft die Möglichkeit einer sehr eleganten Fomulierung der Problemlösung: das Verfahren verwendet sich selber mit modifiziertem Datenfall. Gerade bei rekursiven Verfahren ist besonders darauf zu achten, daß neben dem korrekten Ergebnis die **Terminierung** des Algorithmus gewährleistet wird.

Wie ist sicher gestellt, dass der rekursive Algorithmus terminiert (Beispiel: Fac(4))? Die Implementierung muß sicherstellen, daß das Problem auf die Situation zurückgeführt wird, die einfach also nicht mehr rekursiv gelöst werden kann:

Fac (4):



Die Aufrufhierarchie liefert also folgenden Ausdruck:

fak = 4 * 3 * 2 * 1;

In Zukunft werden wir von der Möglichkeit der rekursiven Beschreibung von Verfahren häufiger Gebrauch machen.

Aufgabe:

Geben Sie den rekursiven Algorithmus an zur Bestimmung der größten Zahl aus einem Feld mit ganzen Zahlen

```
a)  Als Vorüberlegung zunächst die iterative
  Lösung:
  void max (int *f, int n, int *m)
  {
   int i;
   *m = f[n-1]; /* Start */
   for (i = n-2; i >= 0; i--)
      if ( f[i] > *m ) *m = f[i];
  }

b)  Die rekursive Lösung
    void max (int *f, int n, int *m)
    {
     if ( f[n-1] > m ) *m = f[n-1];
     if ( n > 1 ) max (f, n-1, m);
    }

Aufruf:    wert = v[n-1];
           max (f, n, &wert);
```

Bem: Die Schnittstelle muß unabhängig davon sein, ob eine rekursive oder iterative Lösung verwendet wird.

Sortier- und Suchverfahren -Berechnung des Aufwandes eines Algorithmus

Aufgabe:

Beschreiben Sie ein Verfahren zum Sortieren eines Feldes, in dem natürliche Zahlen gegeben sind.

Das Sortieren einer Menge von Werten über einem geordneten Wertebereich (z.B. int, real, string), das heißt die Berechnung einer geordneten Folge aus einer ungeordneten Folge dieser Werte, ist ein zentrales algorithmisches Problem. Sortieralgorithmen haben viele direkte Anwendungen in der Praxis oder sind wichtige Teilschritte in Algorithmen, die ganz andere Probleme lösen (Verdeckungen in der Graphik, alphabet. sortierte Einträge in Listen in einer Video- oder Bibliotheksverwaltung, Datenbankanwendungen, Telefonbücher, ...).

Annahme (Vereinfachung o.B.d.A.): Es handelt sich um n Zahlen, die in dem Feld f mit f(0) bis f(n-1) angesprochen werden können.

Es wird so sortiert, daß folgendes gilt:

$$f(i) < f(j), \text{ mit } i < j \text{ und } 0 <= i,j <= n-1$$

Man kann Sortieralgorithmen nach verschiedenen Kriterien klassifizieren:

- **intern/extern**
  Man spricht von einem internen Verfahren, wenn alle zu sortierenden Werte gleichzeitig im Hauptspeicher gehalten werden können. Ein externes Verfahren lädt jeweils nur eine Teilmenge der Datensätze.
- **methodisch:**
  - Sortieren durch Einfügen
  - Sortieren durch Auswählen
  - Divide-and-Conquer
  - Fachverteilen
  - ...
- **nach Effizienz**
  Besonders interessant für die Praxis sind die Verfahren, die das Problem sehr schnell lösen können. Einfache Verfahren haben eine Laufzeit von $O(n^2)$, gute Verfahren erreichen O(n log n). Andere Verfahren unterschieden sich noch in Durchschnitts- und worst-case-Verhalten.

- **im Array oder nicht (Datenstruktur)**
  Bei internen Verfahren ist häufig an Verfahren interessiert, die die Ausgangs- und Zielfolge im Array darstellen. Solche Methoden sind einfach zu implementieren und brauchen keinen zusätzlichen Platz für Zeiger. Besonders beleibt sind Verfahren, die nur einen einzigen Array brauchen und das Ergebnis durch Vertauschen innerhalb dieses Arrays erzielen.
- **allgemeine/eingeschränkte Verfahren**
  Manche Methoden (Sortieren durch Fachverteilen) lassen sich nur eine Folge von Werten mit speziellen Eigenschaften anwenden.

In vielen Sortieralgorithmen, insbesondere beim Sortieren von Arrays, sind die wesentlichen Operationen der **Vergleich** zweier Werte und das **Vertauschen** von zwei Werten. Deshalb werden oft in der Analyse diese Operationen gezählt und als Kostenmaß (= Wert der angibt, wie aufwendig ein bestimmtes Verfahren ist) benutzt.

Bubble-Sort

Ein populäres und suggestives Verfahren ist der Bubble-Sort Algorithmus. Das Verfahren vergleicht bei jedem Durchgang alle benachbarten Elemente und vertauscht sie gegebenenfalls. Dadurch wandert das k-größte Element nach hinten bzw. das k-kleinste Element nach vorne.

Der Name des Algorithmus resultiert aus der Analogie mit dem Aufsteigen einer Gasblase in einer Flüssigkeit.

Das folgende Verfahren sortiert n Zufallszahlen, indem im k-ten Durchgang das k-kleinste Element durch wiederholtes Vertauschen nach vorne gestellt wird:

```
Für alle Werte von i = 1 bis i = n-1 wiederhole
    Für alle Wert von j = n-1 bis j = i wiederhole
       wenn f(j-1) > f(j) dann vertausche (f, j, j-1)

vertausche(f, i,j)
    tmp  = f(i)
    f(i) = f(j)
    f(j) = tmp
```

Als Beispiel betrachten wir das Sortieren von 10 Zahlen. Das jeweils k-kleinste Element ist unterstrichen:

70	40	10	80	50	20	90	60	30	0
<u>0</u>	70	40	10	80	50	20	90	60	30
0	<u>10</u>	70	40	20	80	50	30	90	60
0	10	<u>20</u>	70	40	30	80	50	60	90
0	10	20	<u>30</u>	70	40	50	80	60	90
0	10	20	30	<u>40</u>	70	50	60	80	90
0	10	20	30	40	<u>50</u>	70	60	80	90
0	10	20	30	40	50	<u>60</u>	70	80	90

Quicksort

Der hier angegebene Algorithmus wird **Quicksort** genannt. Er beruht auf der Idee der Partitionierung, deren Umfang nach der Methode des Teile-und-Herrsche, sukzessive und näherungsweise halbiert wird.

Der Quicksortieralgorithmus ist ein typisches Verfahren, das nach dem Prinzip „Teile-und-Herrsche“ arbeitet:

(sortiert wird so, daß dann gilt f[i] < f[j], wenn i < j)

Der Quicksort-Algorithmus:

1. Wenn das Feld f aus einem Eintrag besteht, dann ist die Aufgabe gelöst.
2. Wähle einen Wert x aus dem Feld f aus.

/// **DIVIDE:**

3. Berechne eine Teilfolge f1, so daß f1 aus f mit Werten kleiner als x ist und eine Teilfolge f2 mit Werten größer als x.

/// **CONQUER:**

4. Wende das Verfahren auf die Teilfolge f1 und auf die Teilfolge f2 an.

Insbesondere Schritt 3 ist weiter zu verfeinern:

3.1 Hierbei ist zu beachten, daß idealerweise x einen mittleren Wert aus den Werten in dem zu sortierenden Feld darstellen sollte. Die Unterteilung startet mit einem Vorschlag/Initialisierung den Bereich in der Mitte zu teilen und sucht dort auch den Wert x aus:

```
i = links;
j = rechts;
x = f[(links + rechts) / 2]
```

Beispiel: links = 0, rechts = 9; => 9 / 2 = 4

3.2 Danach suchen wir von links den Wert in f für den gilt f[i] >= x; d.h. alle anderen werden überlesen

```
while ( f[i] < x )
      i = i + 1;
```

3.3 Von rechts suchen wir den Wert in f der kleiner ist als x,

```
while ( x < f[j] )
      j = j - 1;
```

3.4 Wenn i <= j, dann
werden die Werte an den Positionen i und j vertauscht:

```
y = f[i]; f[i] = f[j]; f[j] = y;
```

und i wird inkrementiert (erhöht), j wird dekrementiert (erniedrigt).

```
i = i + 1; j = j - 1;
```

Die Schritte müssen sooft wiederholt werden, bis i > j
!!! Wenn i = j muß auch eine Inkremtierung bzw. Dekrementierung erfolgen, da sonst u.U. das Abbruchkriterium nicht erreicht wird **!!!**

3.5 3.2 - 3.4 muß wiederholt werden, bis i > j

```
do ... while ( i <= j );
```

4. Wenn links < j dann Quicksort (links, j);
   Wenn i < rechts, dann Quicksort (i, rechts);

```
        if ( links < j  ) Sortiere ( links, j );
        if ( i < rechts ) Sortiere( i, rechts );
Sortiere ( links, rechts )
i = links; j = rechts;
x = f[(links + rechts) / 2];
wiederhole
  solange f[i] < x gilt wiederhole i = i + 1;
  solange x < f[j] gilt wiederhole j = j - 1;
  wenn i <= j dann
      vertausche (i, j)
      i = i + 1;
      j = j - 1;
solange i <= j
wenn links < j dann sortiere ( links, j );
wenn i < rechts dann sortiere ( i, rechts );
```

Was macht der Algorithmus mit folgendem Ausgangsfeld:

70 40 10 80 50 20 90 60 30 0

Feld										Quicksort	i	j	x
70	40	10	80	50	20	90	60	30	0	(0, 9)	5	4	50
[**0**	40	10	**30**	**20**]	**50**	90	60	**80**	**70**	(0, 4)	2	1	10
[0	**10**]	**40**	30	20	50	90	60	80	70	(0,1)	0	1	0
0	10	[40	30	20]	50	90	60	80	70	(2,4)	3	3	30
0	10	**20**	30	**40**	[50	90	60	80	70]	(5,9)	7	6	60
0	10	20	30	40	[**50**	**60]**	**90**	80	70	(5,8)	5	6	50
0	10	20	30	40	50	60	[90	80	70]	(7,9)	8	8	80
0	10	20	30	40	50	60	**70**	80	**90**	**ENDE**	-	-	-

Eigenschaften des Quicksort Algorithmus:

Im Durchschnitt ist das Verhalten(Laufzeit) von Quicksort deutlich besser als beim Bubble-Sort. Nur im schlimmsten Fall erreicht der Algorithmus das Verhalten von Bubble-Sort.

Quicksort ist für große Anzahlen deutlich besser als ein einfacher Sortieralgorithmus (z.B. Bubblesort). Quicksort ist also ein schneller Algorithmus und wurde daher zum Standard-Sortierverfahren. Es ist etwa in der Programmiersprache C implementiert und Bestandteil des Betriebssystems UNIX.

Bewertung der Qualität verschiedener Verfahren (Zeitmessung)

Es bieten sich 2 Möglichkeiten an:

- Zeitmessung
  Prozessorabhängig; u.U. benötigt das Betriebssystem auch Zugang zum Prozessor
- Anweisungen zählen
  Zuweisungen (Additionen), Multiplikationen, Bedingungen (Schleifen), Unterprogrammaufrufe zählen

Voraussetzung für den Vergleich ist:

- gleiche Testbedingungen, die wiederholbar sind:
- gleiches Datenmaterial
- gleicher Rechner

Streng genommen müsste man nun eine ganze Testreihe durchführen

- Sortierung verschiedener, gleichgroßer Felder,
- unterschiedliche großer Felder (sehr kleine bis sehr große)
- unterschiedliche unsortierte Felder (sortiert, umgekehrt sortiert, ...)

Suchverfahren

Gegeben seien folgende Informationen:

Feld f mit n Elementen: f[0 ..n-1] für die die folgende Beziehung gilt:

$f[i] < f[j]$, wenn $i, j \in \{0, 1, 2, .., n-1\}$ und $i < j$

Aufgabe: Ist x in f?

Ansatz zunächst:

```
void suche ( int x, int f[], int n)
/*    x: gesuchter Wert;
f: Feld mit Werten;
n: Größe des Feldes */
{
 int i;
 i = -1;
 do
  i = i + 1;
 while (i < n && x != f[i]);
 if ( i < n ) printf ("x in f gefunden\n");
 else printf ("x nicht gefunden\n");
}
```

Aufruf: `suche ( x, f, n)`

Es fällt auf, dass jedes Element betrachtet wird bis das Ende erreicht ist oder x im aktuellen Element gefunden wird.

Dies bedeutet, dass es gerade bei großem n sehr lange dauern kann, bis eine Aussage über x gemacht werden kann.

Zudem benutzen wir bei diesem Verfahren nicht die Information/Eigenschaft, die in Aufgabenstellung angegeben wurde!

Es wäre also viel geschickter, die Eigenschaft zu nutzen und etwa zunächst die Schrittweite zu erhöhen und dann in Intervallen zu suchen.

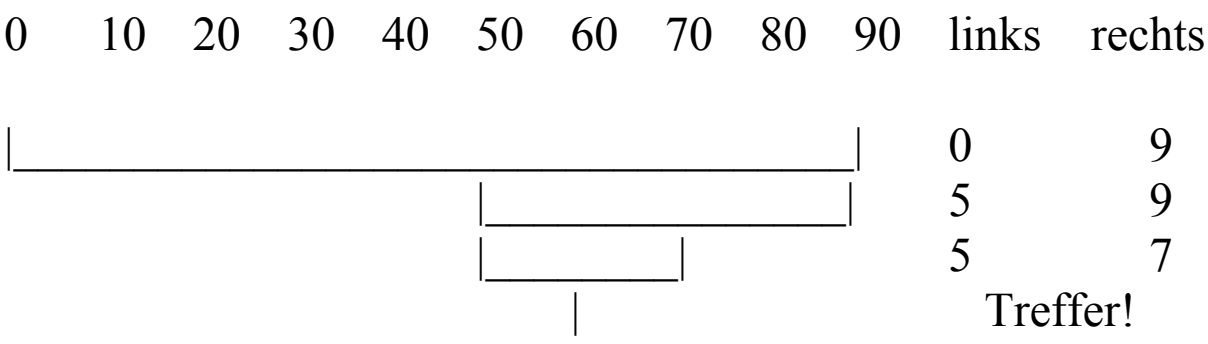
Das schnellste Verfahren beruht darauf, dass wir wieder das Feld/den Bereich so zerteilen, dass wir sukzessive den möglichen Indexbereich durch halbieren bzw. verschieben der Indexgrenzen einengen, bis wir den entsprechenden Wert gefunden haben. Den Aufruf müssen wir nicht modifizieren!

```
void suche ( int x, int f[], int n)
/*    x: gesuchter Wert;
f: Wertefeld;
n: Anzahl der Werte in dem Feld
Bemerkung: wenn x < f[0] oder x > f[n-1] müssen
 wir nicht suchen */
{
 int mitte, links, rechts;
 links = 0; rechts = n – 1;
 do
   {
    mitte = (links + rechts) / 2;
    if ( x < f[mitte] ) rechts = mitte – 1;
    else                links  = mitte + 1;
   }
 while ( f[mitte] != x && links != rechts);

 if  ( f[mitte] == x ||
     (links == rechts && f[rechts] == x))
    printf ("x in f gefunden\n");
}
```

Aufruf: suche (x, f, n)

Wir beobachten das Verhalten an folgendem Datenfall:

x = 60



Achten Sie darauf, dass der Algorithmus auch funktioniert, wenn des Feld

- nur 1 Element oder 2 Elemente besitzt,
- die Zahl am Rand steht (Position 0 oder n-1) oder
- nicht vorkommt

Der Algorithmus weist folgendes Verhalten auf:

- Er verhält sich immer gleich unabhängig vom Datenfall (d.h. es spielt keine Rolle, ob die Zahlen in dem Bereich gleich verteilt sind oder nicht)
- Der Aufwand ist damit immer gleich

Da bei jedem Schritt die Anzahl der prüfenden Elemente halbiert wird, können bei k Schritten 2^k Elemente geprüft werden. Bei 10 Vergleichen können bereits 2^{10} Daten durchsucht werden!

Weiter C-Syntaxelemente

Weitere Anweisungen in C

Manipulation einer Variablen mit Operator <op> und Ausdruck <expr>:

```
<variable> <op>= <expr>;
```

Inkrementierung bzw. Dekrementierung einer Integervariablen

```
<intvariable>++;
<intvariable>--;
++<intvariable>;
--<intvariable>;
```

Bedingte Zuweisung

```
<variable> = (<expr>) ? (<expr1>) : (<expr2>);
```

Sprunganweisungen

`break;`

break verläßt eine Schleife oder Fallunterscheidung und setzt die Ausführung **nach** dem Konstrukt fort.

`continue;`

continue bricht den Schleifendurchlauf ab und springt zum Schleifenkontrollkonstrukt

`goto <label>;`

Spring zu einer symbolisch benannten Stelle (eindeutiger Bezeichner) innerhalb des gleichen Programmteils (in der switch-Anweisung werden Labels verwendet)

`return [<expr>];`

return verläßt eine Funktion, übergibt den Rückgabewert und kehrt in das aufrufende Programmstück zurück

`exit (0);`

exit beendet die Programmausführung und übergibt die Kontrolle an das Betriebssystem

Anweisung: Komma-Operator

Der **Komma-Operator** trennt eine Auflistung von Anweisungen oder Ausdrücken voneinander. Sie werden von links nach rechts abgearbeitet. Das Ergebnis der Auflistung hat Typ und Wert des rechts außen stehenden Ausdrucks.

Bespiel:

```
int i, j = 10;
i = ( j++, j += 100, 999 );
printf ( "%d", i );
```

Zuerst wird j um 1 erhöht, dann auf den Wert von j 100 addiert und schließlich dem gesamten Ausdruck 999 zugewiesen. j hat also nach den Operationen den Wert 111, i den Wert 999; dieser Wert wird auch ausgegeben.

Der Komma-Operator kann sinnvoll im Kopf von for-Schleifen verwendet werden.

Das Komma zwischen Variablennamen in Deklarationen oder in einer Argumentliste ist **kein** Operator. Die Reihenfolge der Abarbeitung solcher Listen (Argumentlisten und Deklarationen) ist nicht definiert!

Der Komma-Operator trägt sicherlich zu kurzen und kompakten Programmen bei. Unter den Aspekten der Lesbarkeit und Wartbarkeit sowie eines guten Programmierstils ist er allerdings nicht uneingeschränkt zu empfehlen.

Ausdrücke als Anweisungen

Aus einer Zuweisung oder einem anderen Ausdruck wird durch Anhängen eines Semikolons eine Anweisung. Kommt eine Zuweisung beispielsweise als Argument in einer Funktion oder einer Bedingung vor, darf sie nicht durch ein eigenes Semikolon abgeschlossen werden. Die Zuweisung wird ausgeführt und der Wert an ihre Stelle gesetzt. Steht der Ausdruck allein muß er mit einem Semikolon beendet werden.

1. Beispiel:

```
printf ( "%d %f \n", x = 3, log(4) );
```

2. Beispiel:

```
while ( ( c = getchar() ) != 125 );
```

Die Schleife liest Zeichen ein und verwirft sie, bis sie auf ein Zeichen Nr. 125 (= rechte geschweifte Klammer) trifft. Dieses Zeichen wird zuletzt von der Schleife gelesen (und damit von der Eingabe entfernt), danach geht es nach der Schleife weiter.

Programmaufruf mit Parametern

Bereits beim Programmaufruf kann ein C-Programm mit Werten versehen werden. Das Programm kann mit Parametern aufgerufen werden, die dann in main abgefragt und verwendet werden können:

main hat dann 2 Parameter:

```
void main (int argc, char *argv[] )
{
 ...
}
```

- **argc**
  (argument counter) enthält als Wert die Anzahl der Parameter, die übergeben wird.
- ***argv[]**
  (argument value) ist ein Zeiger auf ein Feld aus Zeichenfolgen, wobei in C Felder an der Position 0 beginnen

Aufzählungen

Mit Hilfe des Kontruktes enum können Aufzählungen festgelegt werden.

Beispiel:

```
enum ampel {rot, gelb, gruen};
          // Typdefinition farbe

enum ampel kreuzung;
          // Varialbendeklaration
```

oder

```
typedef enum ampel AMPEL;
AMPEL kreuzung;    // Varialbendeklaration
```

Die Werte werden intern auf die Konstanten 0, 1, 2 .. abgebildet. Sie können allerdings auch bei der Definition explizit angegeben werden und in Anweisungen abgefragt werden (die Werte der folgenden Konstanten werden jeweils um 1 erhöht):

```
enum ampel {rot, gelb=4, gruen};
```

Operatoren in C

Operator	Bedeutung	Verarbeitungs-reihenfolge
( )	Funktions- und Ausdruckklammer	L -> R
[ ]	Arrayklammern	
->	Zugriff auf Struktur-element über Pointer	
.	Zugriff auf Struktur-element über Name	
!	Verneinung (0, wenn Wert !=0; 1, wenn Wert = 0)	R -> L
++	Inkrement	
--	Dekrement	
-	Vorzeichen-Minus	
*	Verweisoperator	
&	Adresse	
sizeof	Größe in Bytes	
(<Typ>)	Typumwandlung	

Unions

Unions werden syntaktisch ähnlich behandelt wie Strukturen; sie belegen aber im Speicher nur soviel Platz wie das größte Element der Union benötigt. Zu einem Zeitpunkt können sie immer nur eines der in ihnen definierten Elemente enthalten.

Es besteht somit nicht nur die Gefahr, dass die verschiedenen Elemente sich gegenseitig überschreiben, sondern mit Unions wird **eine** Speicherstelle vereinbart, die Variablen verschiedenen Typs aufnehmen kann.

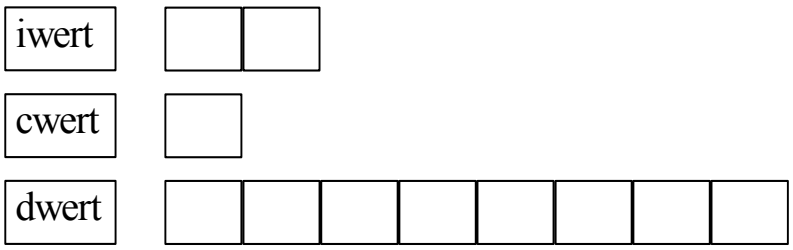
Zur Unterscheidung, wie auf die Speicherstelle zugegriffen werden darf, sollte man daher zusätzlich eine andere Variable oder Komponente definieren, in der man sich den Typ merkt, der derzeit in der union gespeichert ist.

Unionselemente werden genau wie Strukturelemente angesprochen:

```
Unionsname.Elementname oder
Unionszeiger->Elementname
```

Beispiel:

```
int typ;
    union value
          { int   iwert;
            char   cwert;
            double dwert; } w;
if (typ == CHAR)
    printf ("wert = %c\n", w.cwert);
```



Dateizugriff 2

Die Ein- und Ausgabe mit fgets und fputs

Syntax:

```
char *fgets(char *s, int n, FILE *stream);
int   fputs(const char *s, FILE *stream);
```

fgets liest die nächste Eingabezeile einer Datei einschließlich des Zeilenvorschubs in den String ein. Maximal werden n-1 Zeichen gelesen. Der Funktionswert ist normalerweise der Zeiger (char *) auf den String. Bei Dateiende ist der Funktionswert NULL.

fputs schreibt den String auf die mit stream angegebene Datei.

Die Befehle **gets** und **puts** sind für die Standardeingabe bzw. -ausgabe definiert.

Die Ein- und Ausgabe fread und fwrite

Syntax:

```
size_t fread( void *ptr, size_t size,
              size_t n, FILE *stream);
size_t fwrite( const void *ptr, size_t size,
               size_t n, FILE *stream);
```

Beide Funktionen brechen ab beim Auftreten irgendeines Fehlers (fread auch bei EOF) und beide Funktionen liefern die Anzahl der tatsächlich übertragenen Felder als Rückgabewert zurück.

Im Sinne der Eingabekontrolle sei darauf hingewiesen, dass die jeweiligen Rückgabewerte kontrolliert werden sollten!

Wahlfreier Zugriff auf Dateien

In einer großen Anzahl von Anwendungen reicht es aus, wenn das Programm sequentiell auf die Daten in der Datei zugreift; d.h. die einzelnen Elemente werden der Reihe nach gelesen bzw. geschrieben.

In bestimmten Fällen (z.B. Datenbankanwendungen oder Bilddateien oder Cachespeicher) möchte man einen freien, direkten Zugang (random access) zu den einzelnen Daten haben und diese auch selektiv lesen oder schreiben können. Dafür steht der Befehl

```
int fseek (FILE *fp, long offset, int origin);
```

zur Verfügung.

Die nachfolgende Lese- oder Schreiboperation greift ab der neuen Position auf die Daten zu. Für eine binäre Datei wird die Position auf offset Zeichen relativ zu origin eingestellt; dabei können die Werte

```
SEEK_SET  (Dateianfang)
SEEK_CUR  (aktuelle Position) oder
SEEK_END  (Dateiende) angegeben werden.
```

fseek liefert einen von NULL verschiedenen Wert bei einem Fehler.

Der binäre Dateityp

Außer Textdateien (im sog. ASCII-Format) kann man auch binäre Dateien anlegen und lesen. Auch wenn Textdateien üblicherweise umfangreicher sind, bieten sie gerade in der Programmentwicklung den wichtigen Vorteil, dass die Daten vom Entwickler gelesen und somit die Ein- und Ausgabe kontrolliert werden kann. Außerdem sind Textdateien unabhängig von bestimmten Implementierungsannhamen (z.B., ob eine Integervariable 2 oder 4 Byte Speicherplatz belegt). Diese größere Flexibilität muß allerdings durch erhöhte aber vertretbare Sorgfalt bzw. Programmieraufwand bezahlt werden.

Bei Rasterbildern wird allerdings zur (kompakteren) Abspeicherung der Pixelwerten eine Binärdatei oder ein Binärformat verwendet.

In der Regel erfolgt dann byteweiser Zugriff auf die Datei. Kritisch wird diese Vorgehensweise allerdings gerade unter C, wenn man das Programm auf einem anderen Rechnermodell ablaufen läßt (denken Sie daran, dass unter C die Größe der Datentypen für int nicht festgelegt ist!).

Dynamische Datenobjekte: Pointer

Die bisher betrachteten Datenobjekte (Variable) wurden von der Programmiererin/ dem Programmierer bei Programm- oder Blockbeginn vereinbart und sind dann innerhalb des Programms bzw. Blocks permanent unter dem bei der Deklaration angegebenen Bezeichner ansprechbar.

Man nennt diese Variablen auch **statische Datenobjekte**, da ihre Lebensdauer der statischen Programmstruktur entnommen werden kann. Ihre Anzahl und die Speicherbelegung ist vom Compiler überprüfbar und organisierbar.

Diese statischen Datenobjekte weisen eine konstante Größe auf (bereits zum Zeitpunkt der Implementierung muß etwa bekannt sein, wie groß ein Feld ist). Neben dem Vorteil der einfachen direkten Adressierung weist dies allerdings folgende Nachteile auf:

- Felder sind ungeeignet, wenn neue Elemente eingefügt oder vorhandene gelöscht werden sollen.
- die Maximalgröße des Feldes muß vorher festglegt werden (Speichplatz wird reserviert und kann u.U. nicht genutzt werden oder ist doch zu knapp bemessen)
- der Index auf Minimum oder Maximum oder andere ausgezeichnete Werte sind durch Einfügen oder Entfernen nicht stabil.

Im Gegensatz dazu kann man sich auch vorstellen, daß Datenobjekte zu einem beliebigen Zeitpunkt unabhängig von der statischen Blockstruktur angelegt und zu einem anderen Zeitpunkt wieder freigeben werden. Solche Datenobjekte nennt man **dynamische Datenobjekte** (**Pointer** oder **Zeiger**).

Dynamische Datenobjekte eignen sich besonders dann, wenn man zum Zeitpunkt der Programmentwicklung noch nicht den genauen Speicherplatz oder die Komplexität der zu manipulierenden Daten kennt.

Beispiele:

- Verwaltung von Objekten (z.B. von Büchern, CD, Konten, Einwohnern, Mitarbeitern) sowie Operationen um neue Objekte einzufügen, Objekte zu löschen, Objekte zu sortieren, ...
- Der (Zeige-)Finger des Lesers, der beim Lesen eines Buches über den Text wandert, ist ein Zeiger, der auf eine Textstelle (Buchstabe innerhalb eines Wortes einer Zeile einer Seite eines Buches) zeigt
- Die Anschrift einer Person **zeigt auf** die entsprechende Wohnung
- Der 2. Student von links (aus Sicht des Dozenten) in 3. Reihe
- Die Zeiger in den Aufgaben zu den Stacks – sie zeigen auf die nächste freie Stelle oder Adresse

Außerdem haben wir in versch. Übungsaufgaben gesehen, dass zum Zeitpunkt der Implementierung noch nicht bekannt ist, wie groß bestimmte Felder sind:

- Stacks mußten ausreichend - wie groß ist „ausreichend“? - dimensioniert werden
- Die Länge von Strings mußten wir abschätzen
- Die Anzahl der zu sortierenden Zahlen mußte begrenzt werden.

Ein **Pointer/Zeiger** ist ein Datenelement, das die Adresse eines anderen Datenelementes als Inhalt der eigenen Speicherzelle aufnehmen kann.

Mit Hilfe von Standardprozeduren werden während der Programmausführung bei Bedarf neue Datenobjekte (neue Adresse und Platz im Speicher) geschaffen (**malloc**) oder auch wieder freigegeben (**free**), damit dieser wieder für andere Zwecke (z.B. vom Betriebssystem) verwendet werden kann. Pointer können auf solche Objekte verweisen. Dabei muß man deutlich unterscheiden zwischen dem Verweis und dem eigentlichen Datum.

Da Pointer nahezu beliebig manipuliert werden können, kommt ihnen **in C eine zentrale Rolle** zu (z.B. kann ein Feldindex als Pointer=Abstand zum Varialbenanfang interpretiert werden!).

NULL ist ein ausgezeichneter Wert bei Pointern. Ein Pointer kann mit NULL initialisiert werden, wenn etwa erreicht werden soll, daß der Pointer einen definierten Wert hat aber auf keinen Speicher verweisen soll.

Schreibweise und Umgang:

> Beim Umgang mit Pointern werden Sie sehr häufig die beiden Operatoren ‘*’ (= **Inhalt von**) und ‘&’ (= **Adresse von**) verwenden.

Wenn **ptr** eine Pointervariable und **i** eine normale Variable des gleiche Typs ist, dann können folgenden Zuweisungen gemacht werden:

```
i = *ptr; /* der Inhalt der Speicherzelle, auf
             die der Pointer ptr zeigt wird der
             Variablen i zugewiesen */

Ptr = &i; /* die Adresse von i wird in die
             Pointervariable eingetragen */
```

Pointerarithmetik

Pointervariablen können in beschränktem Umfang auch in arithmetischen Ausdrücken vorkommen:

- Werte von Pointervariablen dürfen anderen Pointervariablen (des gleichen Typs) zugewiesen werden.
  WARNUNG: Wenn die Pointertypen nicht übereinstimmen, erhalten sie **keine** Fehlermeldung!
- Pointervariable ohne konkreten Wert sollten den Wert NULL, welches in C ein Schlüsselwort ist, zugewiesen bekommen.
- Pointervariable dürfen inkrementiert und dekrementiert werden (kann z.B. hilfreich bei Feldindices sein). Die Auswertungsreihenfolge ist von links nach rechts.
- Pointervariable können mit anderen Pointervariablen oder mit NULL verglichen werden; möglich sind folgende Vergleiche: ==, !=, <, >, <=, >= .
- Pointervariable dürfen um Integerwerte erhöht/vermindert werden.
- Der Abstand zwischen zwei Pointern darf ermittelt werden (z.B. ptr1 - ptr2).

Pointer in Arrays

Der Zugriff auf Array-Elemente über Pointer ist in der Regel schneller und in der Programmierung effizienter als über Array-Indizierung.

In C sind folgende Konstrukte erlaubt:

```
int arr[10], *ptr;
ptr  = &arr[0];
arr[0] = *ptr;
arr[1] = *(ptr+1);
ptr  = arr;
```

Äquivalent sind die folgenden Schreibweisen:

```
arr[i]  ≡ *(arr+i);
&arr[i] ≡ arr + i;
```

Streng genommen ist der einzige Unterschied zwischen Arrayname und Pointervariable (vom gleichen Typ!), dass ein Arrayname keine Variable, sondern eine Adreßkonstante ist.

Anweisungen der Art

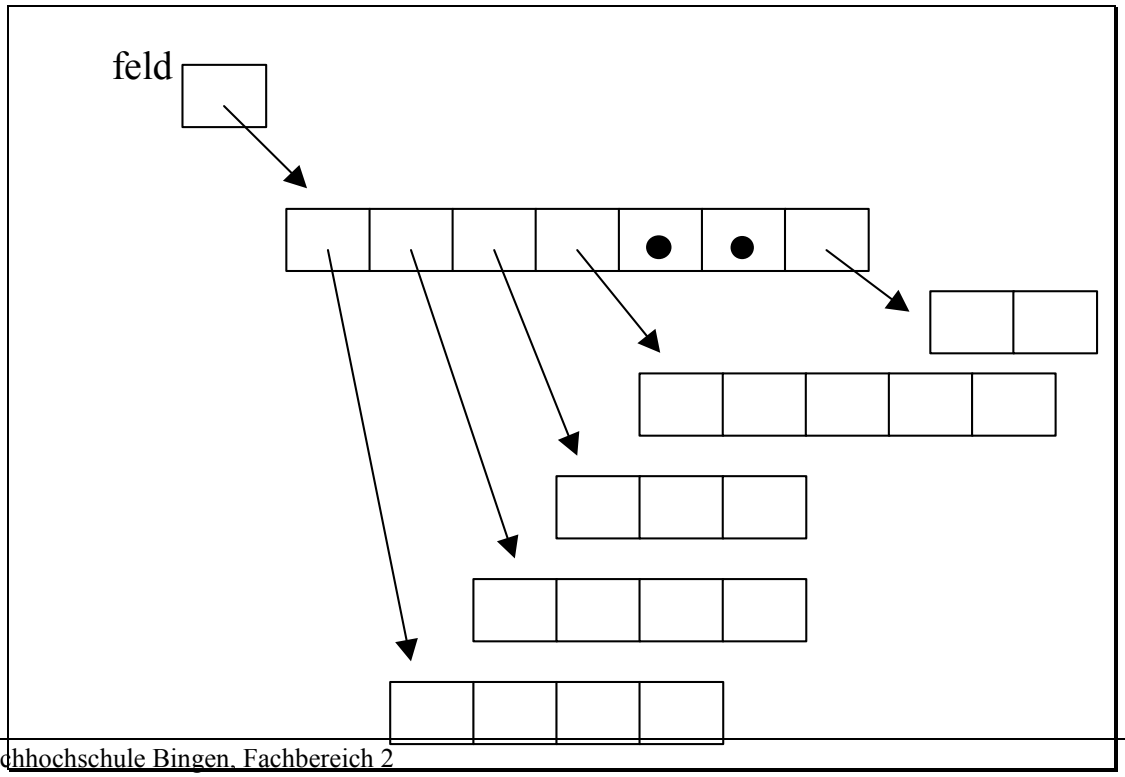
```
arr++;
arr = ptr;
```

sind **nicht** zulässig, wenn **arr** eine Feldvariable ist.

Dynamische Arrays

Felder können in C dynamisch angelegt werden.

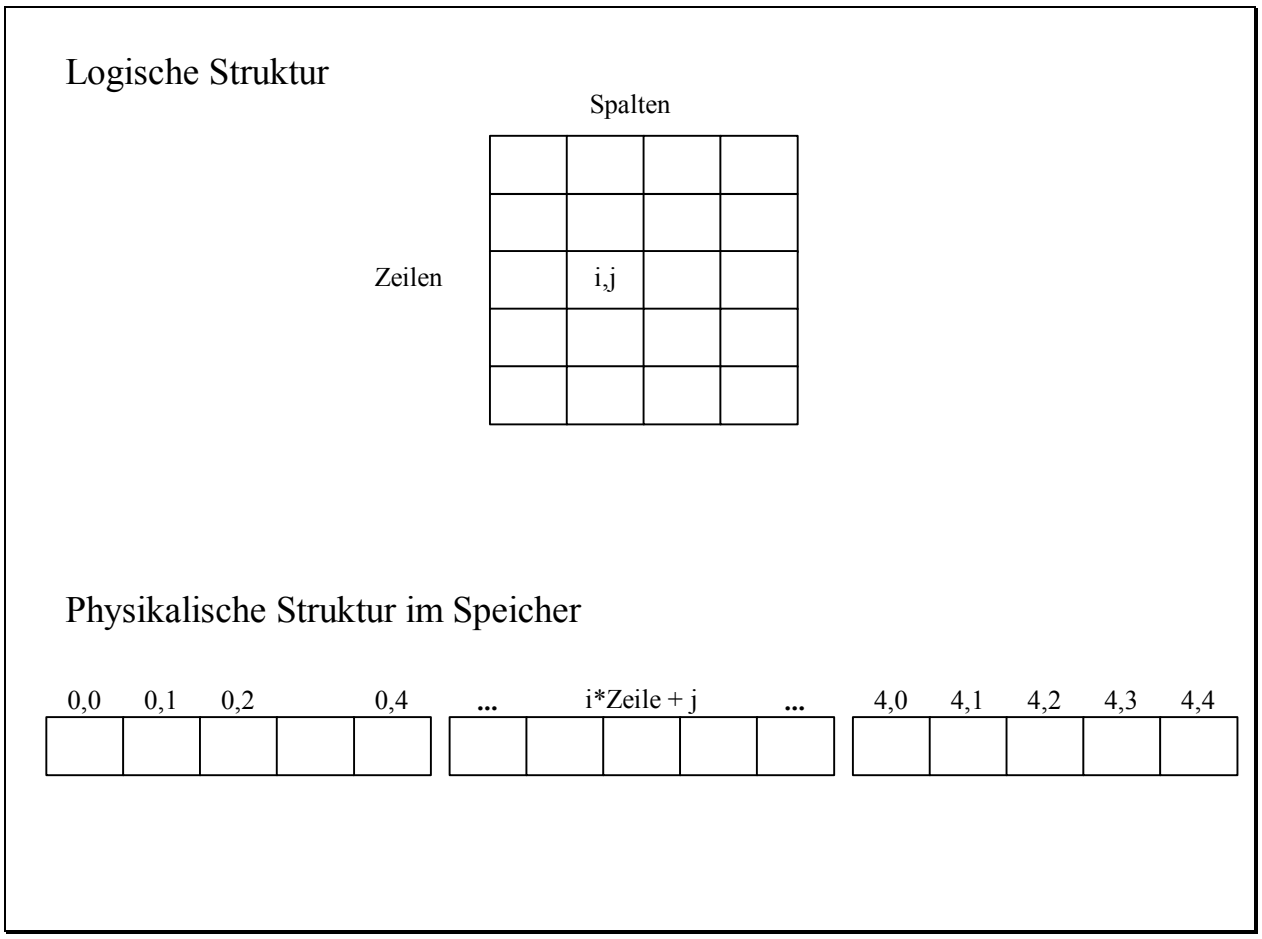
```
typedef int* PtoArray;
PtoArray *feld;

feld = (PtoArray *)
          malloc (7 * sizeof (PtoArray));
if ( feld == NULL ) printf ("FEHLER!\n");

feld[0] = (PtoArray) malloc (4*sizeof (int));
if (feld[0] == NULL) printf ("FEHLER!\n");
...

feld[3] = (PtoArray) malloc (5*sizeof (int));
if (feld[3] == NULL) printf ("FEHLER!\n");
...
```



Bei der Verwendung von mehrdimensionalen Feldern sollte man sorgfältig vorgehen und dabei einige Regeln kennen.

```
int f[5][4];
int *v;
```

Generell gilt, der 2. (am weitesten rechts stehende) Index läuft am schnellsten bzw. die zu diesem Index gehörenden Werte stehen im Speicher nebeneinander.




Aufgabe:

1. Wie kann in einem Unterprogramm unterschieden werden, ob es sich um ein mehrdimensionales Feld oder um Zeiger auf Felder handelt?
2. Sind die beiden folgenden Ausdrücke gleichbedeutend (die Variablen beziehen sich auf die oben benutzten Deklarationen)?

```
v = feld[1];
v = f[1];
```

3. Wie müssen Sie das oben angegebene Beispiel ergänzen, damit Sie im Programm wissen, wie groß die einzelnen Elemente sind?

malloc, calloc, free, sizeof

Mit Hilfe von Pointervariablen können dynamisch zur Programmlaufzeit und in Abhängigkeit des Datenfalles Speicherplatz vom Betriebssystem angefordert und - wenn nicht mehr benötigt - freigegeben werden
(zu verwendende Include-Datei: stdlib.h).

C bietet hierzu die folgenden Standardprozeduren an:

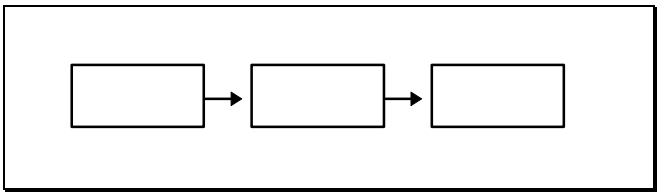
- **void *malloc (<Länge>):**
  Malloc (memory allocation) liefert einen char-Pointer auf einen zusammenhängenden Speicherbereich der Größe <Länge> in Bytes.
  Alle Parameter dieser Funktion werden als vorzeichenlose “unsigned-Werte“ übergeben.
  Ist kein genügend großer zusammenhängender Speicherbereich verfügbar, liefert die Funktion den NULL-Pointer.
  Strenggenommen ist bei der Speicheranforderung eine Typanpassung/-konvertierung des Rückgabewertes erforderlich.
- **void *calloc (<Anzahl>, <ElLänge>):**
  Calloc verhält sich ähnlich wie malloc: es wird Speicherplatz zur Verfügung gestellt, der so groß ist, daß <Anzahl> viele Datenelemente der Länge <ElLänge> hineinpassen. Zusätzlich wird der Speicherbereich mit 0 initialisiert.
- **void free (<Pointer>):**
  Einen zuvor mit malloc oder calloc angeforderten Speicherbereich gibt man mit dieser Funktion wieder frei, in dem man die Funktion free mit dem beim

Reservieren erhaltenen <Pointer> als Parameter aufruft.
Der Speicherbreich steht dann wieder für eine andere Speicheranforderung zur Verfügung. Der alte Inhalt wird nicht automatisch gelöscht, der Zugriff ist dann nicht mehr möglich bzw. führt zu nicht vorhersagbarem Fehlverhalten!

- **int sizeof (<Typ>):**
  Mit diesem Operator ist es möglich, die Größe der Datenelemente eines bestimmten Typs und von Variablen abzufragen. Der Rückgabewert ist die Länge des Datenelementes in Byte. Diesen Operator kann man insbesondere verwenden, wenn man Speicherplatz für eine neue Variable anfordert.

Listenoperationen (Skizze)

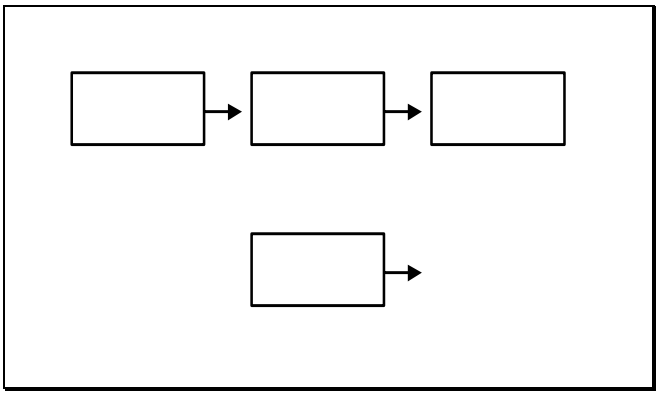
Liste ist die Datenstruktur, mit der gearbeitet wird; p ist das Element, nach dem ein neues Element eingefügt werden soll.



Anfangssituation/Voraus-setzung/ Vorbedingung:

Liste mit bel. vielen Elementen; die Liste kann auch leer sein!

1. Schritt

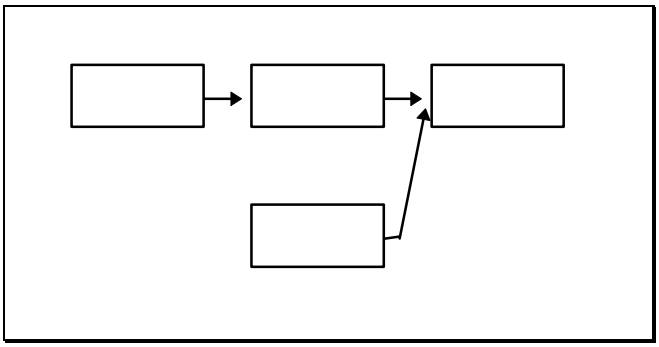


Es wird ein neues Objekt (Instanz/Ausprägung) einer Datenstruktur geschaffen

und die Kompnenten werden initialisiert:

```
neu = (Liste *)malloc (sizeof ( Liste);
if ( neu == NULL ) FEHLERBEHANDLUNG!!
neu->zahl = 0;
```

2. Schritt

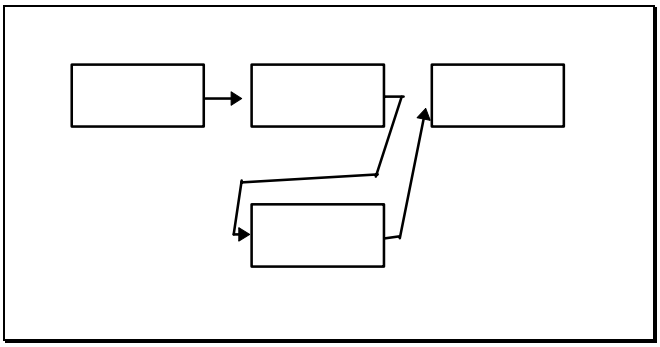


Das neue Objekt zeigt/verweist, auf den gleichen Nachfolger wie das Objekt nach dem eingefügt werden soll:

Was ist am Listenende?

```
neu->nxt = p->nxt;
```

3. Schritt



Das neue Objekt wird Nachfolger des Objektes/Listenelementes, nach dem es eingefügt werden soll:

Was ist, wenn vor dem Listenanfang eingefügt werden soll?

```
p->nxt = neu;
```

Ergebnis: