

Peter A. Darnell, Philip E. Margolis

C: A Software Engineering Approach, 3. Edition

Springer Verlag, 1996, ISBN 0-387-94675-6

Grundlage der Vorlesung. Didaktisch sehr schön. Betont guten Programmierstil im Sinne der Softwaretechnik. Neue, dem ANSI C Standard entsprechende aktualisierte Version. Leider nur in Englisch.

Brian W. Kernighan, Dennis M. Ritchie

Programmieren in C, Zweite Ausgabe, ANSI C

Carl Hanser Verlag, 1990, ISBN 3-446-15497-3

Deutsche Übersetzung des Originals von K&R. Voll dem ANSI C Standard angepaßt. Nicht so schön didaktisch aufbereitet wie das Buch von Darnell & Margolis, eher knapp ausgelegt, aber sehr gute Referenz

RRZN Schriftenreihe

Die Programmiersprache C. Ein Nachschlagewerk

Wie der Name sagt: Eher knapp, eher zum Nachschlagen. Keine Hinweise auf guten Programmierstil o.ä.

Karlheinz Zeiner

Programmieren lernen mit C

Carl Hanser Verlag, 2. Auflage 1996, ISBN 3-446-18637-9

C für Anfänger aufbereitet. Gute Hinweise zu Programmiertechniken.

Jürgen Dankert

Praxis der C-Programmierung

B.G. Teubner Verlag, 1997, ISBN 3-519-02994-4

C für UNIX, DOS und MS-Windows 3.1/95/NT. Gute Beispiele, auch Hinweise zur Windows-Programmierung.

Samuel P. Harbison, Guy L. Steele Jr.

C: A Reference Manual, 4. Edition

Prentice Hall, 1995, ISBN 0-13-326224-3 (Paperback)

Reines Referenzbuch, nicht zum „Lernen,, von C. Aber die ultimative Antwort in Zweifelsfragen

Kurt Ackermann,

Programmieren in C, Eine Einführung, Vorlesung an der Justus-Liebig-Universität Gießen, SS 1995

Recht guter C-Kurs als HTML-Dokument

Standard C Reference

HTML-Dokument auf unserem Server

1. Einleitung

- 1.1. Ziele der Vorlesung
- 1.2. Organisatorisches
- 1.3. Entwicklungsstadien der Hardware
- 1.4. Einbettung der Programmierung in den
Anwendungsentwicklungsprozeß
- 1.5. Sprache als Kommunikationsmittel
- 1.6. Algorithmus und Programm

Vorlesung Programmieren 1

„C“

Ziele

Vermitteln der Fertigkeiten und Techniken des Programmierens

- mit Hilfe einer Sprache der 3. Generation, d.h. einer problemorientierten, imperativen Sprache
- am Beispiel vom C

Dazu gehören

- Prinzipien des Programmierens und
- die Programmiersprache als solche.

Zusammenhänge

Grundlage für weiterführende Vorlesungen über andere Programmiersprachen, wie C++, Java, Pascal (Delphi), FORTRAN, COBOL, etc.

Erste Vorlesung der Reihe über *Programmierungsmethodik* und *Softwaretechnik* mit den Folgevorlesungen

- Programmieren 2 (C++),
- Liste P und
- Softwaretechnik– im Hauptstudium –

Organisatorisches zur Vorlesung *Programmieren 1*

2-std. Vorlesung + 2-std. Praktikum

75% Anwesenheit bei Praktikum ist Pflicht

Ein Teil der Praktikumsaufgaben, meist Programmieraufgaben, sind Pflichtaufgaben. Daneben sind auch Bewertungen mündlicher Leistungen durch die Dozenten möglich.

Gesamtnote Programmiermethodik (PL) über Stoff von
Programmieren 1 und 2:

Programmieren 1 : Klausur (80%) / Praktikum (20%)

Programmieren 2 : Klausur (80%) / Praktikum (20%)

- Jede Klausur kann zweimal wiederholt werden (PL)
- Praktika sind Studienleistungen
- Praktikum kann beliebig oft wiederholt werden
- Jede Teilleistung muß einzeln bestanden sein
- Änderung durch 2. Teilnahme nicht möglich

- Anmeldung zur PL Programmieren 1 im WS!
- Anmeldung zur PL Programmieren 2 im SS!

Vorlesung Programmieren 1

„C“

Inhalte

In dieser Vorlesung:

- Elemente der sog. *Strukturierten Programmierung* zur Darstellung der Algorithmen (Kontrollstrukturen, Methoden)
- Datentypen (Datenstrukturen)
- zusammen --> Objekte !
- Dynamische Datenstrukturen
- Entwicklung des Gefühls für bestmöglichen Einsatz dieser Elemente

In den weiterführenden Vorlesungen

- Bildung von Modulen (Datenkapselung, Abstrakte Datentypen (ADT))
- Objektorientiertes Programmierparadigma

Im Hauptstudium (Softwaretechnik)

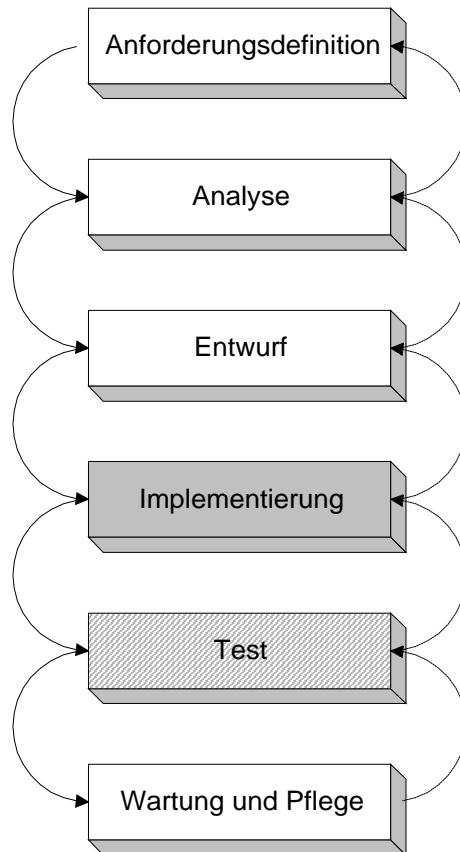
- Phasenmodell des Software-Lebenszyklus
- Methodik der Software-Entwicklung für einzelne Phasen

Im Hauptstudium (Betriebssysteme, Compiler, Graphische DV, Rechnernetze, Vertiefungsfächer)

- Spezielle Anwendungen und Aufgabenstellungen

Phasenmodell des Software-Lebenszyklus

Überblick



Phasen des Software-Lebenszyklus (1/2)

Anforderungsdefinition

- Spezifikation der Anforderungen an das Produkt aus Anwendersicht.

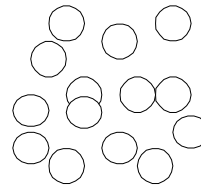
Analyse

- Architektur der Anwendung. Aufteilen der Anwendung in in sich zusammenhängende Teile (Module) mit sauber definierten Schnittstellen zur Kommunikation der Module untereinander.
- Dadurch wird das Gesamtsystem besser verständlich, die Komplexität des Gesamtsystems reduziert und handhabbar. Jedes Modul kann weitgehend separat entwickelt und getestet werden.
- Ein Modul besteht aus der
 - Definition von Datenstrukturen, den tatsächlichen Daten (von diesen Strukturen) und
 - Algorithmen, die diese Daten manipulieren.

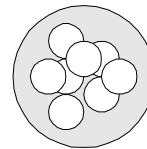
Jedes Modul hat nach außen sichtbare Schnittstellen, die seine Verwendung durch andere Module ermöglichen.

Struktur eines Programmes

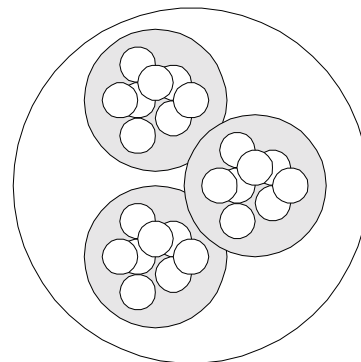
Ein Programm besteht im Quelltext aus einzelnen Befehlen und Ausdrücken der Programmiersprache



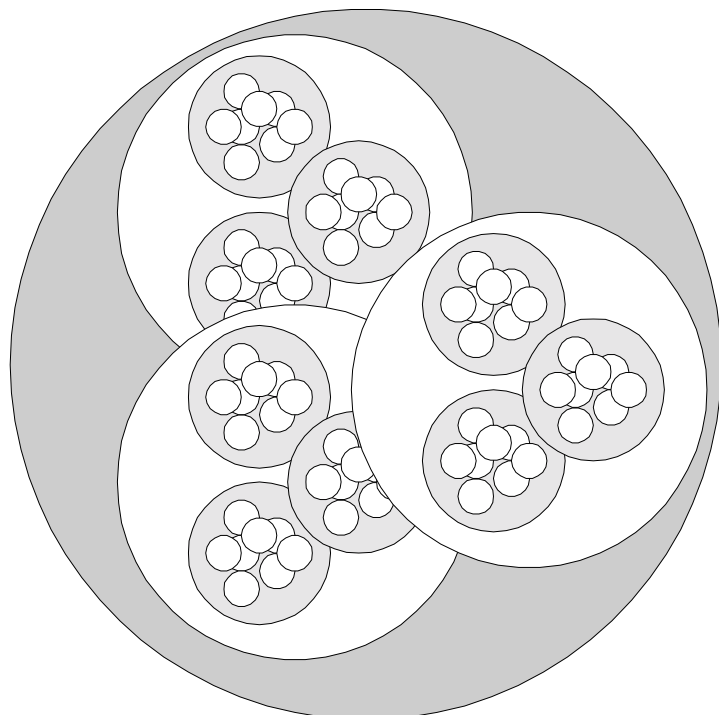
Funktionen bestehen aus mehreren Befehlen und Ausdrücken



Module bestehen aus zusammengehörigen Funktionen



Das gesamte Programm besteht aus mehreren Modulen



Phasen des Software-Lebenszyklus (2/2)

Entwurf (Design)

- Hier werden nun die Module in ihren technischen Einzelheiten geplant.
- Es werden die nötigen Funktionen definiert, deren Schnittstellen und welche Schnittstellen nach außen sichtbar sein müssen.
- Interne Algorithmen werden mehr oder weniger formal definiert.

Implementierung

- Auf der Basis der vorangegangenen Analyse- und Design-Ergebnisse werden jetzt die einzelnen Module programmiert.

Test

- Schon beim Implementieren werden einzelne Funktionen, dann die vollständigen Module und zuletzt die komplette Anwendung getestet.

Wartung und Pflege

- Beseitigung von Fehlern nach Abgabe der fertigen Anwendung und
- Hinzufügen neuer Features aufgrund erst während des Betriebes aufgetretener weiterer Anforderungen.

Programmiersprachen

Maschinen- und Assemblersprache

Maschinen-Code	Assembler-Code	Bedeutung
0030	LDA Zahl1	Lade Zahl1 in Register A
0032	ADA Zahl2	Addiere Zahl2 zu Inhalt des Registers A, Ergebnis in A
0034	STA Zahl3	Speichere Ergebnis aus A in die Spei- cherstelle Zahl3

Hier fehlen noch der Programm-Rahmen und die Definitionen der Variablen!

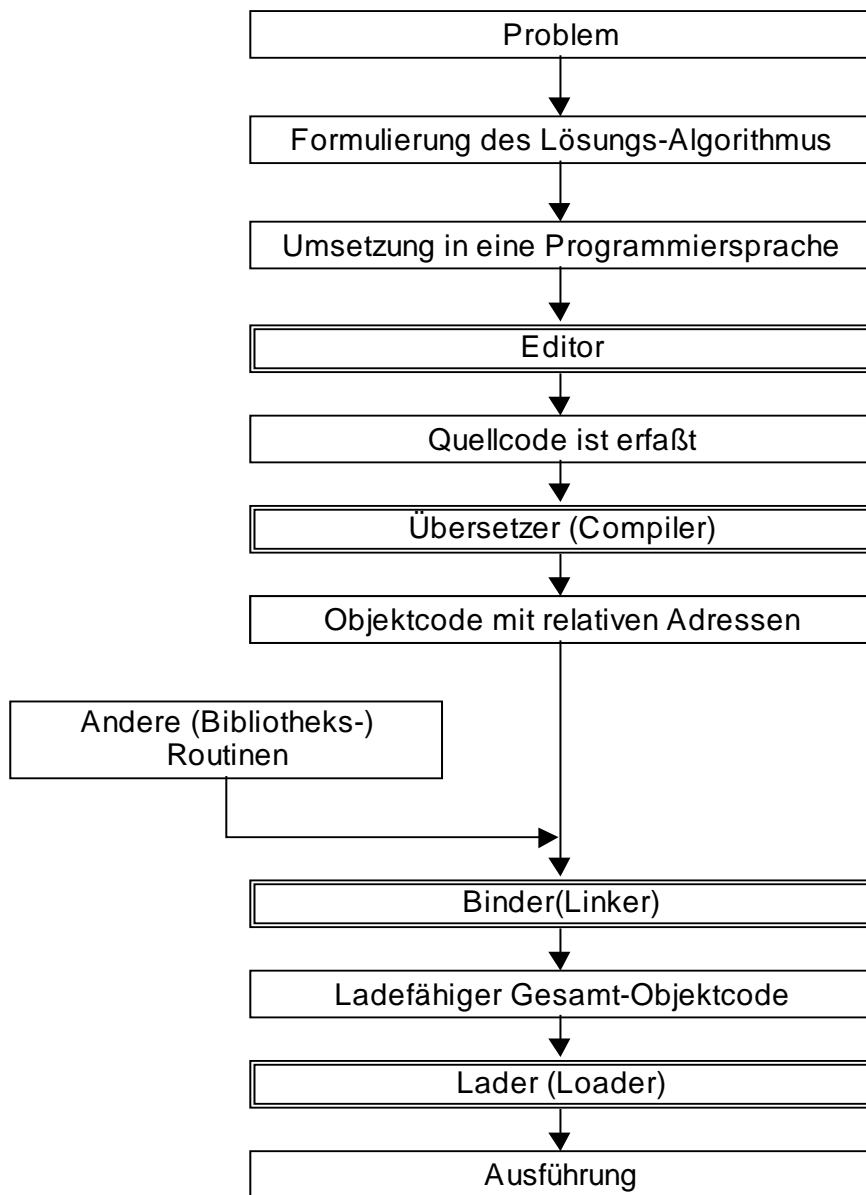
3GL-Sprache („C„)

```
#include <stdio.h>
int main(void)
{
    int Zahl1, Zahl2, Zahl3;                /* Deklarationen */

    printf("Gib 2 Integer-Zahlen: ");
    scanf("%d %d", &Zahl1, &Zahl2);        /* Be-          */
    Zahl3 = Zahl1 + Zahl2;                  /* feh-        */
    printf("Summe = %d\n", Zahl3);          /* le          */
    return 0;
}
```

Vollständiges Programm.

Vom Problem zur Ausführung des Programm



Historie der Programmiersprachen

- Frühe höhere Sprachen: FORTRAN, COBOL: zunächst herstellerbezogen
- Blockorientierte, strukturierte Sprachen: ALGOL 60 und 68, PL/1, Pascal, Modula, Ada; **C**
- Objektorientierte Sprachen: Smalltalk, C++, Eiffel, Borland Pascal 6.0/7.0, Delphi 1/2/3/4, . . .

C

- **C** im Jahre 1972 entwickelt von Dennis M. Ritchie (Bell Labs von AT&T) als Sprache zur Systemprogrammierung (also zur Entwicklung von Betriebssystemen). Vorgänger war B.
- Kompromisse zwischen
 - guter Lesbarkeit des Quellcodes einerseits und
 - den Zugriffsmöglichkeiten auf einzelne Maschineninstruktionen (Performance-Optimierung) andererseits.
- Erste Anwendung für das UNIX-Betriebssystem
-> Portabilität auf verschiedene Hardware-Plattformen.
- Dokumentation zunächst nur „The C Reference Manual,, von Ritchie.
- 1977 Buch „The C Programming Language,, von Dennis M. Ritchie & Brian Kernighan
-> K&R-Standard der Sprachdefinition. Enthielt aber immer noch zu viele Auslegungsmöglichkeiten, d.h. zu wenige Details für die Compiler-Bauer. Jede Compiler-Variante verhielt sich in Details anders.
- Hinzu kam als weiterer de-facto Standard PCC (Portable C Compiler) in der UNIX-Welt.
- Erst 1989 wurde C vom ANSI-Komitee standardisiert (ANSI C), im folgenden Jahr 1990 wurde dieser Standard zur internationalen ISO-Norm erhoben (ISO C). Beides sind heute Synonyme.

Historie von C

Weitere Entwicklung

- C++ ist eine Übermenge von C, die von der *strukturierten* Programmierung zur *objektorientierten* Programmierung führt. Einige heute nicht mehr aktuelle Konstrukte des K&R C werden (zum Glück) nicht mehr unterstützt.
- Beschränkt man sich auf eine – venünftige – Untermenge der Möglichkeiten von C, so kann dieser Code ohne weiteres von einem C++ Compiler übersetzt werden. Diese Untermenge von ANSI C wird auch *Clean C* genannt. Wir wollen hierauf besonderen Wert legen.

Achtung!

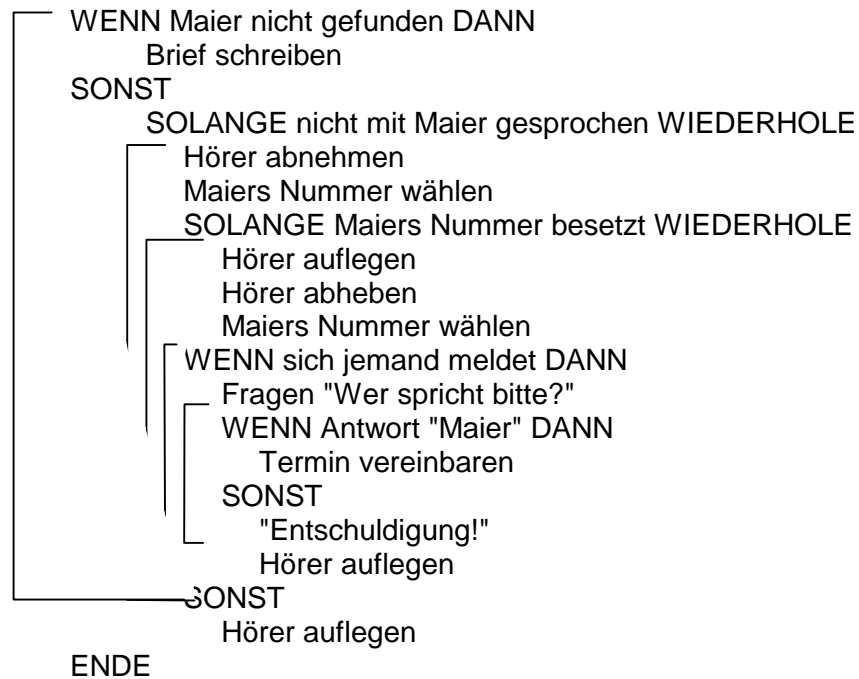
- Natürlich kann man mit C äußerst chaotische und vor allem unverständliche Programme schreiben.
- Unterschied zwischen
 - *guten* und
 - (nur) *funktionierenden* Programmen

Ein gutes Programm hat nicht nur die Eigenschaft, daß es wie vorgesehen *funktioniert*, sondern es ist auch *leicht zu lesen, zu verstehen und zu warten*.

Algorithmus

Tägliches Leben: Gesprächstermin mit Herrn Maier vereinbaren:

Den Namen Maier im Telefonbuch suchen



Anforderungen an einen Algorithmus

Endlich

Darstellbar durch endlich viele Zeichen (statisch endlich) und in endlicher Zeit ausführbar (dynamisch endlich)

Deterministisch

Jeder Schritt bestimmt eindeutig den nächsten Schritt

Effektiv

Jeder Einzelschritt ist eindeutig ausführbar

Effizient

Es wird möglichst wenig Rechen- und Speicheraufwand benötigt

Ein Algorithmus ist

eine in der Beschreibung und Ausführung endliche, deterministische und effektive Vorschrift zur Lösung eines Problems, die effizient sein sollte.

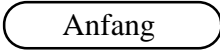
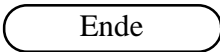
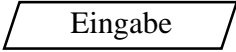
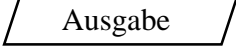
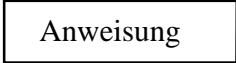




2. Grafische Darstellung von Algorithmen

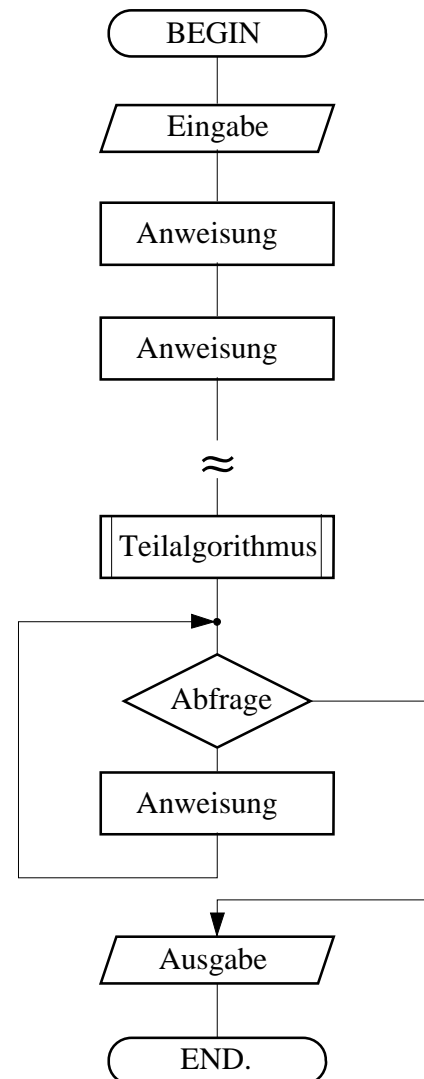
2.1. Ablaufdiagramme

2.2. Struktogramme

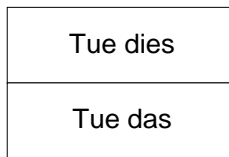
Ablaufdiagramme

Standardisierte Symbole:

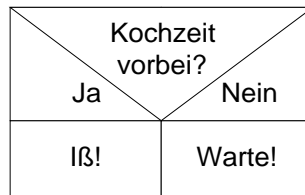
Symbol	Bedeutung
	Beginn des Programms
	Ende des Programms
	Eingabe, Einlesen von Daten
	Ausgabe, Schreiben von Daten
	Aktion, Befehl
	Nicht näher ausgeführter Teil, Prozedur oder Subroutine
	Abfrage, Entscheidung, bedingte Verzweigung
	Sprung, unbedingte Verzweigung
	Übergangsstelle nach/von anderem Programmteil in der Dokumentation



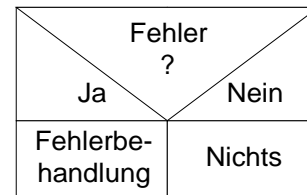
Struktogramme (1/2)



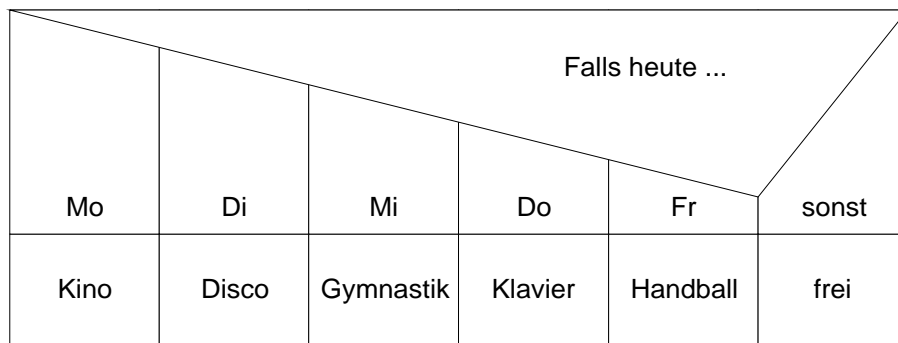
Sequenz



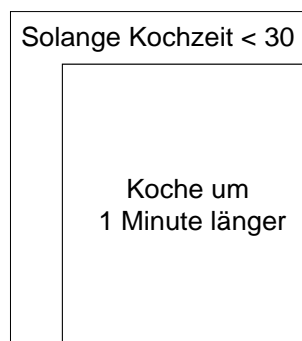
Binäre
Entscheidung



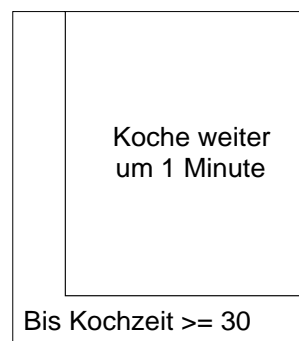
Einseitige
Entscheidung



Fallunterscheidung

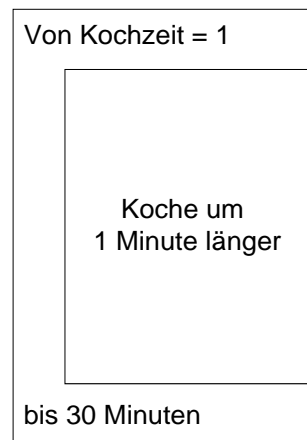


Abweisende
Schleife

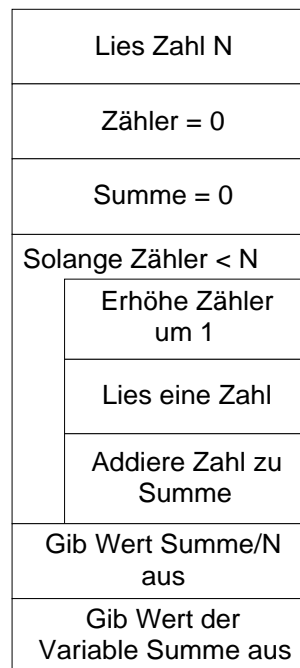


Nicht-abweisende
Schleife

Struktogramme (2/2)



Zählschleife



Gesamtbeispiel

Kontroll- und Datenstrukturen

<i>Grundoperationen der Algorithmen</i>	<i>Beispielhafte Elemente von C</i>
I. Kontrollstrukturen	
ANFANG	{
ENDE	}
ANWEISUNG	Zahl = Zahl + 2
SEQUENZ	<pre> . . . Zahl1 = Zahl2 + 1; Zahl3 = (2 * pi) / 3; . . . </pre>
SCHLEIFE, ITERATION	<pre> while (Kochzeit < 30) koche </pre>
ENTSCHEIDUNG	<pre> if (Kochzeit > 30) Esse; else Warte; </pre>
TEIL-ALGORITHMUS	Brate_Hackfleisch;
II. Datenstrukturen	
EIN-/AUSGABE	scanf(...) / printf (...)
DATENBESCHREIBUNG	<pre> typedef . . . (eigene Beschreibung) int . . . float . . . (vordefinierte Beschreibung) </pre>

3. Syntax und Semantik

3.1. Backus-Naur Form

3.2. Syntax-Diagramme

3.3. Semantik

Syntax-Beschreibung

Backus-Naur Form

Weit verbreitete Notation der Syntax von Programmiersprachen, nach John W. Backus (USA) und Peter Naur (Dänemark).

Abkürzung BNF. Erste Verwendung für ALGOL-60.

$\langle \text{Satz} \rangle ::= \langle \text{Subjekt} \rangle \langle \text{Verb} \rangle \langle \text{Objekt} \rangle.$

"::=" bedeutet: „(die linke Seite) wird definiert als (die rechte Seite)“

"< >" schließt Metazeichen ein, kennzeichnet sie.

Linke Seite wird definiert als Konkatination der Sprachkonstrukte auf der rechten Seite.

Beispiel für obige Definition: "Otto mag Wurst"

Weitere Symbole der BNF

" / " oder " "	Alternative
" . "	Ende der Definition
"[x]"	kein oder genau ein Auftreten von x
"{ x }"	kein, ein oder mehrfaches Auftreten von x
"x y"	Auswahl: x <u>oder</u> y

Beispiel

$\langle \text{Ganze_Zahl} \rangle ::= [+ \mid -] \langle \text{Vorzeichenlose_ganze_Zahl} \rangle.$

$\langle \text{Vorzeichenlose_ganze_Zahl} \rangle ::=$
 $\quad \langle \text{Ziffer} \rangle \mid$
 $\quad \langle \text{Vorzeichenlose_ganze_Zahl} \rangle \langle \text{Ziffer} \rangle.$

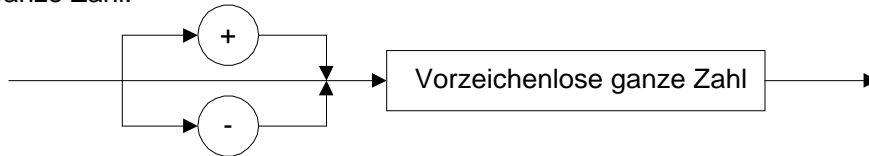
$\langle \text{Ziffer} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$

Syntax-Beschreibung

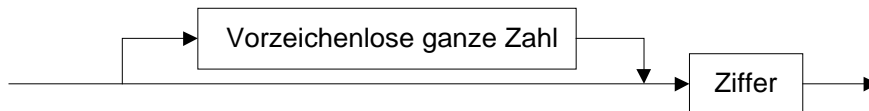
Syntax-Diagramme

Darstellung der syntaktischen Regeln mittels graphischer Hilfsmittel.

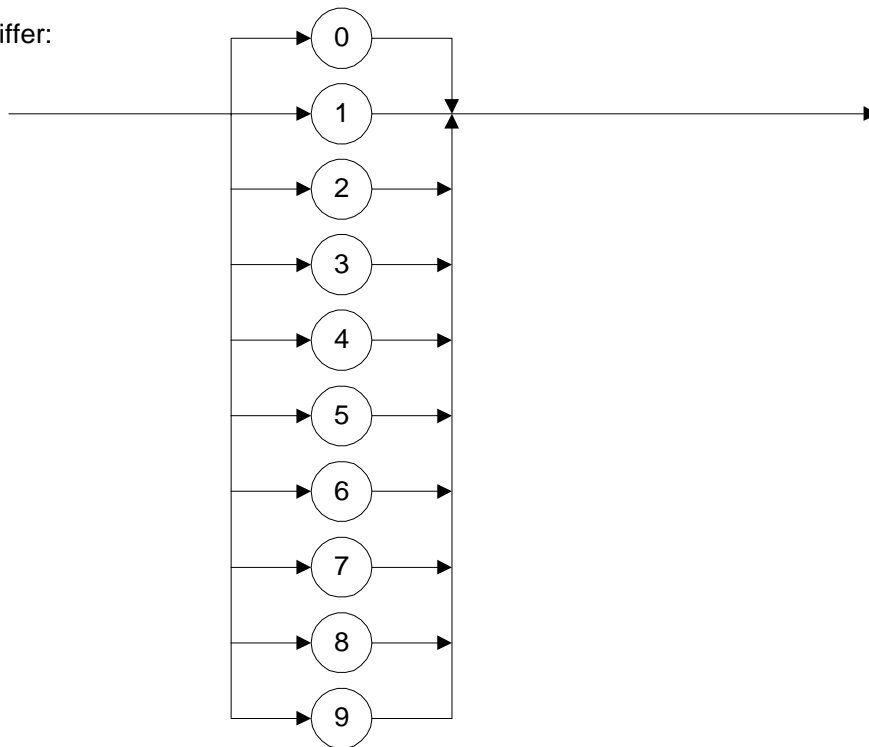
Ganze Zahl:



Vorzeichenlose ganze Zahl:



Ziffer:



Verzweigungen (Weichen): Nur ein Zweig möglich

Rund-gerahmt: "Terminale Symbole", stehen für sich selbst, nicht verfeinerbar. Eckig gerahmt: Stehen für anderes, komplettes Syntaxdiagramm, das mit Namen identifiziert wird ("Nicht-terminale Symbole", "Metasymbole")

Semantik

Semantik einer Sprache =

Beziehungen zwischen Zeichen einer Sprache und deren Bedeutung,
d.h. inhaltliche Bedeutung einer Sprache.

Programme müssen nicht nur **syntaktisch** korrekt, sondern auch
semantisch sinnvoll sein:

Beispiel einer semantisch unsinnigen Funktion, allerdings mit korrekter
Syntax:

```
int unsinn(void)
{
    while ( 0 != 1 ) ;
}
```

Programmiersprache

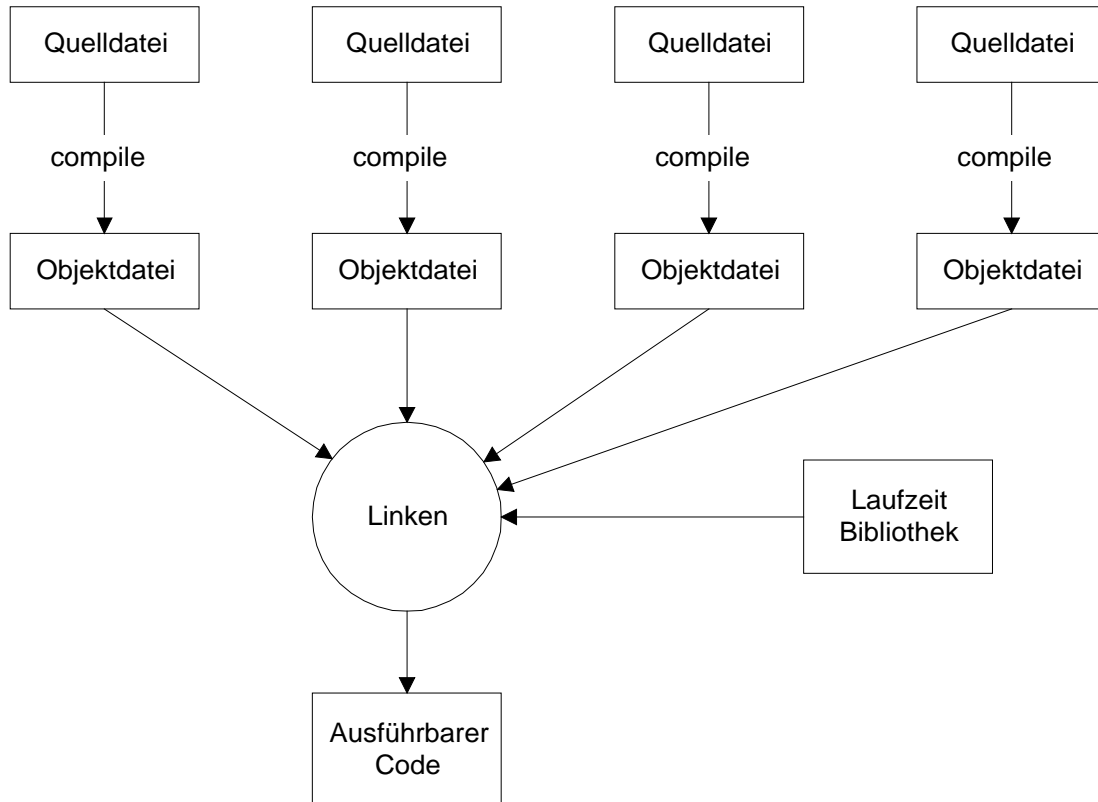
Zusammenfassung

Programmiersprachen sind formale (d.h. formalisierte) Sprachen. Sie
bestehen aus einer Teilmenge der Menge aller Wörter, die aus einer
Menge von Symbolen (dem Alphabet) durch Konkatenation
(Aneinanderreihung) gebildet werden können.

4. Einstieg in C: Einfache Sprachkonstrukte und allgemeiner Programmaufbau

- 4.1. Vom Quellcode zum lauffähigen Programm
- 4.2. Funktionen
- 4.3. Variablen und Konstanten
- 4.4. Namen
- 4.5. Ausdrücke
- 4.6. Formatierung des Quelltextes
- 4.7. Die main() Funktion
- 4.8. Die printf() Funktion
- 4.9. Die scanf() Funktion
- 4.10. Preprozessor

Von Quelldateien zum lauffähigen Programm



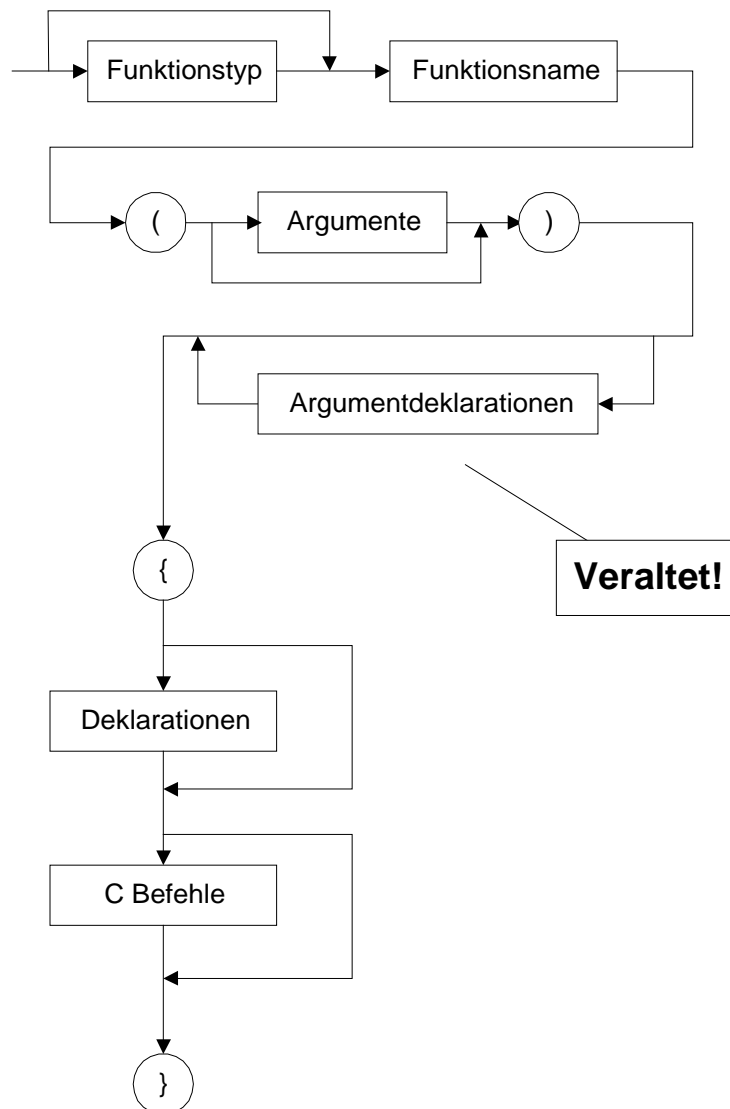
Funktionen

```
int square( int num )
{
    int answer;

    answer = num * num;
    return answer;
}
```

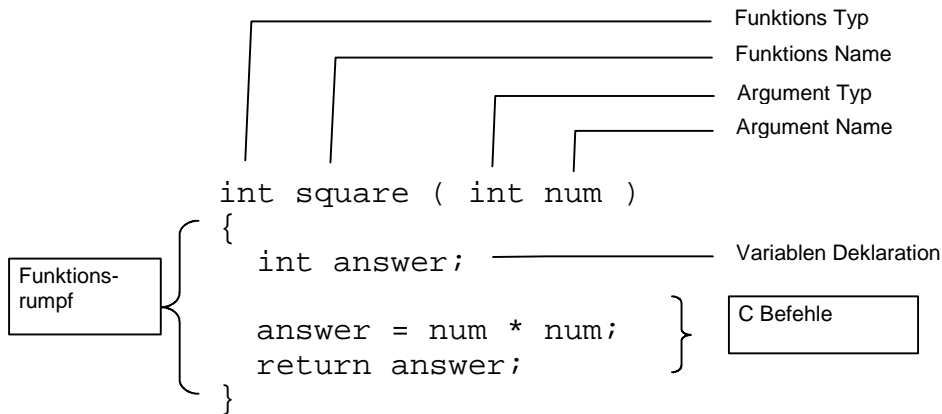
Syntaxdiagramm

Funktion:



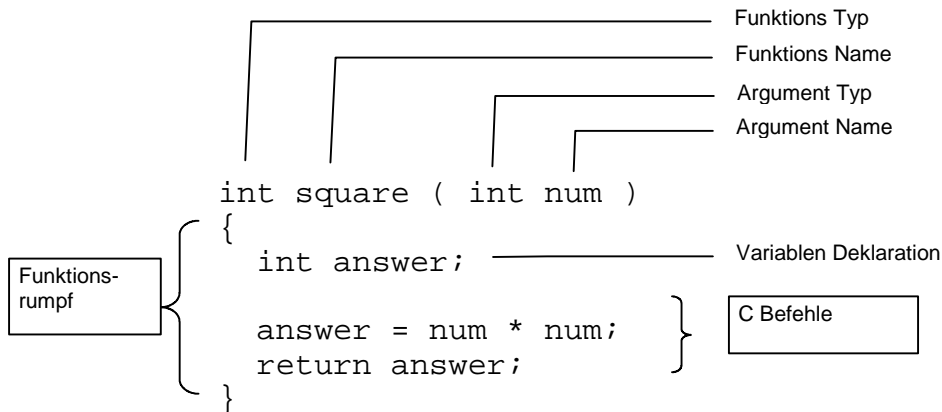
Schnellkurs in C

anhand einer Funktion (1/4)



- *int* in der ersten Zeile ist ein reserviertes Schlüsselwort (*reserved keyword*) und steht für *integer*. Bedeutet, daß die Funktion einen ganzzahligen Wert zurückgeben wird.
- In C gibt es etwa 30 Schlüsselworte; jedes hat eine besondere C-spezifische Bedeutung. Schlüsselworte werden immer *klein* geschrieben und dürfen nicht für andere Zwecke, wie z.B. Bezeichner von Variablen benutzt werden.
- *square* ist der Name (Bezeichner) der Funktion selbst. Wir hätten jeden anderen Namen (außer den Schlüsselworten) verwenden können, grundsätzlich sollte man aber immer *sprechende* Bezeichner benutzen!
- Die folgenden runden Klammern signalisieren, daß *square* eine Funktion ist (und nicht z.B. eine einfache Variable).

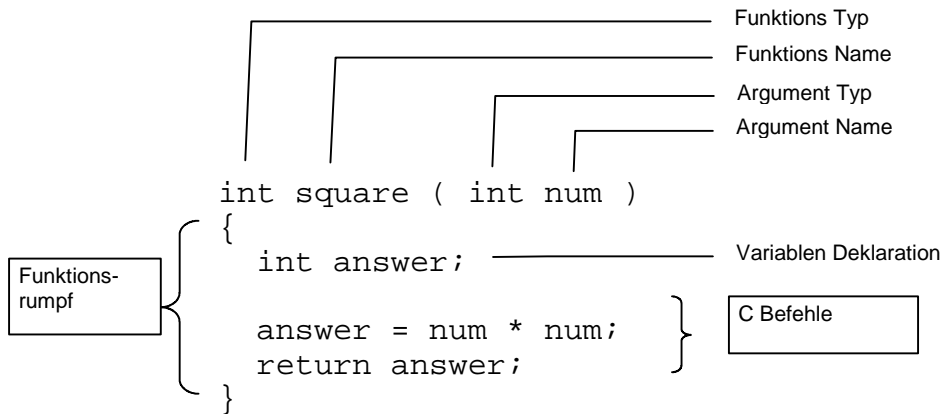
Schnellkurs in C anhand einer Funktion (2/4)



- Diese Funktion hat ein *Argument* mit dem Namen *num* vom Typ *int*.
 - Argumente repräsentieren Daten, die beim Aufruf der Funktion von dem Aufrufer übergeben werden.
 - Auf der aufrufenden Seite heißen sie *aktuelle Argumente*, auf der gerufenen Seite, also bei der Definition der Funktion, *formale Argumente*.
 - Für die Namensgebung gilt das oben gesagte.
- Durch die gleichzeitige Angabe des Typs des Arguments (*int*) weiß der Compiler, daß beim Aufruf von der gerufenen Funktion ein *Integer* Argument erwartet wird. Der Compiler kann also beim Aufruf darauf achten, daß das aktuelle Argument den richtigen Typ besitzt.
- Solche Funktionsdefinitionen bezeichnet man auch als *Funktions-Prototypen*. Wir wollen sie immer verwenden. In diesem Fall entfällt die altertümliche Angabe der Argumentstypen in den Argumentdeklarationen.
- Eine Funktion kann eine beliebige Anzahl von Argumenten besitzen. Bsp.:

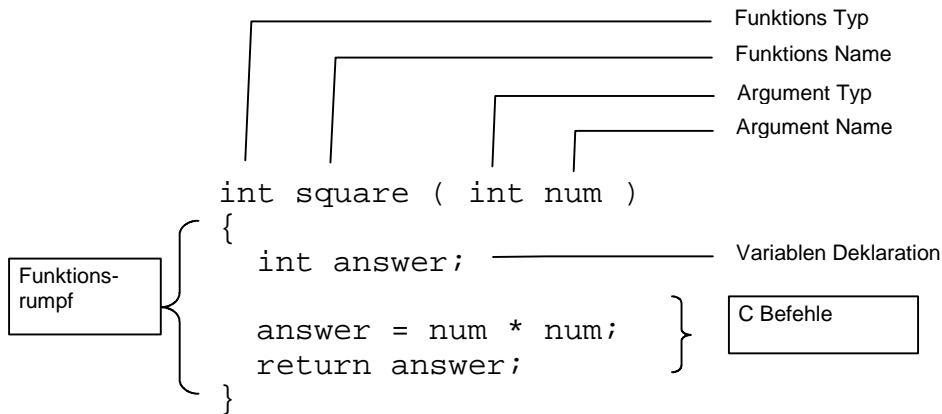
```
int power ( int x, int y );
```

Schnellkurs in C anhand einer Funktion (3/4)



- Der Funktionsrumpf enthält alle ausführbaren Befehle. Hier werden Berechnungen durchgeführt, usw. Der Funktionsrumpf wird durch eine geschweifte Klammer eingeleitet und endet mit einer schließenden geschweiften Klammer.
- In der Zeile nach der öffnenden geschweiften Klammer wird eine – lokale – Integer Variable *answer* definiert.
 - Diese Variable existiert nur so lange (ist nur so lange zugänglich) wie der Code des Funktionsrumpfes abgearbeitet wird.
 - Anschließend ist die Variable nicht mehr definiert.
 - Auch ihr Wert geht verloren.
- Die nächste Zeile ist der erste wirklich ausführbare Befehl (*executable statement*).
- Es ist eine *Wertzuweisung (assignment statement)*.
 - Der Wert des Ausdrucks auf der rechten Seite von dem Gleichheitszeichen wird der Variablen auf der linken Seite zugewiesen.

Schnellkurs in C anhand einer Funktion (4/4)



- Die nächste Zeile enthält den *return* Befehl, welcher bewirkt, daß zu der aufrufenden Funktion zurückgesprungen wird.
- Optional kann dieser Befehl der Funktion ihren Rückgabewert zuweisen, in diesem Fall den Wert von *answer*.

Variablen

Wertzuweisung

Sei die Variable *j* an der Adresse 2486 gespeichert.

Dann bedeutet die Anweisung

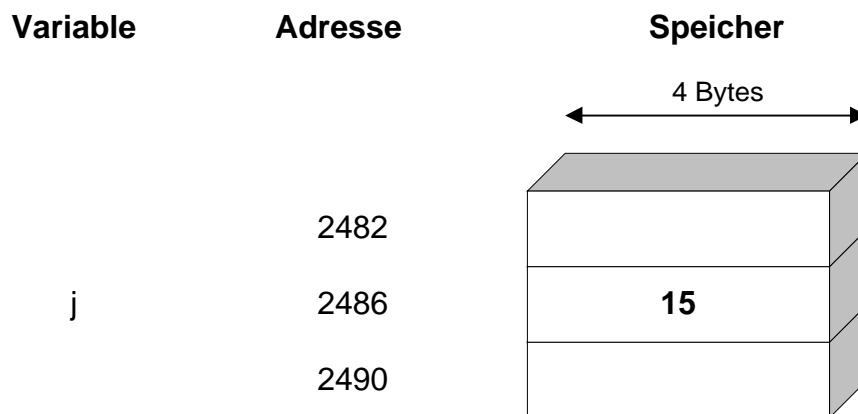
$$j = j - 2;$$

im Detail:

- nimm den Inhalt der Adresse 2486
- ziehe davon (die Konstante) 2 ab
- speichere das Ergebnis wieder in Adresse 2486

Analogie zu Schubladen!

Keine mathematische Gleichung!



Reservierte Schlüsselworte

Reservierte Schlüsselworte in C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Zusätzlich reservierte Schlüsselworte in C++

asm	inline	static_cast
bool	mutable	template
catch	namespace	this
class	new	throw
const_cast	operator	true
delete	private	try
dynamic_cast	protected	typeid
false	public	using
friend	reinterpret_cast	virtual

Bezeichner (Namen)

Reservierte Bezeichner (Namen) identifizieren als Bestandteile der Sprache bestimmte Programm-Elemente (auch: **Schlüsselworte**, **key words**). Sie können nicht umdefiniert werden:

if, while, int, typedef, switch, ...

Standardbezeichner (Standardnamen) kennzeichnen vordefinierte Typen, Konstanten, Variablen, Funktionen und Funktionen der Laufzeitbibliothek. Sie können im Prinzip umdefiniert werden. Dann stehen jedoch die vordefinierten Funktionen etc. **nicht** mehr zur Verfügung:

exp, sin, getchar, printf, scanf, exit, localtime, ...

Selbstdefinierte Bezeichner (Namen) werden durch den Programmierer zur Kennzeichnung seiner Variablen, Konstanten, Funktionen, etc. eingeführt:

laenge, zahl2, oberkante, ...

Ausdrücke

5	Eine Konstante
j	Eine Variable
5 + j	Eine Konstante plus eine Variable
5 * j + 6	Eine Konstante mal eine Variable plus eine Konstante
f ()	Ein Funktionsaufruf
f () / 4	Ein Funktionsaufruf, dessen Ergebnis durch eine Konstante geteilt wird

Die wichtigsten Bausteine von Ausdrücken sind

- Variablen,
- Konstanten und
- Funktionsaufrufe (es gibt noch mehr!).

Diese

- sind selbst bereits *Ausdrücke* und
- werden durch *Operatoren* miteinander zu *komplexeren Ausdrücken* verknüpft.

Die grundlegendsten Operatoren sind:

- + Addition
- Subtraktion
- * Multiplikation
- / Division

Formatierung des Quelltextes

Schlecht!

```
int square (int num) {int answer;  
answer = num*num; return answer; }
```

Schlecht!

```
int  
square (int      num)  
{  
  int      answer;  
  answer   =      num  
  *        num;  
  return   answer; }
```

Gut,
zusätzlich mit Kommentaren

```
/*  
 * Autor: Fritz Müller  
 * Erste Erstellung: 10.10.1997  
 * Funktion zur Berechnung des Quadrates zweier Integer Zahlen  
 */  
  
int square( int num )  
{  
    int answer;  
  
    answer = num * num; /* Überlauf wird nicht abgefangen */  
    return answer;  
}
```

Kommentare

Empfohlen	Vermeiden
<ul style="list-style-type: none">• Beschreibe im Funktionskopf, was die Funktion tut• Erläutere komplexere Operationen oder Algorithmen, denen man ihren Zweck nicht sofort ansieht• Gliedere den Programmtext durch Kommentare• Falls guter Programmierstil (aus guten Gründen) an einer Stelle verletzt werden muß, erläutere es in einem Kommentar• Markiere noch nicht fertiggestellte Teile eines Programms mit Kommentaren während der Entwicklung• Verwende Kommentare, um notwendige Änderungen am Code zu markieren• Wenn der Name einer Variablen oder Konstanten ihren genauen Zweck noch nicht verrät, ergänze dies durch einen Kommentar	<ul style="list-style-type: none">• Versuche zuerst gut leserlichen Code zu schreiben (Wahl der Bezeichner, Wahl der Kontrollstrukturen), so daß Kommentare überflüssig sind• Wiederhole im Kommentar nicht offensichtlichen Code• Wähle lieber sprechende Namen für Variablen als einfache Namen durch Kommentare zu ergänzen

main() Funktion

```
#include <stdlib.h>

int main(void)
{
    extern int square(int n);    /* Funktionsprototyp */
    int solution;

    solution = square(5);
    exit(0);                    /* oder return 0 */
}
```

Mit Ausgabe

```
#include <stdio>
#include <stdlib.h>

int main(void)
{
    extern int square(int n);
    int solution;

    solution = square(27);
    printf("Das Quadrat von 27 ist %d\n", solution);
    exit(0);                    /* oder return 0 */
}
```

Mit Eingabe und Ausgabe

```
#include <stdio>
#include <stdlib.h>

int main(void)
{
    extern int square(int n);
    int solution;
    int input_val;

    printf("Gib eine Integer Zahl ein: ");
    scanf("%d", &input_val);
    solution = square(input_val);
    printf("Das Quadrat von %d ist %d\n", input_val, solution);
    exit(0);
}
```

printf() Funktion

```
printf("Das Ergebnis ist %d", num);
```

Umwandlungsangaben (*format specifier*) als Auswahl

Format specifier	Bedeutung
%d	Dezimaler Integerwert
%x	Hexadezimaler Integerwert
%o	Oktaler Integerwert
%f	Gleitkommazahl (floating point)
%c	ein Zeichen (character)
%s	String (Null-terminiertes Character Array)

Zusätzliche Möglichkeiten

- links- oder rechtsbündige Darstellung
- feste Spaltenbreite (mit Blanks aufgefüllt)
- „+“-Zeichen wird bei positiven Zahlen gedruckt
- Angabe der Stellen hinter dem Dezimalpunkt

Beispiele

```
printf("Drei Werte: %d %d %d", num1, num2, num3);
```

```
printf("Das Quadrat von %d ist %d\n", num, num*num);
```

„\“, Escape-Zeichen.

\n neue Zeile (*new line*).

Fortsetzungszeichen \

```
printf("Dieser Text ist zu lang, um in eine \  
Zeile zu passen. Daher müssen Fortsetzungszeichen \  
verwendet werden!");
```

Laut ANSI-Standard kann man Fortsetzungszeichen sogar dazu einsetzen, um Variablennamen in die nächste Zeile umzubrechen. Dies wird jedoch nicht empfohlen!

Konkatenierung von Zeichenketten

Laut ANSI-Standard ist für Zeichenketten auch die folgende Notation möglich:

```
printf("Dieser Text ist zu lang, um in eine "  
      "Zeile zu passen. Daher müssen Fortsetzungszeichen "  
      "verwendet werden!");
```

Das gilt allgemein für Zeichenketten:

```
"hello, world"
```

und

```
"hello," " world"
```

sind gleichwertig.

scanf() Funktion

Gegenstück zu *printf()*

Beispiel

```
scanf("%d %d", &num1, &num2);
```

Parameter müssen *lvalues* sein.

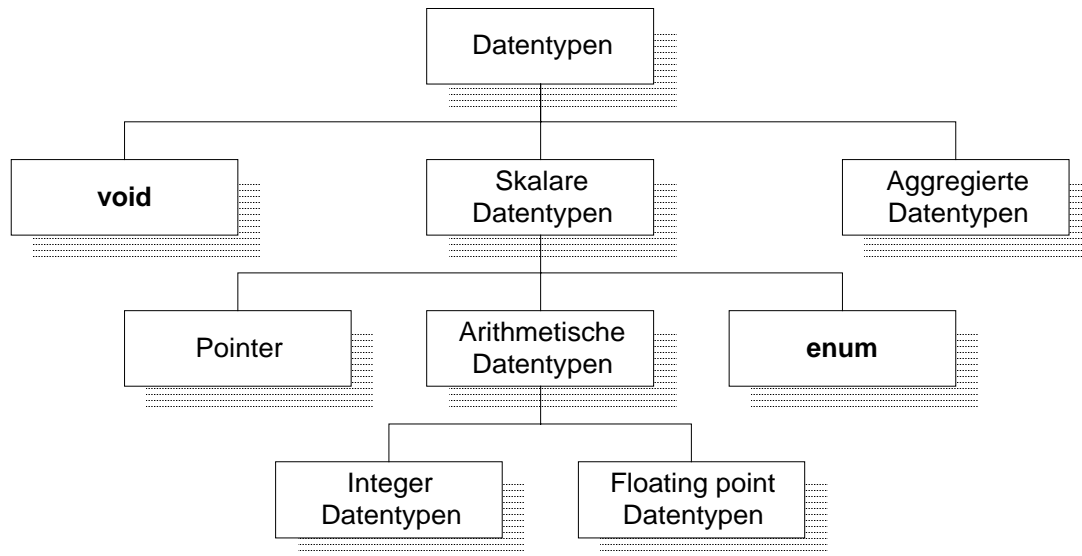
Da sie Rückgabewerte sind, müssen über den vorgestellten Adreßoperator & die Adressen der Parameterübergeben werden.



5. Skalare Standard-Datentypen

- 5.1. Variablendeklaration
- 5.2. Integer Konstanten
- 5.3. Gleitkomma Datentypen
- 5.4. Initialisierung von Variablen
- 5.5. Mixed Mode Arithmetik
- 5.6. Explizite Typkonversion – Type Casts
- 5.7. Aufzählungstypen
- 5.8. Der void Datentyp
- 5.9. typedef Typendeklaration
- 5.10. Adressen von Datenobjekten
- 5.11. Einführung in Zeigervariable – Pointer

C Datentypen



Variablen

Wertzuweisung

```
Guthaben    = 500;  
Einzahlung  = 100;  
Guthaben    = Guthaben + Einzahlung
```

ACHTUNG: Vor der ersten Wertzuweisung hat die Variable keinen definierten Wert!

Deklarationen

```
int    Zahl, Weg  { Integer Typ }  
float  Lohn;      { Gleitkomma Typ }  
int    Flag;      { Boolescher Typ, in C: Integer (0 oder 1) }  
int    i, j, k;  
int    n, m;  
float  x, y, z;
```

Skalare Basisdatentypen

<i>Schlüssel- wort</i>	<i>Bedeutung</i>
char	ein 8 Bit Zeichen, d.g. 1 Byte
int	Integer, meist 32 Bit (natürliche Größe der CPU)
float	Gleitkommazahl, einfach genau
double	Gleitkommazahl, doppelt genau
enum	Aufzählungsdentyp

Qualifier

Können mit den Basistypen kombiniert werden

<i>Schlüssel- wort</i>	<i>Bedeutung</i>
short	geringere Genauigkeit, min. 16 Bit für Integers
long	höhere Genauigkeit, min. 32 Bit für Integers besonders hohe Genauigkeit bei Gleitkommazahlen (long double)
signed	mit Vorzeichen
unsigned	ohne Vorzeichen

<limits.h> und <float.h> definieren symbolische Konstanten der in der aktuellen Implementierung verwendeten Grenzwerte.

Ausschnitt aus <limits.h> für Intel-Rechner

```
#define CHAR_BIT 8
#define CHAR_MAX 127
#define CHAR_MIN (-128)
#define INT_MAX 2147483647
#define INT_MIN (-2147483647-1)
#define LONG_MAX 2147483647L
#define LONG_MIN (-2147483647L-1L)
#define MB_LEN_MAX 5
#define SCHAR_MAX 127
#define SCHAR_MIN (-128)
#define SHRT_MAX 32767
#define SHRT_MIN (-32768)
#define UCHAR_MAX 255
#define UINT_MAX 4294967295U
#define ULONG_MAX 4294967295UL
#define USHRT_MAX 65535
```

Ausschnitt aus <float.h> für Intel-Rechner

```
#define FLT_DIG 6
#define FLT_MANT_DIG 24
#define FLT_MAX_10_EXP 38
#define FLT_MAX_EXP 128
#define FLT_MIN_10_EXP (-37)
#define FLT_MIN_EXP (-125)
...
#define DBL_DIG 15
#define DBL_MANT_DIG 53
#define DBL_MAX_10_EXP 308
#define DBL_MAX_EXP 1024
#define DBL_MIN_10_EXP (-307)
#define DBL_MIN_EXP (-1021)
...
#define LDBL_DIG 18
#define LDBL_MANT_DIG 64
#define LDBL_MAX_10_EXP 4932
#define LDBL_MAX_EXP 16384
#define LDBL_MIN_10_EXP (-4931)
#define LDBL_MIN_EXP (-16381)
```

Character

Numerischer Code des Characters

```
/* Drucke den numerischen Code eines ASCII Zeichens */  
  
#include <stdio.h>  
  
int main(void)  
{  
    char ch;  
  
    printf("Gib Zeichen: ");  
    scanf("%c", &ch);          / Lies Zeichen als solches ein */  
    printf("Sein numerischer Code ist %d\n", ch);  
    return 0;  
}
```


Integer Konstanten

Darstellung in anderer Zahlenbasis

<i>Dezimal</i>	<i>Oktal</i>	<i>Hexadezimal</i>
3	003	0x3
8	010	0x8
15	017	0xF
16	020	0x10
21	025	0x15
-87	-0127	-0x57
187	0273	0xBB
255	0377	0xff

Zugehörigen Formatkonversionen:

 %o für oktale,

 %x für hexadezimale Darstellung.

Programmbeispiel

```
/* Drucke die dezimale und oktale Darstellung
 * einer hexadezimalen Konstanten.
 */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int num;

    printf("Gib eine hexadezimale Konstante ein: ");
    scanf("%x", &num);
    printf("Das dezimale Äquivalent von %x ist: %d\n",
           num, num);
    printf("Das oktale Äquivalent von %x ist: %o\n",
           num, num);
    exit(0);
}
```

Integer Konstanten

Wahl des Datentyps

Der erste passende Typ der rechten Spalte wird gewählt:

<i>Form der Konstanten</i>	<i>Liste der möglichen Datentypen</i>
Dezimal, ohne Suffix	int, long int, unsigned long int
Oktal oder hexadezimal, ohne Suffix	int, unsigned int, long int, unsigned long int
Mit Suffix u oder U	unsigned int, unsigned long int
Mit Suffix l oder L	long int, unsigned long int

Escape Sequenzen

<i>Escape Sequenz</i>	<i>Name</i>	<i>Bedeutung</i>
\a	alert	Erzeugt ein hör- oder sichtbares Signal
\b	backspace	Bewegt den Cursor um ein Zeichen zurück
\f	form feed	Bewegt den Cursor zur nächsten logischen Seite
\n	new line	Zeilenwechsel
\r	carriage return	Wagenrücklauf
\v	vertical tab	Vertikaler Tabulator
\\	backslash	Druckt einen \
\'	single quote	Druckt einfaches Anführungszeichen
\"	double quote	Druckt doppeltes Anführungszeichen
\?	Question mark	Druckt Fragezeichen

Beispiele für oktale oder hexadez. Escapesequenzen:

```
\141      /* oktaler Code des ASCII Zeichens A      */
\x61      /* hexadezimaler Code des ASCII Zeichens A */
\0        /* der 0-Character                          */
```

Gleitkomma Datentypen

<i>Datentyp</i>	<i>Wertebereich</i>	<i>Dezimalstellen</i>	<i>Speicherbedarf</i>
float	etwa $E_{\pm 37}$	6	4 Bytes
double	etwa $E_{\pm 307}$	15	8 Bytes
long double	etwa $E_{\pm 4931}$	18	10 Bytes

Beispiele

```
float      pi;  
double     pi_squared;  
long double pi_cubed;  
  
pi         = 3.141;  
pi_squared = pi * pi;  
pi_cubed   = pi * pi * pi;
```

Gleitkomma Datentypen

Konvertiere Fahrenheit nach Celsius

```
/*
 * Konvertiere Fahrenheit nach Celsius
 */

double fahrenheit_to_celsius(double temp_fahrenheit)
{
    double temp_celsius;

    temp_celsius = (temp_fahrenheit - 32.0) * 100.0 /
                   (212.0 - 32.0);
    return temp_celsius;
}
```

Berechne die Fläche eines Kreises

```
/*
 * Berechne die Fläche eines Kreises bei gegebenem Radius
 */

#define PI 3.14159

float flaeche_von_kreis(float radius);
{
    float flaeche;

    flaeche = PI * radius * radius;
    return flaeche;
}
```

Implizite Konversionen

Regeln

quiet conversions, automatic conversions

1. Bei Wertzuweisungen wird der Wert des Ausdrucks auf der rechten Seite in den Datentyp der Variablen auf der linken Seite konvertiert (*assignment conversion*).
2. Ein *char* oder *short int* in einem Ausdruck wird in einen *int* konvertiert.

unsigned char und *unsigned short* werden zu *int* konvertiert, wenn genügend Bits zur Verfügung stehen, sonst zu *unsigned int*.

Dies sind *integral widening* oder *integral promotion conversions*.

3. In arithmetischen Ausdrücken werden Objekte so umgewandelt, daß sie mit den Konversionsregeln der Operatoren konform gehen (werden wir später sehen).
4. Unter bestimmten Umständen werden Argumente von Funktionen umgewandelt (wir werden auch das später behandeln).

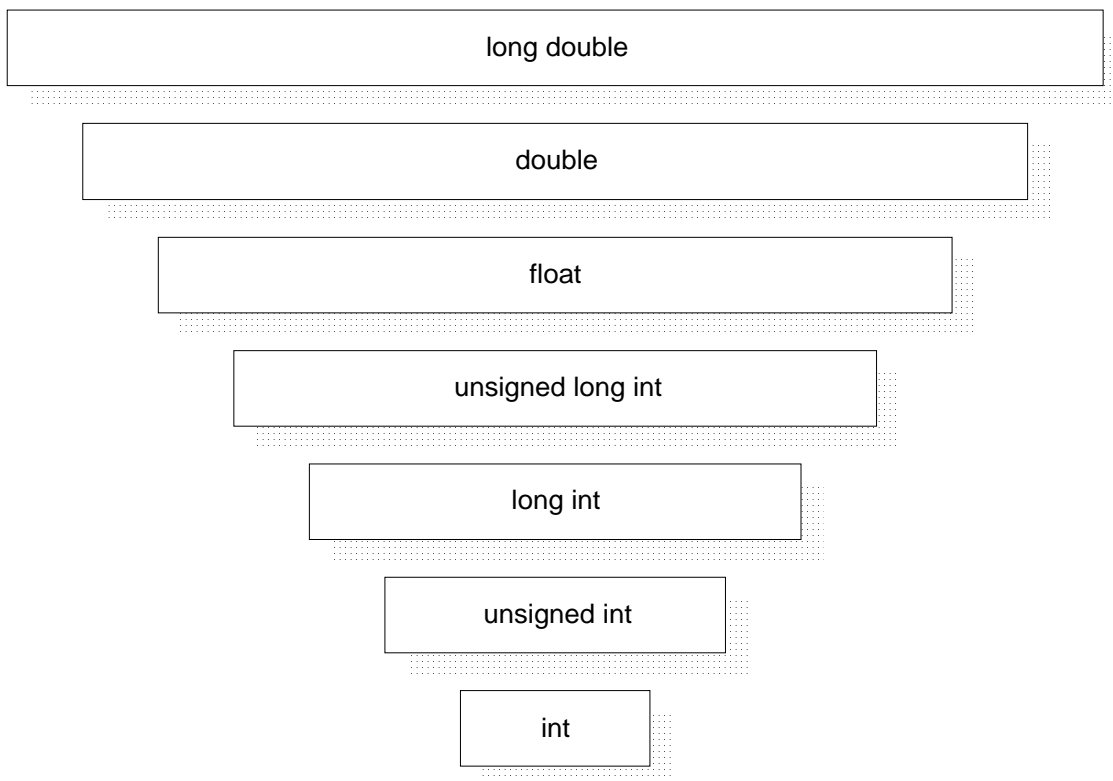
Implizite Konversionen

In Ausdrücken

$-3 / 4 + 2.5$

Operator	Operand(en)	Operatortyp
-	3	unärer
/	-3 und 4	binärer
+	-3/4 und 2.5	binärer

- Binäre Operatoren erwarten i.d.R. Operanden vom selben Datentyp.
- Bei Unterschieden geschieht eine *implizite Konversion* eines der beiden Operanden.
- Dabei wird der niedrigwertigere in den nächst höheren Datentyp konvertiert:



Mischen von Integer Datentypen

Vier „Größen,, von Integers: `char`, `short`, `int`, `long`

Integer-Erweiterung (*integral promotion*)

char und *short* immer \Rightarrow *int*

Daher steht *int* in obiger Abbildung am unteren Fuß der Pyramide.

Ausnahme:

short hat in aktueller Implementierung ebenso viele Bits wie *int*

\Rightarrow *unsigned short* kann nicht immer in einen *int* umgewandelt werden

\Rightarrow Umwandlung in *unsigned int*!

Overflows

Sei *c* ein *char*.

```
c = 882;
```

Resultat ist Overflow, da ein *char* nur Werte zwischen -128 und $+127$ annehmen kann (*signed char*).

Folgen eines solchen Overflows laut ANSI Standard:

- Für *signed types*: Undefiniertes Ergebnis. Üblich: Oberste(s) Byte(s) wird/werden abgeschnitten.
- Für *unsigned types*: Ergebnis ist der ursprüngliche Wert modulo der größten darstellbaren Zahl plus 1 des Ziels der linken Seite. Dies entspricht ebenfalls dem Weglassen des/der obersten Bytes.

Konversion von unsigned Integers

Konversion von *unsigned chars* und *unsigned shorts*?

Erweiterung nach *ints* oder *unsigned ints*?

Sign-preserving Strategy

Konversion nach *unsigned int*.

Problem: Sei *a* ein *unsigned short int* mit dem Wert 2. Dann ergibt

$$a - 3$$

nicht -1 sondern einen sehr großen positiven Wert!

ANSI: Value-preserving Strategy

unsigned chars und *unsigned shorts* \Rightarrow *ints*

Annahme, *int* hat mehr Bits als die anderen beiden Typen

Ist *int* nicht größer \Rightarrow Konversion nach *unsigned int*.

Mischen von Gleitkommawerten

Drei „Größen,, von Gleitkommatypen: float, double, long double

Erweiterung zum breiteren Typ

Performance-Verluste zur Laufzeit sind möglich

Mögliche Probleme:

Wertzuweisungen von breiteren Typen auf schmalere

- Verlust in der Darstellungsgenauigkeit
- Overflows oder Underflows

Sei *f* eine *float* Variable deren maximaler Wert bei 1.0e38 liegen möge.

```
f = 5.0e40;
```

erzeugt dann einen Überlauf.

Nach ANSI ist das Verhalten dabei nicht definiert.

Vernünftige Compiler erzeugen einen Laufzeit-Fehler (*run-time error*).
Jedoch bei weitem nicht alle.

Oft bei Underflow als Ergebnis 0, bei Overflow *Inf*.

Mischen von Integern mit Gleitkommawerten

Integer nach Gleitkomma

I.a. kein Problem

Es kann Genauigkeit verloren gehen:

```
#include <stdio.h>
int main(void)
{
    long int j = 2147483600;
    float x;

    x = j;
    printf("j ist %d\nx ist %10f\n", j, x);

    exit(0);
}
```

Gleitkomma nach Integer

Dezimalbruchanteil wird abgeschnitten. Keine Rundung!

```
j = 2.5;    /* j ist 2 */
j = -5.8;   /* j ist -5 */
```

Achtung Rundungsfehler:

Nach längerer Rechnung \Rightarrow 94.9999987.

Zuweisung an Integer-Variable \Rightarrow 94 (nicht 95)!

Auch Überlauf möglich:

```
j = 1.567e20;
```

Mischen von signed und unsigned Integers

Regel

- Wenn einer der beiden Operanden *unsigned long int* ist, wird der andere in *unsigned long int* umgewandelt.
- Andernfalls, wenn ein Operand *long int* ist und der andere *unsigned int*, hängt das Ergebnis davon ab, ob *long int* alle Werte von *unsigned int* darstellen kann:
 - Falls ja, wird der eine Operand von *unsigned int* in *long int* umgewandelt;
 - falls nein, werden beide in *unsigned long int* umgewandelt.
- Andernfalls, wenn ein Operand *long int* ist, wird der andere in *long int* umgewandelt.
- Andernfalls, wenn einer der beiden Operanden *unsigned int* ist, wird der andere in *unsigned int* umgewandelt.
- Andernfalls haben beide Operanden den Typ *int*.

Beispiel

```
int main(void)
{
    unsigned int jj;
    int k;

    jj = 5;
    k = 10;
    if (jj-k < 0)          /* Bedingung kann niemals */
                          /* erfüllt werden,        */
        printf("xxx");    /* da k in unsigned int   */
                          /* konvertiert wird!      */
    exit(0);
}
```

Explizite Typkonversion – Type Casts

```
j = (float) 2;
```

Beispiel

```
int    j = 2,  
       k = 3;  
float f;  
  
f = k/j;    /* Ergibt f = 1.0! */
```

Abhilfe

```
f = (float) k/j;    /* Cast bezieht sich nur auf k! */
```

Unser obiges Beispiel

```
int main(void)  
{  
    unsigned int jj;  
    int k;  
  
    jj = 5;  
    k = 10;  
    if ((long int)jj-k < 0)    /* Erzwingte Auswertung */  
                                /* mit Vorzeichen! */  
        printf("xxx");  
    exit(0);  
}
```

Aufzählungstypen

Enumerated Type. Gibt es auch in Pascal.

Definition zweier Variablen *farbe* und *intensitaet*:

```
enum { rot, blau, gruen, gelb } farbe;  
enum { hell, mittel, dunkel } intensitaet;
```

Beispiel-Programm

```
int main(void)  
{  
    enum { rot, blau, gruen, gelb } farbe;  
    enum { hell, mittel, dunkel } intensitaet;  
  
    farbe = gelb;  
    farbe = hell;          /* Typkonflikt */  
    intensitaet = hell;  
    intensitaet = blau;    /* Typkonflikt */  
    farbe = 1;             /* Typkonflikt */  
    farbe = blau + gruen;  /* Falscher Gebrauch */  
  
    exit(0);  
}
```

Adressen von Datenobjekten

Adreßoperator &

```
ptr = &j;
```

„Weise der Variablen *ptr* die Adresse von *j* zu.“

Beispiel

```
#include <stdio.h>

int main(void)
{
    int j = 1;

    printf("Der Wert von j ist %d\n", j);
    printf("Die Adresse von j ist %p\n", &j);
    exit(0);
}
```

Konversionsanweisung %p für Adressen.

Adreßoperator kann nicht auf der linken Seite einer Anweisung stehen:

```
&x = 1000;    /* ist illegal! */
```

Zeigervariablen

Beispiel

```
long *ptr;
long long_var;

ptr = &long_var;    /* ptr enthält die Adresse von long_var */
```

Falsch

```
long *ptr;
float float_var;

ptr = &float_var;    /* ILLEGAL: ptr kann nur die Adresse
                       einer long Variablen aufnehmen! */
```

Pointer- versus Integer-Variablen

```
#include <stdio.h>

int main(void)
{
    int j = 1;
    int *pj;

    pj = &j;    /* pj erhält Adresse von j */
    printf("Der Wert von j ist %d\n", j);
    printf("Die Adresse von j ist %p\n", pj);
    exit(0);
}
```

Dereferenzieren von Pointern

```
#include <stdio.h>

int main(void)
{
    char *p_ch;
    char ch1 = 'A', ch2;

    printf("Die Adresse von p_ch ist %p\n", &p_ch);

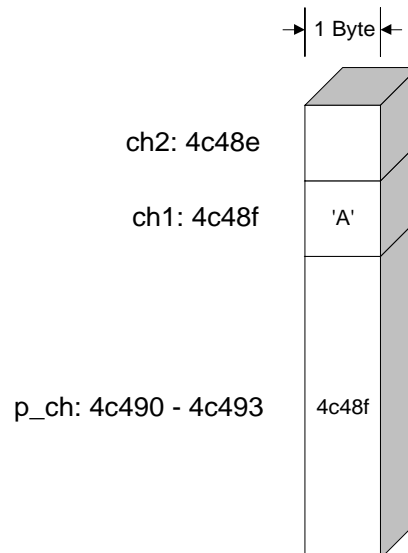
    p_ch = &ch1;
    printf("Der in p_ch gespeicherte Adreßwert ist %p\n", p_ch);
    printf("Der dereferenzierte Wert von p_ch ist %c\n", *p_ch);
    ch2 = *p_ch;

    exit(0);
}
```

Ausgabe

Die Adresse von p_ch ist 4c490
Der in p_ch gespeicherte Adreßwert ist 4c48f
Der dereferenzierte Wert von p_ch ist A

Speicherzuordnung



**p_ch* bedeutet:

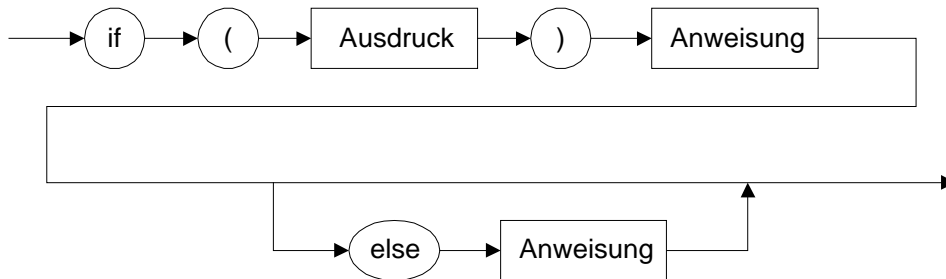
„Nimm den Wert in *p_ch*, interpretiere ihn als Adresse und holen aus dieser Adresse den Wert als Ergebnis dieses Ausdrucks“

6. Kontrollfluß

- 6.1. Bedingte Anweisung (binäre Entscheidung)
- 6.2. Fallunterscheidung (switch)
- 6.3. Wiederholungen, Iterationen, „Loops“
- 6.4. Abweisende Schleife (while)
- 6.5. Nichtabweisende Schleife (do...while)
- 6.6. Zählschleife (for)
- 6.7. Verschachtelte Schleifen
- 6.8. break und continue Anweisungen
- 6.9. goto Anweisung
- 6.10. Endlos-Schleifen
- 6.11. Fehler bei der Verwendung von Schleifenkonstrukten

Entscheidungs-Anweisung

Entscheidung:



Beispiele

Einfache Entscheidung

```
if (x)
    anweisung1; /* Ausgeführt, wenn x ungleich 0 */
anweisung2;    /* Immer ausgeführt */
```

Binäre Entscheidung

```
if (x)
    anweisung1; /* Ausgeführt, wenn x ungleich 0 */
else
    anweisung2; /* Ausgeführt, wenn x gleich 0 */
anweisung3;    /* Immer ausgeführt */
```

Entscheidungs-Anweisung

Vermeide Aufruf von *sqrt* mit negativem Argument

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h> /* Include file for sqrt() */

int main( void )
{
    double num;

    printf( "Enter a non negative number: " );
    /* The %lf conversion specifier indicates a
     * data object of type double.
     */
    scanf( "%lf", &num);
    if (num < 0)
        printf( "Input Error: Number is negative.\n" );
    else
        printf( "The square root is: %f\n", sqrt( num ));
    exit( 0 );
}
```

Vorsicht bei falsch
positioniertem Semikolon

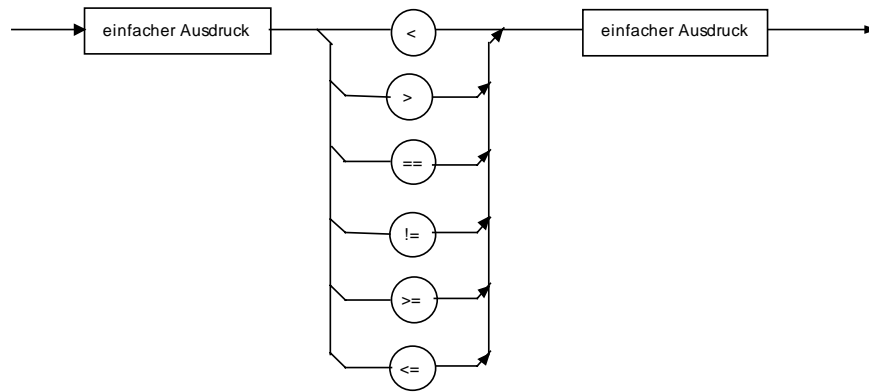
```
if (j==1);    /* Das ist eine leere Anweisung */
    j = 0;    /* Wird IMMER ausgeführt      */
```

Richtig:

```
if (j==1)
    j = 0;    /* Wird ausgeführt, wenn j gleich 1 */
```

Vergleichsausdrücke

Syntaxdiagramm



Logischen Operatoren in C

Vergleichsausdrücke

Häufig angewandte Konstruktion in C

```
if ( func() )
    mache weiter;
else
    Fehlerbehandlung;
```

Vollständiges Beispiel

```
#include <stdio.h>
#include <ctype.h> /* included for isalpha() */
#include <stdlib.h>

int main( void )
{
    char ch;

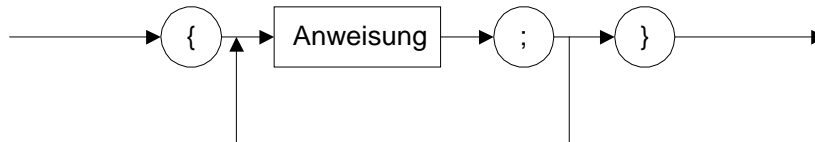
    printf( "Enter a character: " );
    scanf( "%c", &ch );
    if ( isalpha ( ch ) )
        printf( "%c", ch );
    else
        printf( "%c is not an alphabetic character.\n", ch );
    exit( 0 );
}
```

Zusammengesetzte Anweisungen

Compound statements

Jede Anweisung kann durch eine zusammengesetzte Anweisung, eingerahmt in {...} ersetzt werden.

Zusammengesetzte Anweisung (vereinfacht):



Beispiel

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    double num;

    printf( "Enter a non negative number: " );
    scanf( "%lf", &num);
    if (num < 0)
        printf( "That's not a non negative number !\n" );
    else
    {
        printf( "%f squared is: %f\n", num, num*num );
        printf( "%f cubed is: %f\n", num, num*num*num );
    }
    exit( 0 );
}
```

Verschachtelte if-Anweisungen

Nested if-statements

Zweifache Verschachtelung

```
if (x > y)
    printf("x ist größer als y");
else
    if (x < y)
        printf("x ist kleiner als y");
    else
        printf("x ist gleich y");
```

Keine tiefere Einrückung aus Gründen der Übersicht

```
if (x > y)
    printf("x ist größer als y");
else if (x < y)
    printf("x ist kleiner als y");
else
    printf("x ist gleich y");
```

Verschachtelte if-Anweisungen

Vorsicht mit „Zugehörigkeit“, von *e/se*-Klauseln!

Was gilt hier?

```
if (x >= y)
  if (x == y)
    printf("x ist gleich y");
  else
    printf("x ist größer als y");
oder ...
    printf("x ist kleiner als y");
```

Semantische Regel

Ein „verwaister“, *e/se*-Teil wird zur **nahesten**, innersten if-Anweisung zugerechnet („adoptiert“), die sonst ohne *e/se*-Teil wäre.

Klammerung unterstützt Lesbarkeit

```
if (x >= y)
{
  if (x == y)
    printf("x ist gleich y");
  else
    printf("x ist größer als y");
}
```

Mehrfache Verschachtelung zur Fallunterscheidung

```
if (i==1) ... ;
else if (i==3) ... ;
else if (i==6) ... ;
else if (i==10) ... ;
...
else ... ;
```


Switch-Anweisung (1/4)

Fallunterscheidung

Als if-Anweisung

```
if (monat == 1) tage = 31;  
else if (monat == 2) ...;  
...  
else if (monat == 11) tage = 20;  
else tage = 31;
```

Stattdessen als switch-Anweisung

```
switch (monat)  
{  
    case 1:  tage = 31; break;  
    case 2:  if (jahr MOD 4 != 0) ... ; break;  
    case 3:  tage = 31; break;  
    ...  
    case 12: tage = 31; break;  
    default: tage = -1; break;  
}
```

Switch-Anweisung (2/4)

Gleiche Aktion für mehrere Fälle

```
switch (monat)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12: tage = 31; break;

    case 4:
    case 6:
    case 9:
    case 11: tage = 30; break;

    case 2: if (...)
            {
                ...
            }
            break;

    default: printf("Falscher Wert von Monat\n");
            tage = -1;
            break;
}
```

Switch-Anweisung (3/4)

```
/* Print error message based on error_code. Function is
 * declared void because it has no return value.
 */

#include <stdio.h>
#define ERR_INPUT_VAL 1
#define ERR_OPERAND 2
#define ERR_OPERATOR 3
#define ERR_TYPE 4

void print_error( int error_code )
{
    switch (error_code)
    {
        case ERR_INPUT_VAL:
            printf("Error: Illegal input value.\n");
            break;
        case ERR_OPERAND:
            printf("Error: Illegal operand.\n");
            break;
        case ERR_OPERATOR:
            printf("Error: Unknown operator.\n");
            break;
        case ERR_TYPE:
            printf("Error: Incompatible data.\n");
            break;
        default:
            printf("Error: Unknown error code %d\n",
                  error_code);
            break;
    }
}
```

**Definiere
Aufzählungskonstanten statt
#define-Konstanten**

```
typedef enum { ERR_INPUT_VAL,
               ERR_OPERAND,
               ERR_OPERATOR,
               ERR_TYPE;
               } ERROR_SET;

ERROR_SET error_code;
```

Switch-Anweisung (4/4)

Funktion mit 3 Argumenten: 2 Operanden und 1 Operator.

Aufgrund des Operators werden verschiedene Aktionen durchgeführt:

```
/* This function evaluates an expression, given
 * the two operands and the operator.
 */

#include <stdlib.h>
#include "err.h" /* contains the typedef
                  * declaration of ERR_CODE.
                  */

double evaluate( double op1, double op2, char operator )
{
    extern void print_error ();

    switch (operator)
    {
        case '+': return op1 + op2;
        case '-': return op1 - op2;
        case '*': return op1 * op2;
        case '/': return op1 / op2;
        default : /* Illegal operator */
            print_error( ERR_OPERATOR );
            exit( 1 ); /* Beende Programm mit Fehler Code */
    }
}
```

Abweisende Schleife (while)

```
#include <stdio.h>
int main(void)

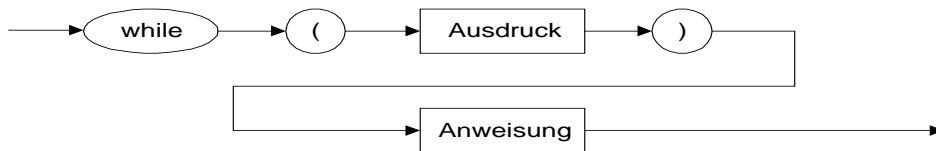
/* Berechne die Summe der Zahlen von 1 bis 100 */

{
    int summand, summe;

    summand = 1;
    summe   = 0;
    while (summand <= 100)
    {
        summe = summe + summand;
        summand++;
    }
    printf("Summe= %d\n", summe)
}
```

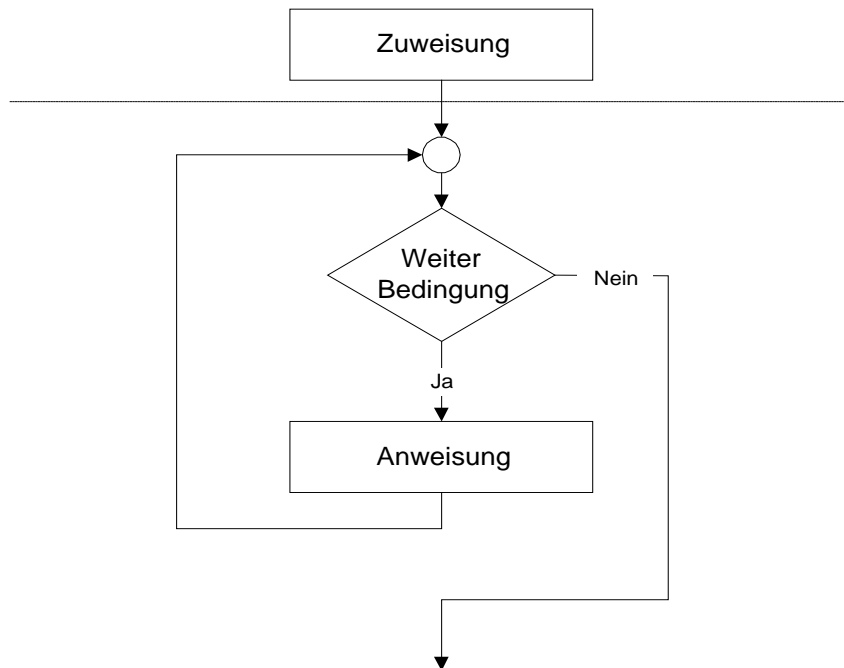
Syntaxdiagramm

while-Schleife:



Abweisende Schleife (while)

Flußdiagramm



Anderes Beispiel

```
#include <stdio.h>
#include <stdlib.h>

/* Liest einen Satz ein und zählt die Leerzeichen darin */

int main( void )
{
    int ch, num_of_spaces = 0;

    printf( "Enter a sentence:\n" );
    ch = getchar(); /* Anfangswert zu Beginn der Schleife */

    while (ch != '\n')
    {
        if (ch == ' ')
            num_of_spaces++;
        ch = getchar();
    }

    printf( "The number of spaces is %d.\n", num_of_spaces );
    exit( 0 );
}
```

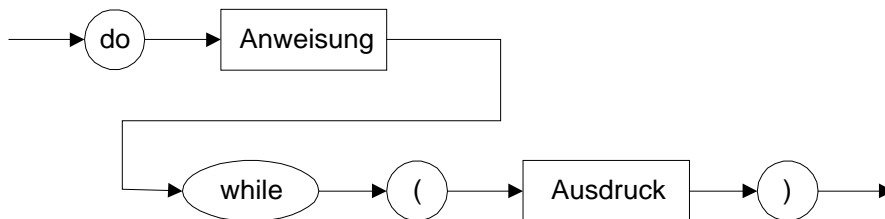
Nichtabweisende Schleife (do...while)

```
#include <stdio.h>
int main(void)
/* Berechne die Summe der Zahlen von 1 bis 100 */
{
    int summand, summe;

    summand = 1;
    summe   = 0;
    do
    {
        summe = summe + summand;
        summand++;
    } while (summand <= 100)
    printf("Summe= %d\n", summe)
}
```

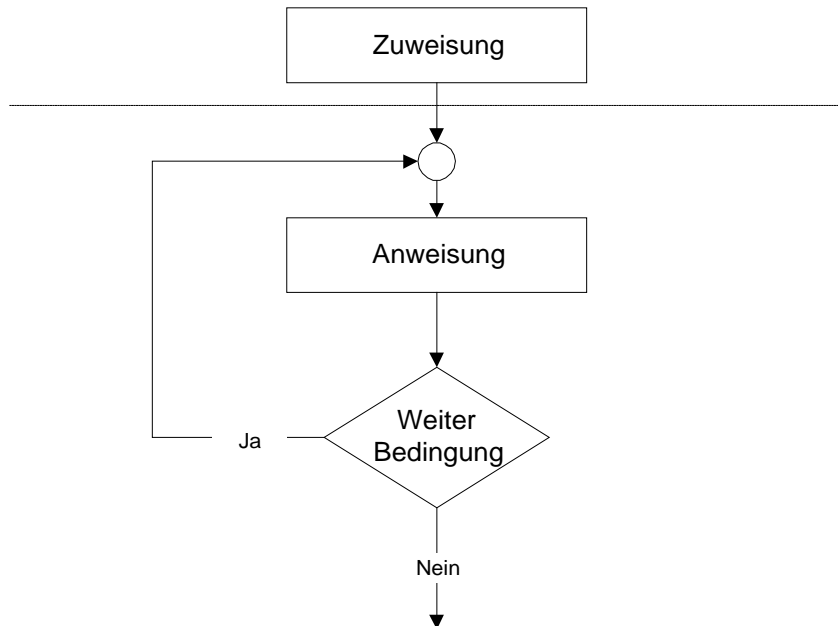
Syntaxdiagramm

do...while-Schleife:



Nichtabweisende Schleife (do...while)

Flußdiagramm



Beispiel von oben

```
#include <stdio.h>
#include <stdlib.h>

/* Liest einen Satz ein und zählt die Leerzeichen darin */

int main( void )
{
    int ch, num_of_spaces = 0;

    printf( "Enter a sentence:\n" );
    do
    {
        ch = getchar();
        if (ch == ' ')
            num_of_spaces++;
    } while (ch != '\n');

    printf( "The number of spaces is %d.\n", num_of_spaces );
    exit( 0 );
}
```


Zählschleife (for)

Optimiert für den Fall, daß Anfangswert, Grenzwert und Inkrement der Schleife von Anfang an bekannt ist.

```
#include <stdio.h>
int main(void)

/* Berechne die Summe der Zahlen von 1 bis 100 */

{
    int summand, summe;

    summe    = 0;
    for (summand = 1; summand <= 100; summand++)
    {
        summe = summe + summand;
    }
    printf("Summe= %d\n", summe)
}
```

Oder sogar noch knapper

```
#include <stdio.h>
int main(void)

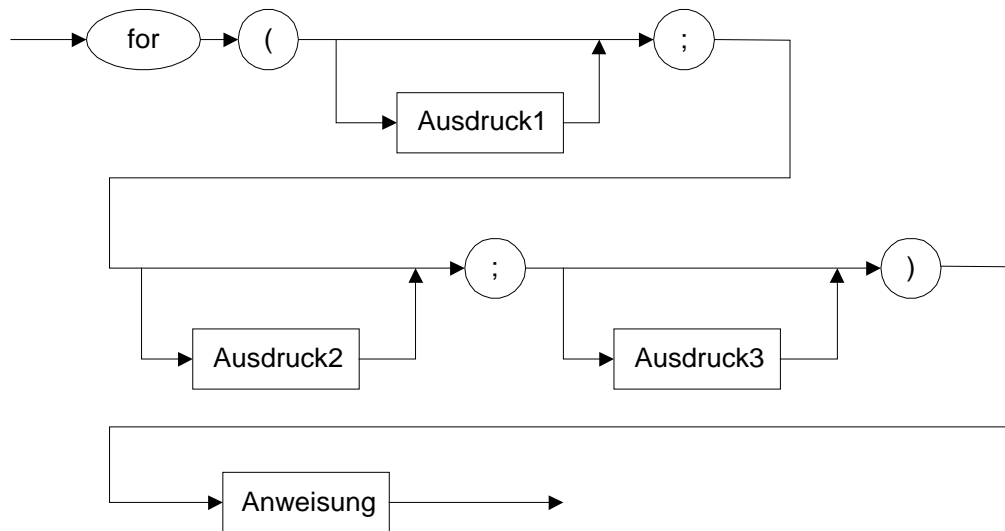
/* Berechne die Summe der Zahlen von 1 bis 100 */

{
    int summand, summe;

    for (summe = 0, summand = 1; summand <= 100; summand++)
    {
        summe = summe + summand;
    }
    printf("Summe= %d\n", summe)
}
```

Zählschleife (for)

Syntaxdiagramm



Ablauf

1. *Ausdruck1* wird ausgewertet. Üblicherweise eine Zuweisung zur Initialisierung einer oder mehrerer Variablen.
2. *Ausdruck2* wird ausgewertet (Bedingung der Schleife)
3. Ist *Ausdruck2* FALSE, so ist die Schleife beendet, ist *Ausdruck2* TRUE, so wird *Anweisung* ausgeführt.
4. Nachdem *Anweisung* ausgeführt wurde, wird *Ausdruck3* ausgewertet (z.B. Erhöhung der Laufvariablen). Anschließend springt die Ausführungskontrolle zu Punkt 2.

Zählschleife (for)

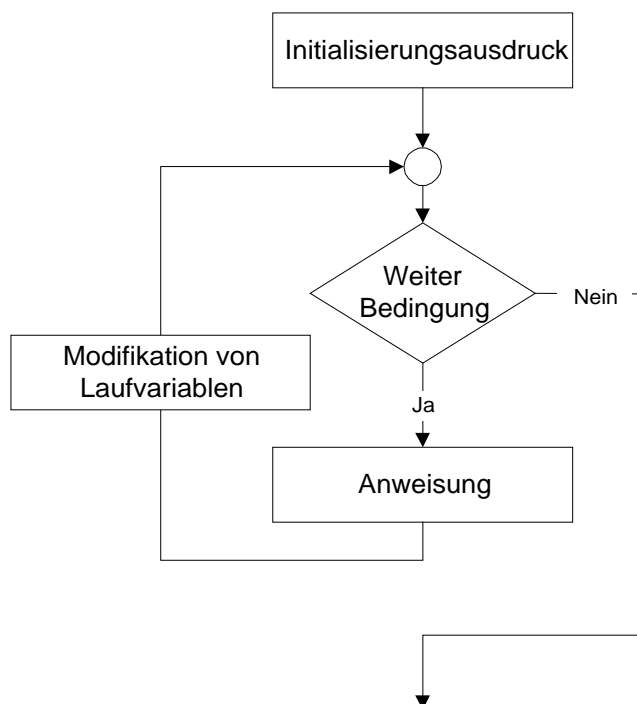
Vergleich mit *while*-Anweisung

```
for (ausdruck1; ausdruck2; ausdruck3)  
    anweisung;
```

entspricht ...

```
ausdruck1;  
while (ausdruck2)  
{  
    anweisung;  
    ausdruck3;  
}
```

Flußdiagramm



Zählschleife (for)

Beispiel 1

```
long int factorial( long val )  
  
/* Berechne Fakultät von val */  
  
{  
    int j, fact = 1;  
  
    for (j=2; j <= val; j++)  
        fact = fact*j;  
    return fact;  
}
```

Zählschleife (for)

Beispiel 2

```
/* This function reads a string of digits from the
 * terminal and produces the string's integer value.
 */
#include <stdio.h>
#include <ctype.h>

int make_int(void )
{
    int num=0, digit;

    digit = getchar();
    for ( ; isdigit(digit); digit = getchar())
    {
        num = num * 10;
        num = num + (digit - '0'); /* Konvertiere Code nach */
                                   /* Integer                */
    }
    return num;
}
```

Nachteil: getchar() wird zweimal aufgerufen.

Kompaktere – wenn auch nicht so leicht verständliche – Version:

```
...
{
    int num=0, digit;

    while (isdigit( digit = getchar() ))
    {
        num = num * 10;
        num = num + (digit - '0');
    }
    return num;
}
```

Wegfall einzelner Bestandteile der for-Anweisung

Ohne Initialisierung

Drucke bestimmte Anzahl von Leerzeilen:

```
#include <stdio.h>

void pr_newline( int newline_num )
{
    for ( ; newline_num > 0; newline_num-- )
        printf( "\n" );
}
```

Ohne Schleifenrumpf

```
/*
 * Read white space from the terminal and discard those
 * characters
 */
#include <stdio.h>
#include <ctype.h> /* Header file for isspace(). */

void skip_spaces(void)
{
    int c;
    for ( c = getchar(); isspace( c ); c = getchar() )
        ; /* Null Statement */
    ungetc( c, stdin ); /* Put the nonspace character back
                        * in the buffer.
                        */
}
```

Noch kompakter als *while*-Schleife

```
...
void skip_spaces (void)
{
    char c;

    while (isspace( c = getchar() ))
        ; /* Null Statement */
    ungetc( c, stdin );
}
```

Verschachtelte Schleifen

Beispiel 1

```
/* print a multiplication table using nested loops */

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int j, k;
    printf("          1      2      3      4      5      6      7      8      9"
           "      10\n");
    printf("-----\n");

    for (j = 1; j <= 10; j++) /* outer loop */
    {
        printf( "%2d|", j );
        for (k=1; k <= 10; k++) /* inner loop */
            printf( "%5d", j*k );
        printf( "\n" );
    }
    exit( 0 );
}
```

Verschachtelte Schleifen

Beispiel 2

Lies eine Integer- oder Gleitkommazahl zeichenweise ein und gib als *double* zurück:

```
/*
 * Lies von der Eingabe eine Integer- oder Gleitkommazahl und
 * weise ihren Wert dem Funktionsergebnis zu
 */
#include <stdio.h>
#include <ctype.h>
#define DECIMAL_POINT '.'

double parse_num()
{
    int c, j, digit_count = 0;
    double value = 0, fractional_digit;

    while (isdigit( c = getchar() ) )
        value = value * 10 + (c - '0');

    /* If c is not digit, see if there's decimal point */
    if (c == DECIMAL_POINT) /* get fraction */
        while (isdigit( c = getchar() ))
        {
            digit_count++;
            fractional_digit = c - '0';
            for (j=0; j < digit_count; j++)
                fractional_digit = fractional_digit/10;
            value = value + fractional_digit;
        }
    ungetc( c, stdin );
    return value;
}
```


break Anweisungen

break beendet eine Schleife und setzt den Programmablauf nach der Schleife fort

```
for (cnt=0; cnt<50; cnt++)
{
    c = getchar();
    if (c=='\n')
        break;
    else
        /* verarbeite das Zeichen */
        ...
}
```

Meistens kann die Schleife anders formuliert werden, um den Einsatz von *break* zu vermeiden:

```
for (cnt=0, c=' '; cnt<50 && c!='\n'; cnt++)
{
    c = getchar();
    if (c!='\n')
        /* verarbeite das Zeichen */
        ...
}
```

continue Anweisungen

continue überspringt die restlichen Anweisungen des Schleifendurchgangs und beginnt mit dem nächste Schleifendurchlauf

```
#include <stdio.h>
#include <ctype.h>

int mod_make_int()
{
    int num = 0, digit;
    while ((digit = getchar()) != '\n')
    {
        /* Überspringe alle Zeichen, die keine Ziffern sind */
        if (isdigit( digit ) == 0)
            continue;
        num = num * 10;
        num = num + (digit - '0');
    }
    return num;
}
```

Auch hier kann die Verwendung von *continue* vermieden werden:

```
#include <stdio.h>
#include <ctype.h>

int mod_make_int()
{
    int num = 0, digit;
    while ((digit = getchar()) != '\n')
    {
        /* Verarbeite nur Zeichen, die Ziffern sind */
        if (isdigit( digit ))
        {
            num = num * 10;
            num = num + (digit - '0');
        }
    }
    return num;
}
```

Fehler bei der Verwendung von Schleifenkonstrukten

1. Abbruchbedingung muß auswertbar sein. Im Schleifenrumpf (oder in der for-Anweisung) muß sich die Bedingung ändern, sonst kann die Schleife nicht abbrechen.
2. Schleifenvariable müssen vor Beginn der Schleife initialisiert sein.
3. Keine Aktionen in die Schleife, die da nicht hingehören. Sonst werden sie mit der Schleife nutzlos n-mal wiederholt. Folge: Zuviel Rechenzeit oder gar Fehler.
4. Richtige „Klammerung„ von zusammengesetzten Anweisungen ({ ... }) beachten.

7. Operatoren und Ausdrücke

- 7.1. Vorrang und Assoziativität
- 7.2. Unäre Operatoren
- 7.3. Binäre arithmetische Operatoren
- 7.4. Arithmetische Zuweisungs-Operatoren
- 7.5. Inkrement- und Dekrement-Operatoren
- 7.6. Komma-Operator
- 7.7. Relationale Operatoren
- 7.8. Logische Operatoren
- 7.9. Bitmanipulations-Operatoren
- 7.10. Bitmanipulations-Operatoren mit gleichzeitiger Zuweisung
- 7.11. Typumwandlungs Operator
- 7.12. sizeof-Operator
- 7.13. Operator für bedingte Ausdrücke
- 7.14. Speicherorientierte Operatoren

Ausdrucksarten

Konstanten-Ausdrücke

```
5  
5 + 6 * 13 / 3.0  
'a'
```

Ganzzahlige-Ausdrücke

(j, k seien Integers)

```
j  
j * k  
j / k + 3  
k - 'a'  
3 + (int) 5.0
```

Gleitkomma-Ausdrücke

(x, y seien float oder double)


```
x  
x + 3  
x / y * 5  
3.0  
3.0 - 2  
3 + (float) 4
```

Zeiger-Ausdrücke

(p sei ein Pointer, j ein Integer)

```
p  
&j  
p + 1  
"abc"  
(char *) 0x000fffff
```

Operatoren

Operatoren-Klasse	Operatoren	Assoziativität	Vorrang
primäre	() [] -> .	links-nach-rechts	am höchsten
unäre	cast Operator sizeof & (Adresse) * (Dereferenzierung) - + ~ ++ -- !	rechts-nach-links	
multiplikativ	* / %	links-nach-rechts	
additiv	+ -	links-nach-rechts	
shift	<< >>	links-nach-rechts	
relational	< <= > >=	links-nach-rechts	
Gleichheit	== !=	links-nach-rechts	
bit-weises AND	&	links-nach-rechts	
bit-weises exklusives OR	^	links-nach-rechts	
bit-weises inklusive OR		links-nach-rechts	
logisches AND	&&	links-nach-rechts	
logisches OR		links-nach-rechts	
konditional	? :	rechts-nach-links	
Zuweisung	= += -= *= /= %= >>= <<= &= ^=	rechts-nach-links	
Komma	,	links-nach-rechts	am niedrigsten

Unäre Operatoren

<i>Operator</i>	<i>Symbol</i>	<i>Form</i>	<i>Operation</i>
unäres Minus	-	-x	Negation von x
unäres Plus	+	+x	Wert von x

Binäre arithmetische Operatoren

<i>Operator</i>	<i>Symbol</i>	<i>Form</i>	<i>Operation</i>
Multiplikation	*	x * y	x mal y
Division	/	x / y	x dividiert durch y
Modulo (Rest)	%	x % y	x modulo y
Addition	+	x + y	x plus y
Subtraktion	-	x - y	x minus y

Vorsicht bei Integer-Division und Modulo-Operation:

5/2 ergibt 2
1/3 ergibt 0

Aber:

-5/2 ergibt -2 oder -3
1/-3 ergibt 0 oder -1

-5 % 2 ergibt 1 oder -1
7 % -4 ergibt 3 oder -3

Ausweg: Funktion *div()* mit definiertem Verhalten.

Arithmetische Zuweisungs-Operatoren

<i>Operator</i>	<i>Symbol</i>	<i>Form</i>	<i>Operation</i>
Zuweisung	=	$a = b$	weise a den Wert von b zu
Addition und Zuweisung	+=	$a += b$	weise a den Wert von $a+b$ zu
Subtraktion und Zuweisung	-=	$a -= b$	weise a den Wert von $a-b$ zu
Multiplikation und Zuweisung	*=	$a *= b$	weise a den Wert von $a*b$ zu
Division und Zuweisung	/=	$a /= b$	weise a den Wert von a/b zu
Modulo und Zuweisung	%=	$a \% = b$	weise a den Wert von $a\%b$ zu

Zuweisungen können mehrfach vorkommen:

```
a = b = c = d = 1;
```

entspricht

```
(a = (b = (c = (d = 1))));
```

Vorsicht

```
j = j * 3 + 4;
j *= 3 + 4;
```

geben verschiedene Resultate!

Bindung der arithmetischen Wertzuweisungsoperatoren ist sehr gering.

Der zweiten Zeile entspricht:

```
j = j * (3 + 4);
```


Inkrement- und Dekrement-Operatoren

<i>Operator</i>	<i>Symbol</i>	<i>Form</i>	<i>Operation</i>
Postfix Inkrement	++	a++	liefere Wert von a, dann inkrementiere a
Postfix Dekrement	--	a--	liefere Wert von a, dann dekrementiere a
Präfix Inkrement	++	++a	inkrementiere a, dann liefere Wert von a
Präfix Dekrement	--	--a	dekrementiere a, dann liefere Wert von a

Komma-Operator

<i>Operator</i>	<i>Symbol</i>	<i>Form</i>	<i>Operation</i>
Koma	,	a, b	werte a aus, werte b aus, Resultat ist b

Mit Hilfe des Komma-Operators können mehrere Ausdrücke an Stellen vorkommen, wo eigentlich nur ein Ausdruck vorgesehen ist.

```
for (j=0, k=100; k-j > 0; j++, k--) ...
```

Vorsicht: Lesbarkeit!

Relationale Operatoren

<i>Operator</i>	<i>Symbol</i>	<i>Form</i>	<i>Operation</i>
größer	>	$a > b$	1 wenn a größer als b ist, sonst 0
kleiner	<	$a < b$	1 wenn a kleiner als b ist, sonst 0
größer oder gleich	>=	$a >= b$	1 wenn a größer oder gleich b ist, sonst 0
kleiner oder gleich	<=	$a <= b$	1 wenn a kleiner oder gleich b ist, sonst 0
gleich	==	$a == b$	1 wenn a gleich b ist, sonst 0
ungleich	!=	$a != b$	1 wenn a ungleich b ist, sonst 0

Logische Operatoren

<i>Operator</i>	<i>Symbol</i>	<i>Form</i>	<i>Operation</i>
logisches UND	&&	$a \&\& b$	1 falls a und b ungleich Null, sonst 0
logisches ODER		$a b$	1 falls a oder b ungleich Null, sonst 0
logische Negation	!	$! a$	1 falls a gleich Null, sonst 0

Wahrheitstafel

(p und q seien vom Typ int)

p	q	$! p$	$! q$	$p \&\& q$	$p q$
0	0	1	1	0	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	1	1

Bitmanipulations-Operatoren

<i>Operator</i>	<i>Symbol</i>	<i>Form</i>	<i>Operation</i>
right shift	>>	$x \gg y$	x um y Bits nach rechts schieben
left shift	<<	$x \ll y$	x um y Bits nach links schieben
bit-weises UND	&	$x \& y$	x mit y bit-weise und-verknüpft
bit-weises ODER		$x y$	x mit y bit-weise oder-verknüpft
bit-weises exklusives ODER (XOR)	^	$x \wedge y$	x mit y bit-weise exklusiv-oder-verknüpft
bit-weises Komplement	~	$\sim x$	bit-weises (Einer-) Komplement von x

Bitmanipulations-Operatoren mit gleichzeitiger Zuweisung

<i>Operator</i>	<i>Symbol</i>	<i>Form</i>	<i>Operation</i>
right shift Zuweisung	>>=	$a \gg= b$	weise $a \gg b$ an a zu
left shift Zuweisung	<<=	$a \ll= b$	weise $a \ll b$ an a zu
bit-weises UND Zuweisung	&=	$a \&= b$	weise $a \& b$ an a zu
bit-weises ODER Zuweisung	=	$a = b$	weise $a b$ an a zu
bit-weises exklusives ODER (XOR) Zuweisung	^=	$a \wedge= b$	weise $a \wedge b$ an a zu

Typumwandlungs Operator

Operator	Symbol	Form	Operation
type cast	(type)	(type) e	Konvertiere e zu type

Beispiel:

```
(float) 3 / 2
```

cast-Operator hat sehr hohen Vorrang!

Obiger Ausdruck entspricht:

```
( (float) 3 ) / 2
```

sizeof-Operator

Operator	Symbol	Form	Operation
sizeof	sizeof	sizeof (t) oder sizeof x	Gibt als Ergebnis die Größe in Bytes des Datentyps t oder des Ausdrucks x

Zwei Typen von Operanden:

- ein Ausdruck:
dieser wird dabei nicht berechnet, d.h. es gibt keine Seiteneffekte;
Klammer ist optional
- oder
- ein Datentyp:
muß in Klammern stehen

Der Ergebnisdatentyp ist laut ANSI immer *unsigned* (*int* oder *long*).

Operator für bedingte Ausdrücke

<i>Operator</i>	<i>Symbol</i>	<i>Form</i>	<i>Operation</i>
bedingter Ausdruck	<code>? :</code>	<code>a ? b : c</code>	falls a ungleich Null (also TRUE), so ist b das Resultat, sonst c

Ein Operator mit drei Operanden (ternärer Operator)!

Anstelle:

```
if (x < y)
    z = x;
else
    z = y;
```

kann man – nicht unbedingt leserlicher – schreiben

```
z = ((x < y) ? x : y);
```

Sollte eher vermieden werden!

Speicherorientierte Operatoren

<i>Operator</i>	<i>Symbol</i>	<i>Form</i>	<i>Operation</i>
Adresse von	<code>&</code>	<code>&x</code>	Adresse von x
Dereferenzierung	<code>*</code>	<code>*a</code>	Wert des Objektes gespeichert an der Adresse in a
Array-Elemente	<code>[]</code>	<code>x[5]</code>	Wert des Array-Elementes 5
Punkt (dot)	<code>.</code>	<code>x.y</code>	Wert der Komponente (member) y der Struktur x
Pfeil nach rechts	<code>-></code>	<code>p->y</code>	Wert der Komponente (member) y der Struktur, auf die p zeigt

8. Felder und Zeiger

- 8.1. Felder, Reihungen (Arrays)
- 8.2. Initialisierung von Feldern
- 8.3. Zeiger-Arithmetik
- 8.4. Zeiger als Funktions-Parameter
- 8.5. Zugriff auf Felder durch Zeiger
- 8.6. Übergabe von Feldern als Funktionsparameter
- 8.7. Zeichenkette (String)
- 8.8. Mehrdimensionale Felder
- 8.9. Arrays of Pointers
- 8.10. Zeiger auf Zeiger

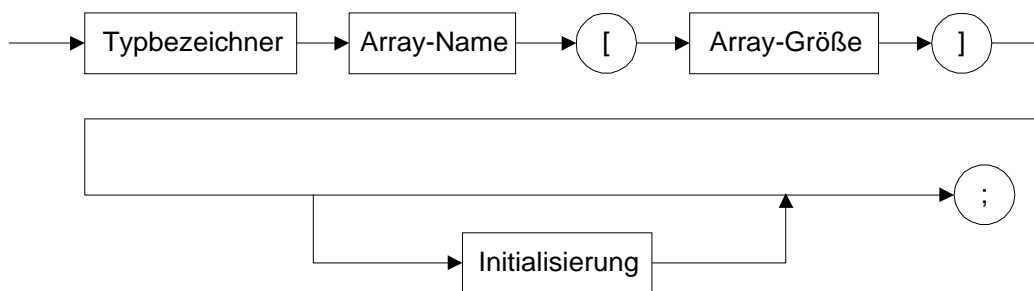
Felder

Beispiel

```
float a[10];  
int i;  
...  
for (i=0; i<10; i++)  
{  
    printf("Gib %d-ten Koeffizienten: ", i);  
    scanf("%f", &a[i])  
}
```

Syntaxdiagramm

Feld-Deklaration:



Felder

Referenz von Array-Elementen

```
x    = a[0] + a[1];  
a[3] = 95.0;
```

Nicht verwechseln mit der Deklaration!

```
/* Dies ist eine Array-Deklaration.  
 * Die 4 gibt die Anzahl von Array-Elementen an  
 */  
int ar[4];  
  
/* Dies ist der Zugriff auf ein Array-Element.  
 * Die 3 spezifiziert das referenzierte Array-Element,  
 * Numerierung beginnt bei 0!  
 */  
ar[3] = 0;
```

Beispiel-Programm

Bestimmung der mittleren Tagestemperatur eines Jahres

```
#include <stdio.h>  
#include <stdlib.h>  
  
#define DAYS_IN_YEAR 365  
  
int main( void )  
{  
    int j, sum=0;  
    int daily_temp[DAYS_IN_YEAR];  
  
    /* Assign values to daily_temp[] here. */  
  
    for (j=0; j < DAYS_IN_YEAR; ++j)  
        sum += daily_temp[j];  
  
    printf( "The average temperature for the year is %d.\n",  
           sum/DAYS_IN_YEAR );  
    exit( 0 );  
}
```


Initialisierung von Feldern

ANSI-Standard:

Sowohl permanente als auch sog. automatische Arrays können initialisiert werden:

```
static int a_ar[5]; /* Elemente werden mit 0 initialisiert */  
int b_ar[5] = {1, 2, 3, 4, 5};
```

Initialisierung durch Angabe der Werte in geschweiften Klammern

Größenangabe kann entfallen (nicht unbedingt empfohlen!):

```
char c_ar[] = {'a', 'b', 'c', 'd'};
```

Zeiger-Arithmetik

C erlaubt arithmetische Ausdrücke mit Zeigern.

Sei p eine Zeigervariable. dann bedeutet

$$p + 3$$

3 Objekte nach dem Objekt, auf das p zeigt.

Es werden in dem obigem Ausdruck nicht 3 Bytes sondern

$3 * (\text{Größe des Objekts, auf das } p \text{ zeigt})$ Bytes
hinzuaddiert.

Subtraktion zweier Zeiger

Müssen auf denselben Objekttyp zeigen

$$p - q$$

Resultat ist Anzahl der Objekte (nicht Bytes!) zwischen den beiden Zeigern.

Zeiger-Arithmetik

Beispiele

`&a[3] - &a[0]`

liefert als Wert 3.

Nehmen wir die folgende Deklaration an:

```
long *p1, *p2;  
int  j;  
char *p3;
```

Dann gilt für die folgenden Zeiger-Ausdrücke:

```
p2 = p1 + 4;      /* legal */  
j = p2 - p1;      /* legal, Ergebnis sei z.B. 4 */  
j = p1 - p2;      /* legal, Ergebnis ist dann -4 */  
p1 = p2 - 2;      /* legal, kompatible Zeigertypen */  
p3 = p1 - 1;      /* falsch, inkompatible Zeigertypen */  
j = p1 - p3;      /* falsch, inkompatible Zeigertypen */
```

Null-Pointer

Zeigt garantiert auf kein gültiges Objekt.

```
char *p;  
p = 0;          /* Mache p zum Null-Pointer */
```

Hier kein Typecasting notwendig!

Anwendung

```
char *p;  
...  
while (p)  
{  
    ...  
    /* Iteriere bis p zum Null-Pointer wird */  
    ...  
}
```

Zeiger als Funktions-Parameter

Zeigertypen der formalen und aktuellen Parameter müssen übereinstimmen.

Beispiel mit unerwartetem Ergebnis

```
#include <stdlib.h>
#include <stdio.h>

void clr( long *p )
{
    *p = 0; /* Store a zero at location p. */
}

int main( void )
{
    static short s[3] = {1, 2, 3};
    clr( &s[1] ); /* Clear element 1 of s[]. */
    printf( "s[0]=%d\ns[1]=%d\ns[2]=%d\n", s[0], s[1], s[2] );
    exit( 0 );
}
```

Ergebnis

```
s[0]=1
s[1]=0
s[2]=0
```

Zugriff auf Felder durch Zeiger

Regeln

- Gegeben ein Zeiger, der auf den Anfang eines Feldes zeigt. Addiert man zu dem Zeiger eine Integerzahl und dereferenziert das Ergebnis, so ist das dasselbe, wie wenn man die Integerzahl als Index für das Feld verwendet.
- Die Ausdrücke

`ar`

und

`&ar[0]`

liefern exakt dasselbe Ergebnis.

D.h. ein *Array-Name* wird in C immer als *Zeiger auf den Anfang des Arrays* interpretiert.

Unterschied: Zeiger können ihren Wert ändern, Array-Namen nicht.
Daher können „nackte“, Array-Namen nicht auf der linken Seite einer Wertzuweisung stehen.

Zugriff auf Felder durch Zeiger

Beispiele

Mit der Deklaration

```
float ar[4], *p;
```

gilt

```
p = ar;    /* Legal; dasselbe wie p = &ar[0] */
ar = p;    /* Falsch: Kann Adresse eines Arrays nicht ändern */
&p = ar;   /* Falsch: Kann Adr. eines Zeigers nicht ändern */
ar++       /* Falsch: Kann Adr. eines Arrays nicht inkrement.*/
ar[1] = *(p+3); /* Legal: Beides Werte von Array-Elementen */
p++;       /* Legal: Zeigervar. kann inkrementiert werden */
```

Übergabe von Feldern als Funktionsparameter

Name eines Feldes als aktueller Parameter wird als Zeiger auf dieses Feld interpretiert und so übergeben.

```
int main(void)
{
    extern float func(float ar[]);
    float x, farray[5];
    ...
    x = func( farray ); /* Adresse wird übergeben */
    ...                /* ohne &-Operator!      */
}
```

Definition der Funktion

```
float func(float ar[])
{
    ...
}
```

oder (weniger empfohlen!)

```
float func(float *ar)
{
    ...
}
```


Übergabe von Feldern als Funktionsparameter

Zusätzliche Übergabe der Feldgröße

```
void foo(float f_array[], int f_array_size)
{
    ...
    if (lauf_index > f_array_size)
    {
        printf("Feldindex zu groß!\n");
        exit(1);
    }
    ...
}
```

Aufruf z.B.

```
foo( f_array, sizeof(f_array)/sizeof(f_array[0]) );
```

Achtung: Vermeide Bereichsüberschreitung

±1 Fehler

```
int main(void)
{
    int ar[10], j;

    for (j = 0; j <= 10; j++)
        ar[j] = 0;
    ...
}
```

Strings

Deklaration und Initialisierung

Ohne Initialisierung:

```
char str[100];
```

Hier ist auch noch kein Null-Character vorhanden!

Mit Initialisierung:

```
char str[] = "Irgendwelcher Text";
```

Das allokierte Array ist um ein Zeichen länger als der Initialisierungstext (Null-Character)

Fallstricke

```
char str[3] = "four"; /* zu wenig Platz */
```

```
char str[4] = "four"; /* In ANSI C erlaubt, kein Null-Char! */
```

Zeiger auf Character

```
char *ptr = "Mein Text";
```

Hier wird zusätzlich zu dem für das Array notwendigen Speicherplatz eine Zeigervariable deklariert.

Anfänger-Probleme mit Strings und Arrays

```
#include <stdlib.h>

int main( void )
{
    char array[10];
    char *ptr1 = "10 spaces";
    char *ptr2;

    array    = "not OK"; /* cannot assign to an address    */
                        /* of an array                      */

    array[5] = 'A';      /* OK */

    ptr1[5]  = 'B';      /* OK */

    ptr1     = "OK";     /* original string ist lost in */
                        /* memory!                      */

    ptr1[5]  = 'C';      /* "legal" but runtime error due */
                        /* to prior assignment of        */
                        /* of 2-character string        */

    *ptr2    = "not OK"; /* type mismatch                */

    ptr2     = "OK";     /* OK although ptr2 was not     */
                        /* initialized                  */

    exit( 0 );
}
```

Strings vs. Chars

Char-Konstanten: 'a'

String-Konstanten "a"

Unterschiedliche Speicherreservierung

```
char c = 'a';      /* 1 Byte */
char *ps = "a";    /* 1 Zeiger plus 2 Bytes */
                  /* (einschl. Null-Char) */
```

Vorsicht bei Wertzuweisungen

Mit obigen Deklarationen:

<code>*ps = 'a';</code>	Char Konstante wird dereferenziertem Pointer auf char zugewiesen
<code>*ps = "a"; /* FALSCH */</code>	Man kann eine String-Konstante nicht an einen dereferenzierten Pointer auf char zugewiesen
<code>ps = "a";</code>	OK: String-Konstante ist Pointer auf char; dessen Wert wird ps zugewiesen
<code>ps = 'a';</code>	Falsch: ps ist ein Pointer, kein char!

Initialisierung und Wertzuweisung nicht symmetrisch!

Gilt für alle Datentypen. Z.B.:

```
float f;
float *pf = &f; /* OK */
...
*pf = &f;      /* FALSCH */
```

Ein-/Ausgabe von Strings

Wie gewohnt:

- Lesen mit *scanf()*
- Ausgeben mit *printf()*

Aber:

- Kein Lesen über *white space* innerhalb des Strings hinweg!
- Führender *white space* wird überlesen.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_CHAR 80

/* Lies Zeichenkette ein und drucke sie 10-mal aus */

int main( void )
{
    char str [MAX_CHAR];
    int i;

    printf( " Enter a string: " );
    scanf( "%s", str );          /* str ist ein Zeiger auf den */
                                /* Anfang des Arrays!          */

    for (i = 0; i < 10; ++i)
        printf( "%s\n", str );
    exit( 0 );
}
```

Problem, wenn eingegebener String länger als der dafür vorgesehene Speicherbereich.

Eingabe von Strings

Funktion *gets()*

Liest String bis zum *newline* (oder EOF) inklusive *white space*.

```
char *gets(char *s);
```

- Resultat ist Zeiger auf Char
- Argument ist Zeiger auf genügend großes Char-Array, in das gelesen wird.
- Bei Fehler: Funktionswert ist NULL-Pointer.

Obiges Beispiel mit *gets()*

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_CHAR 80

/* Lies Zeichenkette ein und drucke sie 10-mal aus */

int main( void )
{
    char str [MAX_CHAR];
    int i;

    printf( " Enter a string: " );
    if (gets(str))
        for (i = 0; i < 10; ++i)
            printf( "%s\n", str );
    exit( 0 );
}
```

String-Funktionen

Stringlänge

Aktuelle Länge eines Strings (ohne den Null-Character)

Unter Verwendung eines Arrays

```
int strlen( char str[] )
{
    int i=0;
    while (str[i])
        ++i;
    return i;
}
```

Mit einer for-Schleife

```
int strlen( char str[] )
{
    int i;
    for (i = 0; str[i]; ++i)
        ; /* null statement in for body */
    return i;
}
```

String-Funktionen

Stringlänge

Pointer-Version

```
int strlen( char *str )
{
    int i;
    for (i = 0; *str++; i++)
        ; /* null statement */
    return i;
}
```

Typische C-Konstruktion:

```
*str++
```

- Der Operator ++ hat denselben Vorrang wie der Operator *
- Damit regelt die Assoziativität die Ausführungsreihenfolge
- Die Assoziativität ist rechts-nach-links

Es geschieht also das folgende:

- Der Operator ++ wird als *post*-inkrement Operator ausgeführt. Daher wird zunächst *str* zum nächsten Operator übergeben, aber der Compiler merkt sich, daß nach Abschluß der Ausführung des gesamten Ausdrucks *str* noch inkrementiert werden muß.
- Mit dem Operator * wird *str* dereferenziert. Der Wert, auf den *str* zeigt, ist das Resultat.
- Inkrementiere den Zeiger *str*.

String-Funktionen

Copy Funktion

Array-Version

```
/* Copy s2 to s1 */
void strcpy( char s1[], char s2[])
{
    int i;

    for (i=0; s2[i]; ++i)
        s1[i] = s2[i];
    s1[++i] = 0; /* add null character; use prefix incrementing
*/
}
```

Dasselbe mit Pointern

```
void strcpy( char *s1, char *s2)
{
    int i;

    for (i=0; *(s2+i); ++i)
        *(s1+i) = *(s2+i);
    s1[++i] = 0;
}
```

- Array- und Pointer-Schreibweisen entsprechen sich in C völlig
- Erzeugter Code der beiden Versionen ist identisch

s1[i] entspricht voll der Schreibweise *(s1+i)

Man beachte die Klammerung!

Wir erachten jedoch die Array-Schreibweise als lesbarer!

String-Funktionen

Copy Funktion

Besonders effiziente Version

In der letzten Versionen nutzen wir so ziemlich alle Features, die uns C bietet:

```
void strcpy( char *s1, char *s2)
{
    while (*s1++ = *s2++)
        ; /* null statement */
}
```

Statt einen Offset zum Zeiger zu benutzen, inkrementieren wir hier die Zeiger mit post-inkrement Operatoren selbst.

Das Resultat der Zuweisung wird als Testbedingung benutzt. Damit erreichen wir sogar, daß auch der Null-Character in der Schleife kopiert wird.

Diese Version erzeugt wohl den effizientesten Code, ist aber – für einen Anfänger – am wenigsten leicht lesbar.

String-Funktionen

Suchen nach Zeichenkettenmustern

In der ANSI C Laufzeitbibliothek heißt die folgende Funktion *strstr()*.

```
#include <stdio.h>
#include <string.h>

/* Return the position of str2 in str1; -1 if not
 * found.
 */

int pat_match( char str1[], char str2[])
{
    int j, k;

    /* Compare str1[] beginning at index j with each
     * character in str2[].
     * If equal, get next char in str1[].
     * Exit loop if we get to end of str1[],
     * or if chars are equal.
     */

    for (j=0; j < strlen(str1); ++j)
    {
        for (k=0; k < strlen(str2)
            && (str2[k] == str1[k+j]); k++) ;
        /* Check to see if loop ended because we arrived at
         * end of str2. If so, strings must be equal.
         */
        if (k == strlen( str2 ))
            return j;
    }
    return -1;
}
```

-1 als Funktionswert bei Mißerfolg hier angebracht, da im Erfolgsfall der Index größer oder gleich Null ist.

Nachteil:

Selbst in der innersten Schleife wird bei jedem Durchlauf die *strlen()* Funktion aufgerufen.

String-Funktionen

Suchen nach Zeichenkettenmustern

Verbesserte Version bzgl. Laufzeitverhalten

```
#include <stdio.h>
#include <string.h>
/*
 * Return the position of str2 in str1; -1 if not
 * found.
 */
pat_match( char str1[], char str2[])
{
    int j, k;
    int length1 = strlen( str1 );
    int length2 = strlen( str2 );

    for (j=0; j < length1; ++j)
    {
        for (k=0; k < length2
            && (str2[k] == str1[k+j]); k++) ;

        if (k == length2)
            return j;
    }
    return -1;
}
```

String-Funktionen

Suchen nach Zeichenkettenmustern

Beispiel-Hauptprogramm

```
#include <stdio.h>
#include <stdlib.h>

extern int pat_match( char s1[], char s2[]);

int main( void )
{
    char first_string[100] , pattern[100];
    int pos;

    printf( "Enter main string:" );
    gets( first_string ); /* Funktionswert nicht genutzt */
    printf( "Enter substring to find: " );
    gets( pattern );
    pos = pat_match( first_string, pattern );
    if (pos == -1)
        printf( "The substring was not matched.\n" );
    else
        printf( "Substring found at position %d\n"
            , pos );
    exit( 0 );
}
```

String Funktionen in der Laufzeitbibliothek

<i>Funktion</i>	<i>Beschreibung</i>
strcpy()	Kopiert String in ein Array
strncpy()	Kopiert Teil eines Strings in ein Array
strcat()	Konkateniert zwei Strings
strncat()	Konkateniert Teil eines Strings mit einem anderen
strcmp()	Vergleicht zwei Strings
strncmp()	Vergleicht zwei Strings bis zu einer bestimmten Anzahl von Zeichen
strchr()	Finde das erste Auftreten eines Characters in einem String
strcoll()	Vergleicht zwei Strings aufgrund einer spezifizierbaren Sortierreihenfolge
strcspn()	Ermittelt die Länge eines Strings, bis zu der Zeichen eines anderen Strings nicht vorkommen
strerror()	Erzeugt aufgrund einer Fehlernummer einen Fehlertext
strlen()	Ermittelt die Länge eines Strings
strpbrk()	Ermittle die erste Stelle in einem String, in der ein Zeichen aus einem anderen String vorkommt
strrchr()	Ermittle die letzte Stelle in einem String, in der ein Zeichen aus einem anderen String vorkommt
strspn()	Ermittle die Stelle in einem String, bis zu der nur Zeichen aus einem zweiten String vorkommen
strstr()	Finde die erste Stelle in einem String an der ein zweiter String vorkommt
strtok()	Zerteile eine String aufgrund von Trennzeichen
strxfrm()	Transformiere einen String gemäß einer definierbaren Sortierfolge, so daß er dann in strcmp() verwendet werden kann

Mehrdimensionale Felder

Magisches Quadrat:

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

Um dies zu speichern deklariert man mit Initialisierung:

```
int magisch[5][5] =  
    { { 17, 24, 1, 8, 15 },  
      { 23, 5, 7, 14, 16 },  
      { 4, 6, 13, 20, 22 },  
      { 10, 12, 19, 21, 3 },  
      { 11, 18, 25, 2, 9 }  
    };
```

Der innerste (rechtsstehende) Index läuft am schnellsten und beschreibt somit die Reihen (row-major order).

Übergabe mehrdimensionaler Felder an Funktionen

- Als aktuellen Parameter übergibt man den Feldnamen wie üblich.
- Tatsächlich wird ein Zeiger auf den Array-Anfang übergeben.
- Auf der empfangenden Seite muß der formale Parameter geeignet deklariert werden, damit die Funktion den Offset vom Array-Anfang korrekt berechnen kann.

```
int main(void)
{
    extern void f2( int received_ar[] [6] [7]);
    int ar[5] [6] [7];
    ...
    f2( ar );
    ...
}
```


Arrays of Pointers

Arrays von Zeigern zum Speichern von Strings:

```
#include <stdio.h>
#include <stdlib.h>

void drucke_monat( int m )
{
    char *monat[13] = { "Fehler", "Januar",
        "Februar", "März", "April", "Mai", "Juni",
        "Juli", "August", "September", "Oktober",
        "November", "Dezember"
    };

    if (m > 12 || m < 1)
    {
        printf( "Illegaler Monat Wert. \n" );
        exit( 1 );
    }
    printf( "%s\n", monat [m] );
}
```

- *monat* ist ein Array aus 13 Elementen, wobei jedes Element ein Zeiger auf den Anfang eines Strings ist.
- Um Rechenaufwand zu sparen und die Funktion etwas intuitiver zu gestalten, wurde das 0-te Element verschenkt.

Zeiger auf String als Resultat

```
#include <stdio.h>
#include <stdlib.h>

char *monats_name( int m )
{
    static char *monat[13] = { "Fehler", "Januar",
        "Februar", "März", "April", "Mai", "Juni",
        "Juli", "August", "September", "Oktober",
        "November", "Dezember"
    };

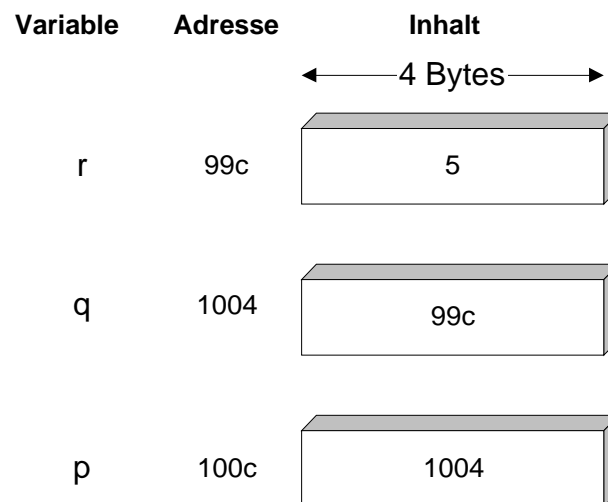
    if (m > 12 || m < 1)
    {
        printf( "Illegaler Monat Wert. \n" );
        exit( 1 );
    }
    return monat [m];
}
```

Zeiger auf Zeiger

Betrachten wir die folgende Deklaration:

```
int r    = 5;  
int *q   = &r;  
int **p  = &q;
```

Im Speicher stellt sich die Situation wie folgt dar:



Rechtschreibprüfung

Funktion zur Rechtschreibprüfung unter Verwendung eines mehrdimensionalen Arrays

Feld mit allen bekannten Worten einer Sprache

Eine weitere Dimension für mehrere Sprachen

spellf.h

```
typedef enum { DEUTSCH, ENGLISCH, LANG_NUM } LANGUAGE_T;
```

Hauptprogramm

```
#include <stdlib.h>
#include <stdio.h>
#include "spellf.h"
int main(void)
{
    extern char *check_spell( char *str, LANGUAGE_T language );
    char wort[80];
    char *help;

    printf("Gib Wort: ");
    scanf("%s", wort);
    if ((help = check_spell(wort, DEUTSCH)) == NULL)
        printf("O.K.\n");
    else
        printf("Kein Match; dicht bei %s\n", help);
    exit(0);
}
```

Rechtschreibprüfung

Funktion *check_spell* mit 2-dim. Array

```
#include <string.h>
#include "spellf.h"

#define MAX_WORDS 50
/* Dictionary in alphabetic order
 * with NULL as last entry.
 */
static char *dict [LANG_NUM] [MAX_WORDS] = {
    { "aal", "aalen", "aas", "ab",
      "abänderlich", "abarbeiten", "abart", "abbau", NULL
    },
    { "aardvark", "aback", "abacus", "abandon",
      "abash", "abbey", "abbot", "abbreviate",
      NULL
    }
};

/* Return NULL pointer if str is found in
 * dictionary. Otherwise, return a pointer to
 * the closest match
 */
char *check_spell( char *str, LANGUAGE_T language)
{
    int j, diff;
    /* Iterate over the words in the dictionary */
    for (j=0; dict[language][j] != NULL; ++j)
    {
        diff = strcmp( str, dict[language][j] );
        /* Finished if str is not greater than dict entry */
        if (diff <= 0)
            if (diff == 0)
                return NULL; /* Match! */
            else
                return dict[language][j]; /* No match, return closest
*/
/* spelling */
    }
    /* Return last word if str comes after last
 * dictionary entry
 */
    return dict[language][j - 1];
}
```

Rechtschreibprüfung

Funktion *check_spell* mit Pointer to Pointer

```
#include <string.h>
#include "spellf.h"

#define MAX_WORDS 50
/* Dictionary in alphabetic order
 * with NULL as last entry.
 */
static char *dict [LANG_NUM] [MAX_WORDS] = {
    { "aal", "aalen", "aas", "ab",
      "abänderlich", "abarbeiten", "abart", "abbau", NULL
    },
    { "aardvark", "aback", "abacus", "abandon",
      "abash", "abbey", "abbot", "abbreviate",
      NULL
    }
};

/* Return NULL pointer if str is found in
 * dictionary. Otherwise, return a pointer to
 * the closest match.
 * This time use pointers instead time consuming array
references
 */
char *check_spell( char *str, LANGUAGE_T language)
{
    int diff;
    char **cur_word;
    /* Iterate over the words in the dictionary */
    for (cur_word = dict[language]; *cur_word; cur_word++)
    {
        diff = strcmp( str, *cur_word );
        /* Finished if str is not greater than dict entry */
        if (diff <= 0)
            if (diff == 0)
                return NULL; /* Match! */
            else
                return *cur_word; /* No match, return closest */
                                   /* spelling */
    }
    /* Return last word if str comes after last
    * dictionary entry
    */
    return cur_word[-1];
}
```

9. Speicherklassen

- 9.1. Automatische vs. statische Variablen
- 9.2. Gültigkeitsbereich (scope)
- 9.3. Globale Variablen
- 9.4. Die register-Spezifikation
- 9.5. Die const-Modifikation bei der Speicherklassendefinition
- 9.6. Die volatile-Modifikation bei der Speicherklassen-
definition
- 9.7. Zusammenfassung von Speicherklassen
- 9.8. Dynamische Speicherallokierung

Speicherklassen (1/2)

Gültigkeit von Variablen bzgl. Ort und Zeit

Gültigkeitsbereich (engl. *scope*):

Die Gültigkeit bezüglich Ort im Sourcecode,

- in welcher Datei oder Funktion der Name aktiv ist oder
- ob er gar in dem gesamten Programm denselben Speicherbereich bezeichnet

Lebensdauer (engl. *duration*)

Die Gültigkeit einer Variablen bezüglich Zeit.

Dies kann

- die ganze Zeit, in der das Programm läuft, sein (*statische* Variablen, *fixed duration*) oder auch
- nur die Zeit, während der die Kontrolle in einer Funktion ist. Wird die Funktion beendet, existiert die Variable und ihr Inhalt nicht mehr (*automatisch, automatic duration*).

Beide Eigenschaften faßt man als *Speicherklasse* (*storage class*) zusammen

Speicherklassen (2/2)

Beispiel

```
void func()  
{  
    int j;  
    static int ar[] = {1, 2, 3, 4};  
    ...  
}
```

Gültigkeitsbereich

j und *ar* sind beide nur in der Funktion gültig (sichtbar), in der sie definiert sind (*block scope*).

Man nennt solche Variablen auch *lokale* Variablen.

Lebensdauer

- *j* ist eine automatische Variable (die Voreinstellung für lokale Variablen).
Ihr Speicherplatz wird erst zu Beginn der Abarbeitung der Funktion allokiert, beim Verlassen der Funktion wird er wieder freigegeben.
Damit ist auch der Wert der Variablen verloren! Im Prinzip kann *j* bei jedem Aufruf der Funktion eine andere Adresse erhalten.
- *ar* ist eine statische, lokale Variable mit fester Adresse während des Programmlaufs.
Sie ist zwar auch nur innerhalb der Funktion sichtbar, aber ihre Werte bleiben auch beim Verlassen erhalten.
Wird die Funktion ein zweites Mal aufgerufen, sind die Werte wieder verfügbar.

Automatische vs. statische Variablen (1/3)

- Automatische Variablen erhalten ihre Speicherzuweisung, sobald ihr Gültigkeitsbereich „betreten,, wird. Wird er wieder verlassen, wird auch ihr Speicher wieder freigegeben.
- Statische Variablen erhalten ihren Speicher beim Start des Programms und verlieren ihn erst wieder, wenn das Programm beendet wird.

Initialisierung

- Da der Speicherplatz von automatischen Variablen erst jedesmal bei Eintreten in den Gültigkeitsbereich reserviert wird, muß dabei auch jedesmal die Initialisierung durchgeführt werden, soweit sie spezifiziert ist.
Das kostet Zeit.
- Statische Variablen werden nur einmal zu Beginn des Programms initialisiert.

Automatische vs. statische Variablen (2/3)

Beispiel

```
#include <stdio.h>

void increment(void)
{
    int j=1;
    static int k=1;

    j++;
    k++;
    printf( "j: %d\tk: %d\n", j, k );
}

int main( void )
{
    increment();
    increment();
    increment();
}
```

Ergebnis

```
j: 2    k: 2
j: 2    k: 3
j: 2    k: 4
```

Weiterer Unterschied

- Als Voreinstellung werden automatische Variablen nicht initialisiert,
- Statische Variablen erhalten als Voreinstellung eine Initialisierung mit 0.

Automatische vs. statische Variablen (3/3)

Beispiel leicht modifiziert

```
#include <stdio.h>

void increment(void)
{
    int j;
    static int k;

    j++;
    k++;
    printf( "j: %d\tk: %d\n", j, k );
}

int main( void )
{
    increment();
    increment();
    increment();
}
```

Ergebnis

(Wert von j ist zufällig)

```
j: 12620    k: 1
j: 12620    k: 2
j: 12620    k: 3
```

- Die Initialisierung von automatischen Variablen kann aus einem Ausdruck bestehen, der auch – bis dahin bereits bekannte – Variablen enthält.
- Die Initialisierung statischer Variablen muß durch einen konstanten Ausdruck geschehen.

Gültigkeitsbereich (scope) (1/4)

In C: 4 Gültigkeitsbereiche

- **Programm**

Sog. *globale* Variablen, die für das gesamte Programm (das aus mehreren Quelldateien bestehen kann) während seines Ablaufs gültig sind. Von überall benutzbar.

- **Datei**

Variable ist gültig vom Punkt ihrer Deklaration bis zum Ende der Quelldatei.

- **Funktion**

Variable ist gültig vom Punkt ihrer Deklaration in der Funktion bis zum Ende der Funktion, also normalerweise in der gesamten Funktion.

- **Block**

Variable ist gültig vom Punkt ihrer Deklaration bis zum Ende des Blockes, in dem sie deklariert ist.

Blöcke sind jede Sequenz von Anweisungen, die durch geschweifte Klammern eingeschlossen sind.

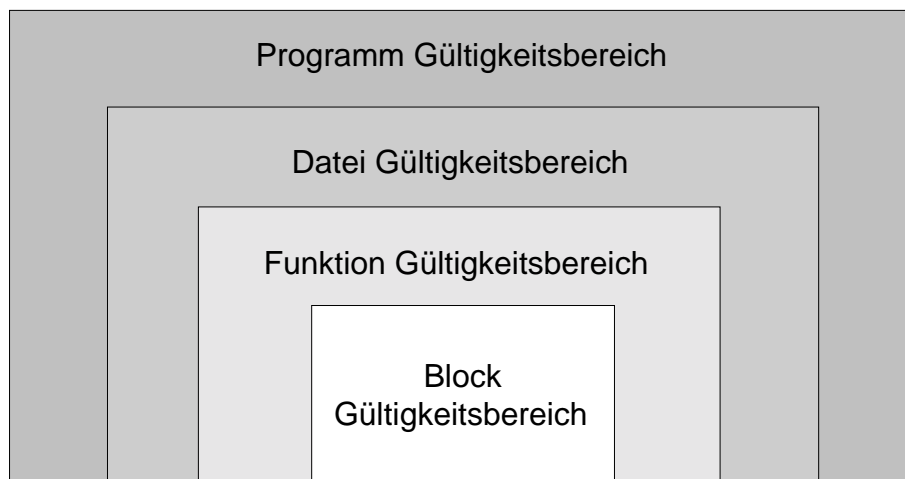
Also sowohl zusammengesetzte Anweisungen als auch Funktionsrümpfe.

Gültigkeitsbereich (scope) (2/4)

Abhängig vom Ort der Deklaration

- In einem Block deklarierte Variablen haben den Block als Gültigkeitsbereich
- Variablen, die außerhalb eines Blockes – somit auch eines Funktionsrumpfes – deklariert sind haben
 - Datei als Gültigkeitsbereich, wenn das Schlüsselwort *static* vorhanden ist,
 - das ganze Programm als Gültigkeitsbereich, wenn das Schlüsselwort *static* nicht vorhanden ist.
- *goto*-Marken haben immer die Funktion als Gültigkeitsbereich.

Die vier Gültigkeitsbereiche bilden eine Hierarchie:



Gültigkeitsbereich (scope) (3/4)

Beispiel für verschiedene Gültigkeitsbereiche

```
int      i;      /* Program scope */
static int j;    /* File scope */
int func(int k)
{
    int m;       /* Function/Block scope */

    for (m=1; m<100; m++)
    {
        int n;   /* Block scope */
        ...
    }
    ...
}
```

Lokale Variablen verschiedener Funktionen können denselben Namen haben; haben nichts miteinander zu tun:

```
int func1(float x)
{
    int j;
    ...
}

int func2(float x)
{
    int j;
    ...
}
```

Die beiden Definitionen von j bezeichnen unterschiedliche Datenobjekte und bilden keinen Namenskonflikt.

Gültigkeitsbereich (scope) (4/4)

- *Formale Parameter* verhalten sich wie *lokale Variablen* ihrer Funktion. Ihr Scope ist derselbe wie der äußerste Block der Funktion.
- Es kann vorkommen, daß die Gültigkeitsbereiche zweier gleichnamigen Variablen überlappen.
Dann verbirgt die Variable mit dem mehr inneren Gültigkeitsbereich (näherer Punkt der Definition) die andere Variable:

```
#include <stdio.h>

int j=10;           /* Program scope */

int main( void )
{
    int j;          /* Block scope hides j at program scope */
    for (j=0; j < 5; ++j)
        printf( "j: %d", j );
}
```

Block Gültigkeitsbereich

- Auf Variablen mit *Block Scope* kann von außerhalb des Blockes nicht zugegriffen werden.
- Dies kann zum Vorteil genutzt werden:
Schutz der Variablen vor irrtümlicher Modifikation (Datenkapselung).
- Durch die Beschränkung der Gültigkeit einer Variablen wird die Komplexität einer Anwendung vermindert.
- Es können Teile eines Programms geschrieben werden, ohne sich um die Namensgebung in anderen Teilen des Programms kümmern zu müssen.
- Programm wird dadurch auch leichter lesbar.

Datei-Gültigkeitsbereich

- Variablen mit Datei-Gültigkeitsbereich (*file scope*) können von allen Funktionen innerhalb der Datei (genauer: nach der Deklaration der Variablen) benutzt werden.
- Jede dieser Funktionen kann sie verwenden und auch verändern.
- Man erhält eine Variable mit Datei-Gültigkeitsbereich indem man sie außerhalb einer Funktion mit dem Schlüsselwort *static* definiert.

Module

- Variablen mit Datei-Gültigkeitsbereich können nützlich sein, wenn eine Reihe von Funktionen, die in dieser Datei definiert sind, mit diesen Daten arbeiten müssen, die Daten nach außen – außerhalb dieser Datei – jedoch verborgen bleiben sollen.
- Eine solche Datei definiert ein sog. *Modul*.

Programm-Gültigkeitsbereich

- Variablen mit Programm-Gültigkeitsbereich (*program scope*) sind *globale Variablen* und können von überall aus dem Programm zugegriffen werden.
- Sie sind also auch für Funktionen, die in anderen Quelldateien definiert sind, sichtbar.
- Mögliche Namenskonflikte sind zu beachten.
- Man erhält eine Variable mit Programm-Gültigkeitsbereich indem man sie außerhalb einer Funktion *ohne* das Schlüsselwort *static* definiert.

Globale Variablen

- Im allgemeinen sollte man es vermeiden, globale Variablen zu verwenden.
- Liest man fremden Quelltext, so limitiert das Schlüsselwort *static* wenigstens den Blick auf die aktuelle Quelldatei.
Fehlt es und hat man *program scope*, so ist Vorsicht angesagt.
- Solche Variablen sollte man sehr überlegt einsetzen, auf keinen Fall, um irgendwelche Funktionsparameter einzusparen.
- In großen Anwendungen können Variablen mit Programm-Gültigkeitsbereich das „Rückgrat“, der Anwendung bilden.
- Da die Namen globaler Variablen nicht nur vom Compiler sondern auch vom Linker verarbeitet werden müssen, kann es Beschränkungen in der erlaubten Namenslänge geben.
- Nach ANSI werden *nur die ersten 6 Zeichen* eines globalen Namens zur Unterscheidung herangezogen.

Definition und Deklaration (1/2)

Engl. *definition* und *allusion* (wörtl. Hinweis, Bezugnahme).

- Bisherige Annahme: Jede Deklaration einer Variablen stellt auch den damit verbundenen Speicherplatz zur Verfügung.
- Steng genommen wird jedoch der Speicherplatz nur bei der *Definition* von Variablen allokiert.
- Globale Variablen erlauben eine zweite Form der Vereinbarung, die *Deklaration* (engl. *allusion*).
- Eine *Deklaration* in diesem engeren Sinne informiert den Compiler nur darüber, daß eine Variable dieses Namens und Typs existiert, er damit arbeiten kann, der Speicherplatz jedoch woanders allokiert wird.

Deklaration (allusion) einer externen Funktion

```
int main(void)
{
    extern int f(int i);          /* Deklaration (allusion) von f */
    extern float g(float x);     /* Deklaration (allusion) von g */
    ...
}
```

Deklaration (allusion) globaler Variablen, die woanders definiert sind

```
void func(...)
{
    extern int glob_j;           /* eine Deklaration (allusion) */
    extern float array_of_f[];  /* eine Deklaration (allusion) */
    ...
}
```

Schlüsselwort *extern* sagt dem Compiler, daß diese Variable oder Funktion woanders definiert ist.

Definition und Deklaration (2/2)

- Durch die *Deklaration* wird es dem Compiler ermöglicht, Typprüfungen durchzuführen und den korrekten Code zu erzeugen.

Syntax zur korrekten Unterscheidung zwischen Definition und Deklaration von globalen Variablen variiert stark zwischen verschiedenen C-Versionen.

Beste Portabilität garantiert folgende Regel:

- Zur *Definition* einer globalen Variablen füge eine *Initialisierung* hinzu und verwende *nicht* das Schlüsselwort *extern*.
- Zur *Deklaration* (*allusion*) einer globalen Variablen füge das Schlüsselwort *extern* hinzu und *unterlasse* eine *Initialisierung* (wäre auch nicht sehr sinnvoll!).

register-Spezifikation

- Hilft dem Compiler bei der Zuordnung von Variablen zu den Registern der CPU.
- Compiler ist jedoch nicht verpflichtet, sich an diesen Hinweis zu halten.
- Da solche Variablen möglicherweise nie im Arbeitsspeicher landen, können sie auch keine Adresse haben, d.h. der Adreßoperator ist nicht erlaubt.

Beispiel

```
int strlen( register char *p)
{
    register int len = 0;
    while (*p++)
        len++;
    return len;
}
```

In der Praxis hat jede CPU nur eine limitierte Anzahl von Registern.

const-Modifikation bei der Speicherklassendefinition (1/2)

- Definition der Variablen als Konstante
- Nach ihrer Initialisierung innerhalb der Vereinbarung nicht mehr veränderbar.

Beispiel

Nach der Definition

```
const char str[10] = "Konstant";
```

kann z.B. die folgende Anweisung

```
str[0] = 'C';
```

nicht mehr ausgeführt werden.

const ist eine Alternative zu *#define*

```
const long double pi = 3.1415926535897932385;
```

Feine Details in Verbindung mit Zeigern

```
int *const const_ptr;    /* Ein Zeiger, dessen Wert    */  
                        /* nicht verändert werden    */  
                        /* kann                          */  
int const *prt_to_const; /* Ein veränderbarer Zeiger, */  
                        /* der auf Integer-Konstanten */  
                        /* zeigt                          */
```

const-Modifikation bei der Speicherklassendefinition (2/2)

Hauptanwendungen:

- Definition reiner Konstanten
- Übergabe von Nicht-Werteparameter als read-only Parameter an Funktionen

```
char *strcpy (char *p, const char *q)
/* Kopiere Inhalt von q nach Objekt von p */
{
    ...
}
```

strcpy() kann den Inhalt von q nicht verändern. q ist ein reiner Eingabeparameter.

volatile-Modifikation bei der Speicherklassendefinition

Diese Modifikation informiert den Compiler darüber, daß solche Variablen ggf. durch äußere Einflüsse modifiziert werden können, ohne daß der Compiler davon weiß

Zusammenfassung der Speicherklassen (1/2)

Es gibt vier Speicherklassen-Schlüsselworte:

- **auto**
Nur gültig für Variablen mit *block scope*. Dort ist es die Voreinstellung. Also eher überflüssig.
- **static**
 - Innerhalb Funktionen: Variable ist statisch (*fixed duration*) auch bei Verlassen der Funktion.
 - Außerhalb von Funktionen: Variable hat *file scope* (anstatt *program scope* ohne *static*)
- **extern**
 - Innerhalb Funktionen: Dies ist eine (externe) Variablen-Deklaration (*global allusion*).
 - Außerhalb von Funktionen: Gleichwertig einer globalen Definition (*program scope*). Dasselbe wie wenn *extern* fehlen würde.
- **register**
Nur für Variablen, die in Funktionen vereinbart sind. Macht die Variable automatisch und gibt Compiler Hinweis, sie in Registern zu halten.
- **Kein Schlüsselwort**
 - Variablen mit *block scope*: Dasselbe wie *auto*.
 - Variablen definiert ausserhalb von Funktionen: Globale Definition.

Zusammenfassung der Speicherklassen (2/2)

Modifikationen einer Speicherklasse:

- **const**
Der Wert der Variablen kann nicht geändert werden.
- **volatile**
Bestimmte Optimierungen des Compilers dürfen nicht durchgeführt werden.

Tabellarisch

GB: Gültigkeitsbereich (scope), LD: Lebensdauer (duration)

<i>Speicherklassen Spezifikation</i>	<i>Außerhalb einer Funktion</i>	<i>Innerhalb einer Funktion</i>	<i>Bei Funktions- parametern</i>
auto oder register	nicht erlaubt	GB: Block LD: automat.	GB: Block LD: automat.
static	GB: Datei LD: statisch	GB: Block LD: statisch	nicht erlaubt
extern	GB: Programm LD: statisch	GB: Block LD: statisch	nicht erlaubt
ohne	GB: Programm LD: statisch	GB: Block LD: automat.	GB: Block LD: automat.

Dynamische Speicherallokierung

Hauptprogramm zum Aufruf der Bubblesort-Funktion

Statisches Array

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_ARRAY 100

int main( void )
{
    extern void bubble_sort(int list[], int list_size);
    int list [MAX_ARRAY], j, sort_num;
    printf ( "How many values are you going to enter? ");
    scanf( "%d", &sort_num );
    if (sort_num > MAX_ARRAY)
    {
        printf ( "Too many values, %d is the maximum\n"
                , MAX_ARRAY);
        sort_num = MAX_ARRAY;
    }

    for (j=0; j < sort_num; j++)
        scanf( "%d", &list[j] );
    bubble_sort( list, sort_num );
    exit( 0 );
}
```

Funktionen zum Verwalten von dynamischem Speicherplatz

- malloc()* Allokiert die spezifizierte Anzahl an Bytes im Arbeitsspeicher. Liefert als Funktionsergebnis einen Zeiger auf den Anfang des Speicherblocks.
- calloc()* Ähnlich wie *malloc()*, initialisiert jedoch den Speicherbereich mit Nullen. Erlaubt es auch, mehrere gleichlange, zusammenhängende Speicherblöcke auf einmal zu allokiieren.
- realloc()* Verändert die Größe eines zuvor allokierten Speicherbereichs.
- free()* Gibt einen vorher allokierten Speicherbereich wieder frei.

Dynamische Speicherallokierung

Hauptprogramm mit dynamischer Speicherplatz- allokierung

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    extern void bubble_sort(int list[], int list_size);
    int *list, sort_num, j;

    printf("How many numbers are you going to enter? ");
    scanf("%d", &sort_num);

    list = (int *) malloc (sort_num * sizeof(int) );

    for( j=0; j < sort_num; j++)
        scanf( "%d", list + j );
    bubble_sort( list, sort_num );

    free(list);      /* Gib Speicher wieder frei */

    exit( 0 );
}
```

malloc() ist definiert als

```
void *malloc (size_t size)
```

Liefert *generischen* (untypisierten) Pointer zurück, der über einen *typecast* umgewandelt wird.



10. Strukturen und Unionen

10.1. Strukturen

10.2. Verkettete Listen

10.3. Unionen

10.4. Einige Bemerkungen zum Programmierstil

Strukturen

Personaldaten der Mitarbeiter einer Firma, bestehend aus Komponenten sehr unterschiedlicher Typen:

```
typedef enum {led, verh, gesch, verw} STANDTYP;

struct person
{
    char        name[30], vorname[30];
    struct datum geb_tag;
    int         pers_nr;
    STANDTYP    stand;
    char        adresse[80];
};
```

Bereits zuvor muß dabei der Datentyp *struct datum* vereinbart worden sein:

```
struct datum
{
    char    tag;
    char    monat;
    short   jahr;
};
```

Etikett,
tag name

Komponenten,
fields, members

Die bisherigen Definitionen bezeichnet man auch als *structure templates*.

Variablendefinition

```
struct person mitarbeiter1;
struct person mitarbeiter[100], *pma;
```

Letzteres definiert ein Array, wobei jedes Array-Element eine Struktur ist, und einen Zeiger auf eine solche Struktur.

Strukturen

Definitionsmöglichkeiten

Gleichzeitig eine Variable definieren:

```
struct person
{
    char        name[30], vorname[30];
    struct datum geb_tag;
    int         pers_nr;
    STANDTYP    stand;
    char        adresse[80];
} mitarbeiter[100];
```

tag name weglassen und sofort eine Variable definieren:

```
struct
{
    char        name[30], vorname[30];
    struct datum geb_tag;
    int         pers_nr;
    STANDTYP    stand;
    char        adresse[80];
} mitarbeiter[100];
```

Definition eines Datentyps:

```
typedef struct
{
    char        name[30], vorname[30];
    struct datum geb_tag;
    int         pers_nr;
    STANDTYP    stand;
    char        adresse[80];
} PERSON;
```

Variablen werden dann wie folgt definiert:

```
PERSON mitarbeiter1;
PERSON mitarbeiter[100], *pma;
```

Strukturen

Definition in Header-Dateien

Üblicherweise werden Strukturtypen in Header-Dateien vereinbart, so daß sie in mehreren Quelldateien verwendet werden können.

Speicherung

Die Komponenten einer Struktur werden in der angegebenen Reihenfolge gespeichert. Es können jedoch Lücken zwischen den einzelnen *members* im Speicher sein.

Noch ein Beispiel

```
typedef struct
{
    float realteil, imagteil;
} KOMPLEX;
```

Initialisierung von Strukturen

Nur möglich bei Definition konkreter Variablen mit Speicherallokation:

```
KOMPLEX z1 = {3.5, 7.9};
PERSON mitarbeiter1 = {"Müller", "Paul",
                       {5, 8, 1956},
                       4711,
                       verh,
                       "Jahnstr. 5, 65196 Wiesbaden"};
```

Zugriff auf Strukturkomponenten

Unterschiedlich für

- Strukturen selbst oder
- Zeiger auf Strukturen.

Jeweils verschiedene Operatoren.

Unterschied zwischen *array* und *structure*:

Zugriff auf Array-Komponenten durch *Indizes*, auf Struktur-Komponenten durch *Feld-Namen*.

Zugriff über Struktur selbst

```
...  
PERSON neuer;  
PERSON angest[max_pers_nr];  
...  
neuer.name           = "Müller";  
neuer.vorname        = "Horst";  
neuer.geb_tag.tag    = 12;  
neuer.geb_tag.monat  = 6;  
neuer.geb_tag.jahr   = 1955;  
neuer.pers_nr        = 4711;  
neuer.stand          = verh;  
neuer.adresse        = "Lutherstr. 45, 65196 Wiesbaden";  
  
angest [neuer.pers_nr] = Neuer;  
...
```

Zugriff über Zeiger

```
pma->name = "Müller";
```

Pfeilnotation ist Kurzschreibweise für Dereferenzierung und
Punktoperator:

```
pma->name      entspricht    (*pma).name
```

Arrays von Strukturen

person.h enthalte die folgenden Definitionen:

```
typedef struct
{
    char    tag;
    char    monat;
    short   jahr;
} DATUM;

typedef struct
{
    char          name[30], vorname[30];
    DATUM         geb_tag;
    int           pers_nr;
    STANDTYP     stand;
    char          adresse[80];
} PERSON;
```

Anwendung

Bestimme Anzahl an Personen in einer bestimmten Altersgruppe:

```
#include "person.h" /* Contains declaration of PERSON */

int agecount( PERSON par [], int size, int low_age,
              int high_age, int current_year)
{
    int i, age, count = 0;

    for (i = 0; i < size; ++i)
    {
        age = current_year - par[i].geb_tag.jahr;
        if (age >= low_age && age <= high_age)
            count++;
    }
    return count;
}
```

Arrays von Strukturen

Alternative: Verwendung eines Zeigers

```
#include "person.h" /* Contains declaration of PERSON */

int agecount ( PERSON par[], int size, int low_age,
              int high_age, int current_year)
{
    int i, age, count = 0;

    for (i = 0; i < size; ++par, ++i)
    {
        age = current_year - par->geb_tag.jahr;
        if (age >= low_age && age <= high_age)
            count++;
    }
    return count;
}
```

Der Zeiger *par* ist durch die Anweisung *++par* um die der Größe der Struktur entsprechende Anzahl an Bytes inkrementiert. Dies ist lediglich eine Addition.

Aus Sicht des Programmierstils ist es nicht so optimal, wie hier einen formalen Parameter zu verändern. Technisch ist es o.k., da dies eine Kopie des Originals im übergeordneten Programm ist.

Arrays von Strukturen

Bessere Alternative: Verwendung eines lokalen Zeigers

```
#include "person.h" /* Contains declaration of PERSON */

int agecount ( PERSON par[], int size, int low_age,
              int high_age, int current_year)
{
    int age, count = 0;
    PERSON *p, *p_last = &par[size];

    for ( p = par; p < p_last; ++p)
    {
        age = current_year - p->geb_dat.jahr;
        if (age >= low_age && age <= high_age)
            count++;
    }
    return count;
}
```

Namensräume von Strukturen und Unionen

Der ANSI-Standard verlangt, daß der Compiler für verschiedene Strukturen unterschiedliche Namensräume definiert.

D.h. zwei verschiedene Strukturen können namensgleiche Komponenten besitzen:

```
struct s1 {  
    int a, b;  
};  
struct s1 {  
    float a, b;  
};
```

Selbstreferenzierende Strukturen

Struktur darf keine Instanzen von sich selbst besitzen, aber sie kann einen **Zeiger auf eigene Instanzen** enthalten:

```
struct s {
    int a, b;
    float c;
    struct s *pointer_to_s; /* Ist legal */
}
```

- **pointer_to_s* wird hier verwendet, ohne daß er zuvor deklariert wurde.
- Referenziert das (schon bekannte) Etikett der Struktur
- Dies ist für Zeiger auf Strukturen in der gezeigten Notation erlaubt.

Anderes Beispiel

```
struct s1
{
    int a;
    struct s2 *b;
};
struct s2
{
    int a;
    struct s1 *b;
};
```

Wechselseitige Referenzen der beiden Strukturen. Zeiger auf s2 wird benutzt bevor s2 definiert wurde.

Man bezeichnet dies als eine *Vorwärtsreferenz*.

Eine solche Vorwärtsreferenz ist jedoch in Form einer *typedef* Definition *nicht* erlaubt:

```
typedef struct
{
    int a;
    STRUKTUR *p; /* STRUKTUR ist noch nicht bekannt: Falsch!
*/
} STRUKTUR;
```

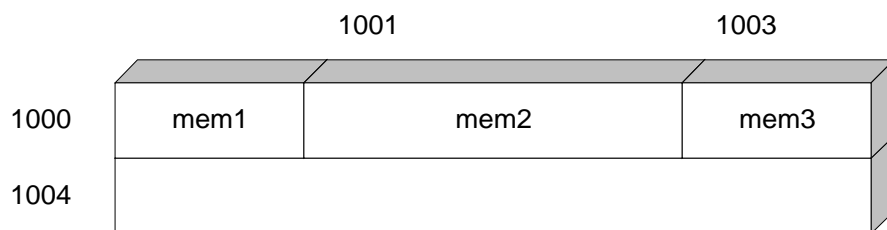

Alignment von Strukturenkomponenten

Natürliches Alignment: Adresse der Komponenten ist durch die Länge der Komponenten in Bytes ohne Rest teilbar.

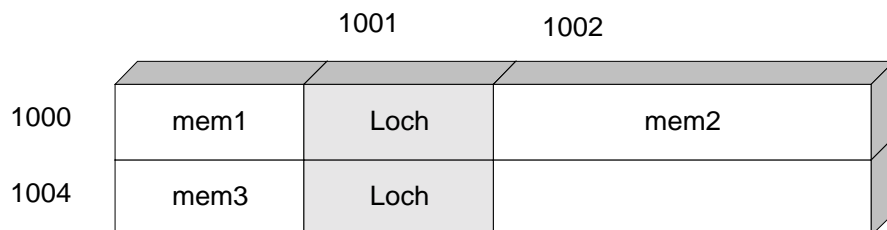
Beispiel

```
struct align_example
{
    char  mem1;
    short mem2;
    char  mem3;
} s1;
```

Ohne Alignment-Vorschriften



„Natürliches„ Alignment



Alignment von Strukturenkomponenten

- Normalerweise uninteressant für den Programmierer, es sei denn er versucht irgendwelche Tricks.
- Wichtig jedoch, wenn solche Strukturen in eine Datei geschrieben werden und – eventuell unter verschiedenen Alignment-Bedingungen – wieder gelesen werden sollen.

In bestimmten Fällen ist dieses Problem durch Umstrukturierung vermeidbar, wie in unserem Beispiel:

```
struct align_example
{
    char  mem1;
    char  mem3;
    short mem2;
} s1;
```

offsetof Makro

Vordefinierter Makro nach dem ANSI-Standard
Ermittelt Byte-Offset einer Komponente einer Struktur

```
offsetof( strukturtyp, komponenten_name )
```

Der Resultattyp davon ist *size_t*.

Alles ist in der Header-Datei *stddef.h* definiert.

```
#include <stddef.h>
```

```
typedef struct  
{  
    char ding_name [MAX_NAME];  
    int  ding_anzahl;  
    enum DING_TYP ding_art;  
} DING_INFO;
```

```
...
```

```
size_t art_offset = offsetof (DING_INFO, ding_art);
```

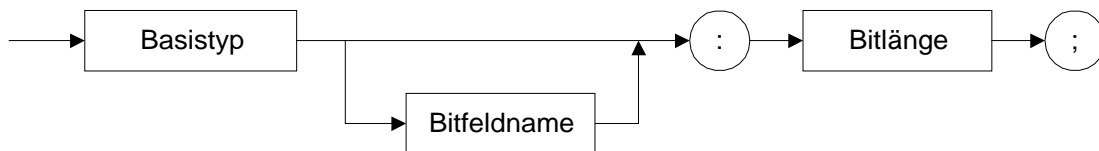
So kann man z.B. herausfinden, wie der Compiler Komponenten anordnet.

Bit-Felder

- Die kleinste Speichereinheit in C ist ein Byte (*char*).
- In Strukturen kann man jedoch noch kleinere Komponenten definieren.
- Man bezeichnet diese als Bit-Felder (*bit fields*).

Syntaxdiagramm:

Bitfeld-Deklaration:



- *Basistyp*: *int*, *unsigned int* oder *signed int*.
- Bei *int* ist signed oder unsigned implementierungsabhängig.
- Viele Compiler erlauben als *Basistyp* auch *enum*, *char* oder *short*!
- Nicht benannte Bitfelder können nicht zugegriffen werden und dienen nur dem Auffüllen von Leerstellen.
- Sonderfall: Unbenanntes Bitfeld mit Bitlänge 0 bewirkt, daß die nächste Komponente der Struktur an der nächsten *int*-Grenze beginnt.
- Die Bitlänge darf nicht länger als ein *int* sein.
- Als minimale Größe wird ein *char* allokiert, möglicherweise aber auch mehrere.

Bit-Felder

Gesamtgröße

Die Struktur

```
struct
{
    int a : 3;
    int b : 7;
    int c : 2;
} s;
```

paßt in ein 16-Bit Datenobjekt.

Die Implementierung ist jedoch frei, die Bits links- oder rechtsbündig anzuordnen.

Weiteres Beispiel

Annahme: 16-Bit *ints*:

```
struct
{
    int a : 10;
    int b : 10;
} s;
```

Hier kann *b* möglicherweise erst in einem zweiten *int* beginnen.

Bit-Felder

Anwendungen

- Man muß ernsthaft Speichergröße sparen (heute kaum mehr gegeben)
- Man muß eine extern vorgegebene Bitstruktur abbilden (PDV)

Beispiel

Möglichst kompakte Speicherung eines Datums:

```
typedef struct
{
    unsigned int tag    : 5;    /* max 31 */
    unsigned int monat  : 4;    /* max 15 */
    unsigned int jahr   : 11;   /* max 2047 */
} DATUM;
```

Hier genügen 20 Bits.

Ob der Compiler dafür 24 oder 32 Bits allokiert, ist nicht definiert.

Auch die Verteilung der einzelnen Komponenten ist nicht spezifiziert.

Parameterübergabe von Strukturen

Zwei grundsätzlich verschiedene Arten der Parameterübergabe:

- Die ganze Struktur selbst (als Kopie).
Parameterübergabe als Wert (*pass by value*)
- Ein Zeiger auf die Struktur.
Parameterübergabe als Referenz (*pass by reference*)

Beide Alternativen

```
PERSON ma;  
...  
func ( ma );    /* Werteparameter: Kopie der ganzen Struktur */  
func ( &ma );  /* Übergabe als Referenz: Adr. der Struktur */  
...
```

Referenzübergabe

Normalerweise schneller, aber das Original kann verändert werden.

Werteübergabe

Angezeigt, wenn

- Struktur klein ist (nicht viel größer als ein Zeiger) und sie nicht verändert werden soll
- man 100%-ig sicher sein muß, daß das Original nicht verändert werden kann (Alternative: Übergabe als *const*).

Parameterübergabe von Strukturen

Je nach Art der Parameterübergabe muß der Empfänger (die Funktion) unterschiedlich deklariert sein:

```
void func (PERSON mitarbeiter)          /* Pass by value */
```

oder

```
void func (PERSON *p_mitarbeiter)      /* Pass by reference */
```

oder auch (besser, wir werden später sehen)

```
void func (const PERSON *p_mitarbeiter)
                                   /* Pass by reference, */
                                   /* schreibgeschützt!  */
```


Parameterübergabe von Strukturen

Achtung:
Inkonsistenz in C bezüglich Behandlung von Arrays und
Strukturen!

```
int          ar[100];
struct tag st;
...
func ( ar ); /* Übergib Zeiger auf 1. Element des Arrays */
func ( st ); /* Übergib Kopie der ganzen Struktur */
```

Empfängerseite für das Array:

```
void func( int ar[] ); /* Erwarte einen Zeiger auf int */
void func( int *ar )   /* Erwarte einen Zeiger auf int */
```

Und für die Struktur:

```
void func( struct tag st )
                /* Erwarte die gesamte Struktur      */
void func( struct tag *st )
                /* Erwarte Zeiger auf die Struktur */
```

Struktur als Rückgabewert einer Funktion

Als Funktionswert können sowohl eine Struktur als auch ein Zeiger auf eine Struktur zurückgegeben werden.

```
struct tag f(void)    /* Funktion gibt eine Struktur */
{                    /* als Funktionswert zurück    */
    struct tag st;
    ...
    return st;        /* Gib gesamte Struktur zurück */
}
```

Oder

```
struct tag *f1(void) /* Funktion gibt Zeiger auf Struktur */
{                  /* als Funktionswert zurück          */
    static struct tag st;
    ...
    return &st;    /* Zeiger auf Struktur */
}
```

Wertzuweisung zwischen Strukturen

Gemäß ANSI C kann man zwei Struktur-Variablen einander zuweisen, soweit sie denselben Datentyp haben.

```
struct
{
    int a;
    float b;
} s1, s2, sf(void), *ps;

...
s1 = s2;        /* Struktur zu Struktur */
s2 = sf();       /* Funktionswert zu Struktur */
ps = &s1;        /* Strukturadresse zu Zeiger auf Struktur */
s2 = *ps;        /* Dereferenzierter Zeiger zu Struktur */
```

Verkettete Listen

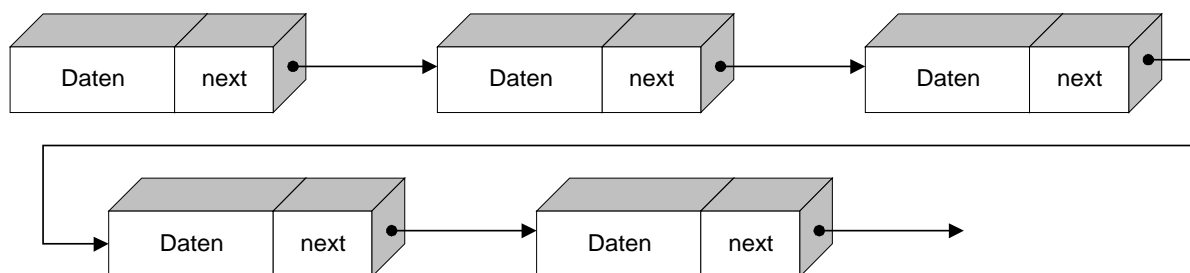
- Verkettung einzelner Elemente.
- Enthalten jeweils einen Zeiger auf das nächste Element

```
typedef enum {led, verh, gesch, verw} STANDTYP;
```

```
typedef struct  
{  
    char    tag;  
    char    monat;  
    short   jahr;  
} DATUM;
```

```
typedef struct person  
{  
    char          name[30], vorname[30];  
    DATUM         geb_tag;  
    int           pers_nr;  
    STANDTYP      stand;  
    char          adresse[80];  
    struct person *next;  
} PERSON;
```

Als Bild:



Verkettete Listen

Funktionen

- Erzeuge ein neues Listenelement
- Füge ein Listenelement an das Ende der Liste an
- Füge ein Listenelement in die Mitte der Liste ein
- Lösche ein Element aus der Liste
- Finde ein bestimmtes Listenelement in der Liste

Alle Funktionen, bis auf die letzte, können als C-Funktionen so allgemein formuliert werden, daß sie unabhängig von den internen Details der Daten funktionieren.

Verkettete Listen

Typendeklaration leicht modifiziert: *element.h*

```
typedef enum {led, verh, gesch, verw} STANDTYP;

typedef struct
{
    char    tag;
    char    monat;
    short   jahr;
} DATUM;

typedef struct person
{
    char          name[30], vorname[30];
    DATUM         geb_tag;
    int           pers_nr;
    STANDTYP      stand;
    char          adresse[80];
} PERSON;

typedef struct element
{
    PERSON ps;
    struct element *next; /* Hier muß der tag name    */
                        /* benutzt werden */
} ELEMENT;
```

Erzeugen eines Listenelementes

Hierzu allokieren wir den benötigten Speicherplatz mit *malloc()* und geben den darauf zeigenden Zeiger zurück.

```
#include "element.h"
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>

ELEMENT *create_list_element()
{
    ELEMENT *p;

    /* Erzeuge Datenobjekt vom Typ ELEMENT */
    p = (ELEMENT *) malloc( sizeof ( ELEMENT ) );

    if (p == NULL) /* Kein Erfolg */
    {
        printf( "create_list_element: malloc failed.\n");
        exit( 1 );
    }

    p->next = NULL; /* Vorbesetzung */
    return p;
}
```

Hinzufügen eines Listenelementes

Diese Funktion fügt das mit *create_list_element()* erzeugte Element an das Ende der Liste an. Dazu wird ein Zeiger mit dem neuen Element übergeben:

```
#include "element.h"

static ELEMENT *head; /* File scope */

void add_element( ELEMENT *e)
{
    ELEMENT *p;
    /* Ist die Liste noch leer (head==NULL), setze head
     * auf das neue Element.
     */
    if (head == NULL)
    {
        head = e;
        return;
    }

    /* Anderenfalls finde das letzte Element in der Liste */
    for (p = head; p->next != NULL; p = p->next)
        ; /* leere Anweisung */

    p->next = e; /* Hänge neues Element an bisher letztes */
}
```

Die Variable *head* mit *file scope* ist das „Gedächtnis“, dieser Sammlung von Funktionen.

Alle Funktionen arbeiten mit diesem Kopf der Liste.

Für Funktionen außerhalb der Datei ist *head* jedoch nicht sichtbar (Modul-Eigenschaft).

Verkettete Listen

Hauptprogramm

Erzeuge eine Liste mit 10 (leeren) Elementen:

```
int main(void)
{
    int j;

    for (j=0; j < 10; ++j)
        add_element ( create_list_element() );
}
```

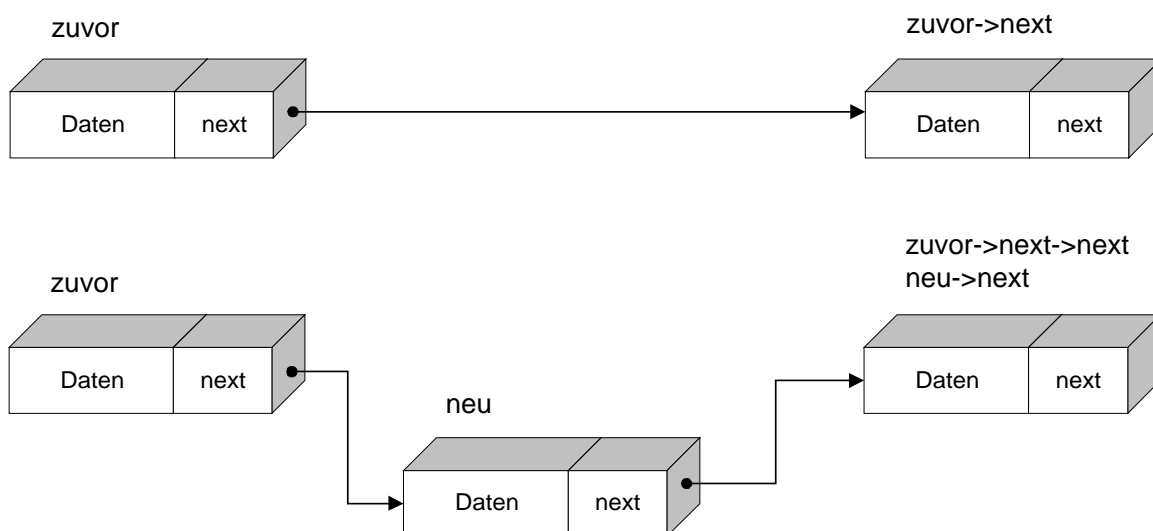
Diese Quelldatei müßte zusammen mit der Quelldatei der Liste übersetzt und gelinkt werden.

Einfügen eines Listenelementes

Annahme, wir wüßten die Stelle, an der das neue Element in die Liste eingefügt werden soll, in Form eines Zeigers auf das Element zuvor.

```
/* insert neu after zuvor */  
  
#include "element.h"  
  
void insert_after( ELEMENT *neu, ELEMENT *zuvor)  
{  
    /* Prüfe Argumente auf Plausibilität.  
     * Wenn neu und zuvor beide gleich oder NULL sind, oderr wenn  
     * neu schon hinter zuvor steht, melde Fehler.  
     */  
    if (neu == NULL || zuvor == NULL || neu == zuvor ||  
        zuvor->next == neu)  
    {  
        printf( "insert_after: Ungültige Argumente\n" );  
        return;  
    }  
  
    neu->next = zuvor->next;  
    zuvor->next = neu;  
}
```

Als Bild:



Löschen eines Listenelementes

Um dies bewerkstelligen zu können, muß das Element *vor* dem zu löschenden bekannt sein:

```
#include "element.h"
static ELEMENT *head;

void delete_element( ELEMENT *loesch_mich)
{
    ELEMENT *p;    /* Hilfszeiger */

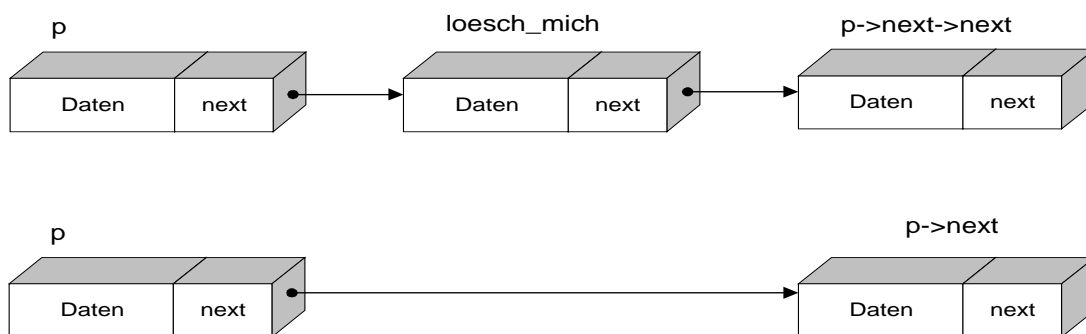
    if (loesch_mich == head)    /* Kopfelement ist zu löschen */
        head = loesch_mich->next;
    else
    {
        /* Finde Element, das vor dem zu löschenden steht */

        for (p = head; (p != NULL) && (p->next != loesch_mich);
              p = p->next)
            ; /* null statement */

        if (p == NULL)    /* Nicht gefunden */
        {
            printf( "delete_element: Kann Element in der Liste"
                    "nicht finden.\n");
            return;
        }

        p->next = p->next->next;
    }
    free(loesch_mich);    /* Gib Speicher von loesch_mich frei */
}
```

Als Bild:



Finden eines Listenelementes

- Dies ist jetzt nicht so einfach inhaltsunabhängig für beliebige Elemente zu formulieren.
- Wir behelfen uns für diesen Fall, daß wir nach einem passenden Nachnamen suchen.

```
#include "element.h"
static ELEMENT *head;

ELEMENT *find( char *name )
{
    ELEMENT *p;

    for (p = head; p != NULL; p = p->next)
        if (strcmp(p->ps.name, name) == 0)
            return p;
    return NULL;
}
```

Unionen

Beispiel

Koordinaten: Kartesische oder Polar-Koordinaten

```
typedef union
{
    struct
    {
        float x, y;
    } kart;

    struct
    {
        float r, phi;
    } polar;
} KOORD;
KOORD beispiel;
```

Beide Koordinatentypen weisen auf denselben Speicher. Zugriff auf die Komponenten der Union:

```
beispiel.kart.x = 5.0;
beispiel.kart.y = 10.0;
```

bzw.

```
beispiel.polar.r = 3.0;
beispiel.polar.phi = 90.0;
```

Unionen

Beispiel mit unterschiedlich großen Komponenten

```
typedef union
{
    struct
    {
        char c1, c2;
    } s;
    long j;
    float x;
} U;
U beispiel;
```

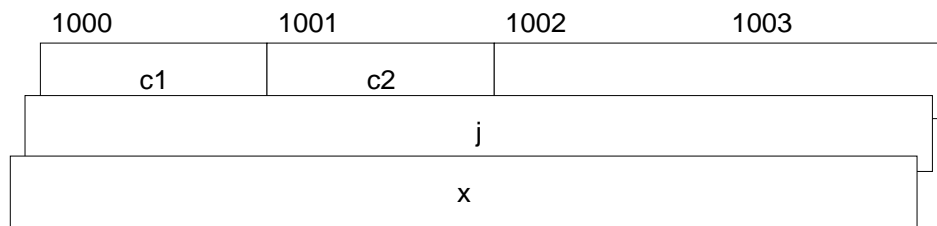
Zugriff hier:

```
beispiel.s.c1 = 'a';
beispiel.s.c2 = 'b';
```

```
beispiel.j      = 95;
```

Die dritte Zuweisung würde die zuvor gespeicherten beiden *chars* überschreiben.

Anschaulich:



Unionen

Initialisierung

Unions können (lt. ANSI) initialisiert werden. Die Initialisierung betrifft immer die erste Komponente:

```
union init_beispiel
{
    int i;
    float f;
};
union init_beispiel test = { 1 };    /* test.i erhält Wert 1 */
```

Oder

```
union u
{
    struct { int i; float f; } s;
    char ch[6];
};
union u test = { {1, 10.0} };    /* Äußere Klammern können
entfallen */
```

Einige Bemerkungen zum Programmierstil (1/2)

- Professionelles Programmieren ist eine anspruchsvolle Tätigkeit die man nicht *mit links* machen kann. Es erfordert eine ingenieurmäßige Vorgehensweise:
 - Fassen Sie die Aufgabenstellung schriftlich zusammen. Vergewissern Sie sich, daß Sie den Auftraggeber richtig verstanden haben. Es ist ärgerlich, vergeblich für ein Mißverständnis gearbeitet zu haben.
 - Stellen Sie den Lösungsalgorithmus verbal, durch Pseudo-Code oder auch graphisch (durch Nassi-Shneidermann Struktogramme) dar.
- Erstellen Sie keine *mundfaulen* Programme. Machen Sie Gebrauch von printf()- und scanf()-Anweisungen. Legen Sie Wert auf klare, leichte Benutzerführung. Sagen Sie dem Benutzer genau, wie die Eingabe sein muß und was sie bedeutet.
- Versuchen Sie Benutzer-Fehler (Eingabe-Fehler) abzufangen. Lassen Sie das nicht das Betriebssystem tun. Ein Abbruch des Programms durch das Betriebssystem ist hart und verunsichert den Benutzer.

Einige Bemerkungen zum Programmierstil (2/2)

Das Programm sollte selbstdokumentierend und ansprechend zu lesen sein:

- *Steckbrief* im Programmkopf
 - Name des Programms
 - Funktion: Was macht es? Erklärung im Telegrammstil.
 - Versionsnummer, Datum der letzten Änderung, evtl. Historie der Änderungen
 - Autor und Datum der Ersterstellung
- Gliederung des Programmtextes durch Leerzeilen und Einrückungen.
- Sinnvolle Kommentare in ausreichendem Umfang (auf 1 Anweisung ein Kommentar ist zuviel, auf 10 Anweisungen einer kann zuwenig sein)
 - Benutzen Sie dazu die Sprache des *fachlichen* Problems, nicht die der DV-mäßigen Implementierung
 - Kommentare müssen mit den Anweisungen synchron sein
- Verwenden Sie sinnvolle, sprechende Namen, nicht einfach einzelne Buchstaben.
- Seien Sie vorsichtig bei der Verwendung spezieller Eigenschaften der Programmiersprache. Ist die von Ihnen gewählte Konstruktion allgemeinverständlich?
- Verwenden Sie die für das Problem am besten geeigneten *strukturierten* Kontrollstrukturen, vermeiden Sie gotos!
- Machen Sie Ihre Datenstrukturen so wenig komplex wie möglich

11. Unterprogramm-Techniken (Routinen)

- 11.1. Grundbegriffe
- 11.2. Funktionen
- 11.3. Parameter-Übergabe
- 11.4. Vereinbarung und Aufruf
- 11.5. Regeln
- 11.6. Zeiger auf Funktionen
- 11.7. Die main()-Funktion
- 11.8. Komplexe Deklarationen

Routinen (Grundbegriffe)

Umfangreiche Programme können sehr *kompliziert* werden:

Die Kontrollstrukturen bedingen möglicherweise tief verschachtelten Code.

Dies macht die Programme *unübersichtlich*, *schwer wartbar* und *fehleranfällig*.

Folgerung: Man muß Programme, ähnlich wie ein Buch in Kapitel, in kleinere Einheiten aufteilen: Unterprogramme.

Dadurch werden Programme nicht nur

übersichtlicher,

man kann auch ein einmal formuliertes Unterprogramm

an anderer Stelle wiederverwenden (reusable Code).

Grundbegriffe

Durch die Unterprogramm-Technik wird die Lösung eines umfangreichen, komplexen Problems so lange in kleinere Teile zerlegt, bis diese *überschaubar* geworden sind. Diese Einheiten sollen weitgehend *in sich logisch abgeschlossen* sein.

Sie stellen *selbständige Teillösungen* dar. Musterbeispiel sind die *Software-Tools* im Umkreis der Unix-Entwickler (Kernighan, Plauger, Ritchie): „keine Routine länger als eine Seite“.

Routinen: Sammelbegriff für *Prozeduren*, *Funktionen* und *Subroutinen*, auch *Unterprogramme*. In C sprechen wir immer von *Funktionen*.

Routinen (Grundbegriffe)

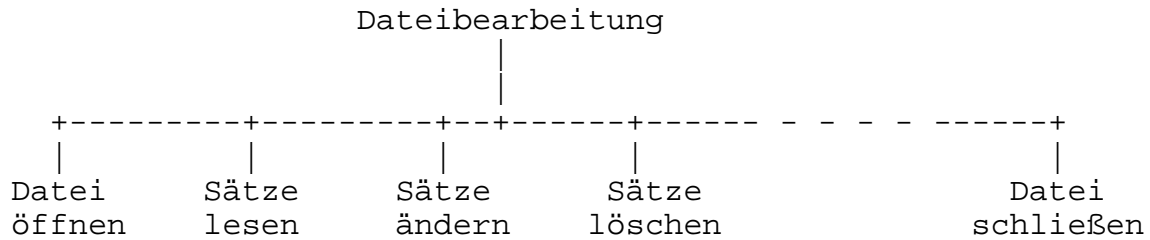
Formulierung von Unterprogrammen im Rahmen einer Quelldatei:

- **Definition/Deklaration globaler Variablen (*program scope*)**
- **Definition von Variablen mit *file scope***
- **Deklarationen externer Funktionen**
- **Definition und Implementierung von Funktionen**
 - Formale Parameter
 - Lokale Variablen (automatisch oder statisch)
 - Lokale benutzte externe Funktionen oder Variablen
 - Funktions-Anweisungen (Funktionsrumpf)
- **Definition und Implementierung der main-Funktion**
 - Struktur wie bei Funktionen

Unterprogramme haben i.a. ein eigenes Innenleben, welches weitgehend nach außen verborgen sein soll. Nach außen nur das sichtbar, was unbedingt notwendig ist.

Es gibt *lokale* und *globale* Variablen.

Funktionen



Programm schematisch

```

/* Globale Vereinbarungen */
int i_global = 1;
static i_file;
typedef ...

/* Funktionen */
void datei_oeffnen(void)
{
  const int i_const;      { Lokale Vereinbarungen}
  float x;

  ...
  ...      { Anweisungen der Funktion }
};

void saetze_lesen(void)
...

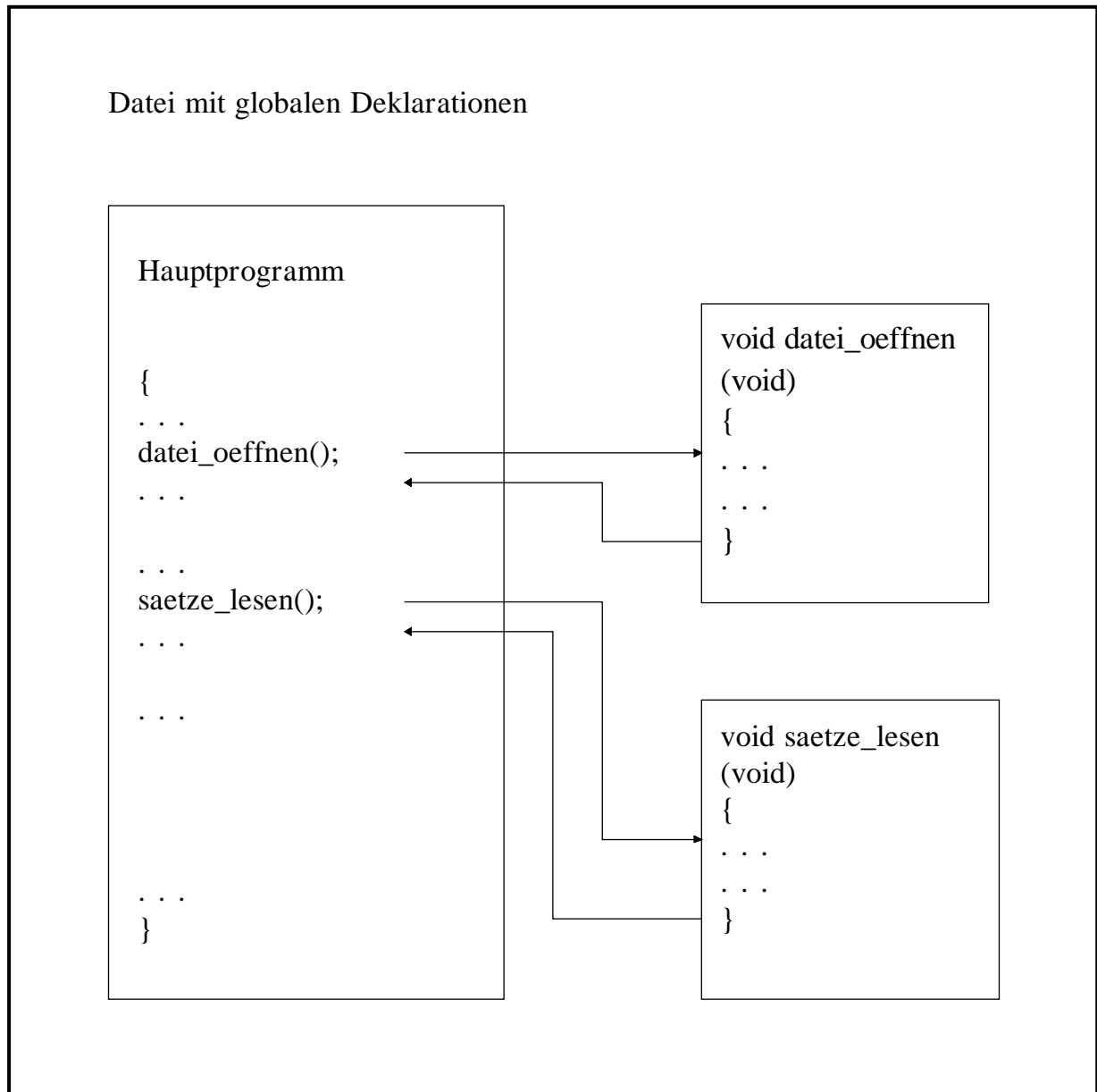
void saetze_aendern(void)
...

void saetze_loeschen(void)
...

void datei_schliessen(void)
...
int main(void)
{
  ...      /* Lokale Definitionen des Hauptprogramms */
  ...
  datei_oeffnen();      /* Ausführungsteil des Hauptprogramms */
  saetze_lesen();      /* mit Aufruf der Funktionen */
  saetze_aendern();
  saetze_loeschen();
  ...
  datei_schliessen();
  exit(0);
}      /* Ende Hauptprogramm */
  
```

Funktionsaufruf

Graphisch dargestellt:



Werteparameter

- Parameter werden *in Kopie* an das Unterprogramm weitergereicht.
- Es werden Werte übergeben, nicht etwa die Adressen der Originale.
- Die Original-Parameter im rufenden Programm können durch das Unterprogramm *nicht* modifiziert werden.

Es wird tatsächlich eine Kopie erstellt, d.h. der Speicherplatz muß zweimal aufgebracht werden. Daher wird man i.a. keine großen Datenstrukturen wie Arrays als Werteparameter übergeben.

In C haben wir überwiegend Werteparameter!

Ein Beispiel

```
#include <stdio.h>
#include <stdlib.h>

void f( int received_arg )
{
    received_arg = 3;    /* Weise Kopie den Wert 3 zu */
}

int main(void)
{
    int a = 2;

    f( a );    /* Übergib Kopie von a */
    printf("%d\n", a);
    exit(0);
}
```

Das Ergebnis der *printf()*-Anweisung ist 2.

Der Wert von a kann im Hauptprogramm durch die Funktion nicht geändert werden.

Referenzparameter

- Man setzt *call by reference* ein, wenn Änderungen an den Parameterwerten innerhalb des aufgerufenen Unterprogrammes der aufrufenden Programmeinheit **nicht** verborgen bleiben soll.
- Oder es sollen ausdrücklich *Ergebnisse* zurückgeliefert werden.
- Diese können durch Modifikation der Eingabeparameter (dies sind dann Ein- und Ausgabeparameter) oder auch über separate Ausgabeparameter geliefert werden.

Tatsächlich wird beim Aufruf nur die Adresse der entsprechenden Speicherstelle des Parameters übergeben.

Durch sog. *indirekte Adressierung* kann dann die Funktion auf den Originalspeicherplatz in der rufenden Programmeinheit zugreifen und diesen verändern.

Damit wird in C ein *call by reference* simuliert, formal ist es nach wie vor ein *call by value*.

- Es wird der Wert einer Adresse (also ein Zeiger) übergeben.
- Auch dies ist eine Kopie der Originaladresse.
- Diese kann im rufenden Programm ebenfalls nicht geändert werden.

```
/* Swap the values of two int variables */
```

```
void swap(int *x, int *y)
{
    register int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

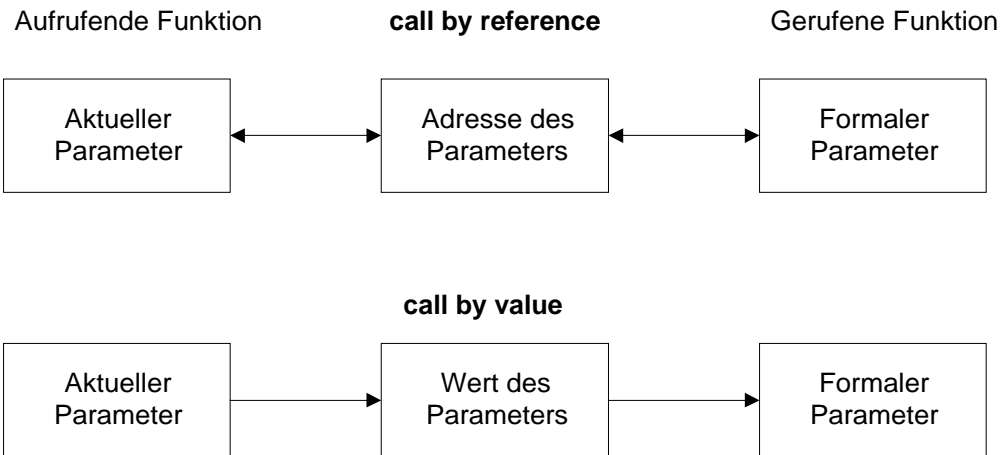
Der Aufruf muß hier wie folgt geschehen:

```
int main( void )
{
    int a = 2, b = 3;
    swap ( &a, &b );
    printf( "a = %d\t b = %d\n", a, b );
}
```

Das Ergebnis ist

```
a = 3      b = 2
```

Werte- und Referenzparameter



- Inkonsistente Ausnahme bei C:
Felder (*arrays*) werden immer über ihre Adresse übergeben.
Übergabe des Feldnamens entspricht also einem Referenzaufruf.
- Will man auch bei einem *call by reference* die Veränderung des Originals verhindern, so kann man mit ANSI-C die Typmodifikation *const* benutzen.

Aktuelle und formale Parameter

- Parameter, die im rufenden Programm an das Unterprogramm übergeben werden, heißen **aktuelle Parameter**.
- Bei der Deklaration des Unterprogramms müssen Platzhalter bereitgestellt werden, unter denen die aktuellen Parameter *formal* in der Prozedurvereinbarung geführt werden. Man nennt sie **formale Parameter**. Bei Laufzeit werden die formalen Parameter durch die aktuellen (mit konkreten Werten) ersetzt.

Beispiel

Funktionskopf:

```
void position(int x, int y);
```

Dabei sind die formalen Parameter:

```
(int x, int y)
```

Aufruf durch aktuelle Parameterliste:

```
position(2, 4)    /* 2 und 4 sind aktuelle Parameter */
```

oder auch

```
spalte = 2;  
zeile  = 3;  
position(spalte, zeile+1); /* spalte und zeile+1 sind */  
                           /* aktuelle Parameter      */
```

- Übergabeparameter können also Konstanten, Variablen oder auch Ausdrücke von entsprechendem Datentyp sein (dies gilt streng nur bei *Werteparametern*!).
- Sie ersetzen eins-zu-eins die formalen Parameter in der vereinbarten Reihenfolge. Wichtig ist, daß sie in *Anzahl* und *Datentypen* exakt mit den formalen Parametern übereinstimmen.

Vereinbarung und Aufruf von Funktionen

Funktionen kommen in einem Programm in drei Kontexten vor:

Definition	Definition der Schnittstelle der Funktion: Name, Parameter (Anzahl, Datentyp), Rückgabotyp und Implementierung der Funktion
(Externe) Deklaration, <i>allusion</i>	Deklariert, daß die Funktion woanders definiert ist. Geschieht dies in Form eines Funktionsprototyps, so sind damit Name, Parameter (Anzahl, Datentyp) und Rückgabotyp bekannt
Aufruf	Ruft Funktion auf. Parameterübergabe wird vorbereitet. Programmablauf wird in der Funktion fortgesetzt. Nach Rückkehr wird Programmablauf an der Stelle direkt nach dem Aufruf fortgesetzt. Evtl. Funktionswert steht zur Verfügung

Traditionelle Form der Funktionsdefinition

```
int func_def(c, f)
char *c;
double f;
{
    ...
}
```

Dies sollte heute bei neuen Programmen nicht mehr verwendet werden.

ANSI-C Standard: Prototype-Form

```
int func_def(char *c, double f)
{
    ...
}
```

Dies ist auch mit der (externen) Prototype-Deklarationssyntax identisch und sollte nur noch verwendet werden.

Funktionswert

- Der Typ des Rückgabewertes voreingestellt mit *int*
- Sollte zur Klarheit immer spezifiziert werden
- Gibt die Funktion keinen Wert zurück, so muß sie den Typ *void* erhalten

Achtung:

Generell sollten Funktionen keine *Seiteneffekte* haben. Dies wäre z.B. der Fall, wenn globale Daten, die nicht als Parameter übergeben wurden, genutzt oder manipuliert würden. Dies wird als *schlechter Programmierstil* angesehen und ist unbedingt zu vermeiden!

Regeln

Funktionsnamen

- Wahl des Funktionsnamens ist der Schlüssel, um lesbare Programme zu schreiben
- Soll möglichst gut das Wesen der Funktion wiedergeben (Abstraktion dessen, was intern abläuft)
- Bei numerischen Rückgabewerten den Namen so wählen, daß er sich im Kontext eines Ausdrucks gut liest

```
int get_buf_len()
```

- Bei booleschen Rückgabewerten Namen so wählen, daß ersich in Verbindung mit if-Anweisung gut liest

```
if ( has_correct_value(expression) )
```

Regeln

Parameter

- Nicht zu viele Parameter (7-er Regel)
- Sortiere Output- und Input-Parameter
- Als Speicherklasse ist nur *register* erlaubt
- Bei Funktionen mit Prototyp werden die aktuellen Parameter zu den Typen des Prototyps (den formalen Parametertypen) konvertiert (analog einer Wertzuweisung)
- Ein Array als formaler Parameter wird in einen Zeiger auf den Datenobjekt der Arraykomponenten konvertiert
- Eine Funktion als formaler Parameter wird in einen Zeiger auf die Funktion konvertiert
- Initialisierung ist bei formalen Parametern nicht erlaubt

Regeln

Funktionswerte

- Nur ein einzelner Wert
- Kann von jedem Typ außer Array oder Funktion sein
- Umfangreichere Datentypen können in Form von Zeigern auf Aggregattypen zurückgegeben werden
- Strukturen können auch als Ganzes zurückgegeben werden (ineffizient)
- Der Wert in der *return*-Anweisung muß zuweisungskompatibel mit dem Datentyp der Funktion sein

Regeln

Funktionsprototypen

- Deklaration einer Funktion, die an anderer Stelle definiert ist, üblicherweise in einer anderen Quelldatei
- Informiert den Compiler über Anzahl und Typen der Parameter und Typ des Rückgabewertes
- Default-Rückgabewert ist *int*
- Immer die ANSI-Form der Funktionsprototypen verwenden
Beispiel:

```
extern void func (int, float, char *);
```

besser:

```
extern void func (int a, float b, char *c);
```
- Stellt sicher, daß korrekte Anzahl und Typen von Parametern übergeben werden (evtl. geschieht stille Konversion zum erwarteten Datentyp)
- Verwende *void* für Funktionen ohne Parameter

```
extern int f(void);
```
- Guter Stil: Fasse alle Funktionsprototypen an einer Stelle zusammen
- Die voreingestellte Speicherklasse ist *extern*
- Funktionsdefinitionen sind üblicherweise global, durch Voranstellen der Speicherklasse *static* gilt die Funktion nur in der aktuellen Quelldatei, bleibt also nach außen verborgen!

Regeln

Funktionsaufrufe

- Übergibt Ablaufkontrolle an die aufgerufene Funktion, nachdem die Parameter übergeben wurden
- Ein Funktionsaufruf ist ein Ausdruck und kann überall vorkommen, wo Ausdrücke erlaubt sind
- Ergebnis des Ausdrucks ist der Rückgabewert der Funktion (Funktionswert); darf dann nicht *void* sein!
- Der Rückgabewert kann ignoriert werden (obwohl dies eigentlich kein guter Programmierstil ist!)
- Funktionen können eine variable Anzahl von Parametern haben. Beispiel ist die *printf()*-Funktion. Ihr Prototyp:

```
int printf(const char *format, ...);
```

Wegen Details siehe in der Laufzeitbibliothek unter *<stdarg.h>* die Makros *va_start* und *va_arg*, die Funktion *va_end()* und der Datentyp *va_list*.

Zeiger auf Funktionen

Wertzuweisung

```
...
extern int f1(void);

/* Deklariere pf als Zeiger auf eine Funktion, die int als
 * Funktionswert zurückgibt
 */
int (*pf) (void);
...
pf = f1; /* Weise Adresse von f1 dem Zeiger pf zu */
```

Falsche Konstruktionen

Mit Klammer wäre die Zeile ein Funktionsaufruf:

```
pf = f1(); /* Falsch: f1() liefert int, pf ist Zeiger */
```

Ebenfalls falsch ist

```
pf = &f1();
/* Falsch: Rechte Seite ist Adresse des Funktionswertes */
```

genauso wie

```
pf = &f1; /* Falsch: &f1 ist ein Zeiger auf einen Zeiger */
```

Allerdings erlaubt der ANSI-Standard diese Syntax und ignoriert das & einfach.

Zeiger auf Funktionen

Funktionswertübereinstimmung

Funktionswerte muß im Typ übereinstimmen, genauso wie Anzahl und Typ der Parameter

```
extern int if1(), if2(), (*pif)();
extern float ffl(), (*pff)();
extern char cfl(), (*pcf)();

int main( void )
{
    pif = if1; /* Legal      types match */
    pif = cfl; /* ILLEGAL   type mismatch */
    pff = if2; /* ILLEGAL   type mismatch */
    pcf = cfl; /* Legal      types match */
    if1 = if2; /* ILLEGAL   Assign to a constant */
}
```

Aufruf

```
...
extern int f1(int);

int (*pf) (int);
int a, answer;

pf = f1;
a = 25;
answer = (*pf) (a); /* Ruft Funktion f1 mit Parameter a auf */
...

```

Klammer um **pf* ist unbedingt notwendig!

Eine Besonderheit bei Zeigern auf Funktionen ist, daß beliebig viele „*“, zur Dereferenzierung angegeben werden können, ohne daß etwas anderes geschieht.

```
(*pf) (a)
ist dasselbe wie
(****pf) (a)
```

Tatsächlich kann man die Dereferenzierung (nach ANSI-Standard) völlig weglassen, so daß auch das folgende korrekt ist:

```
pf (a)
```

Zeiger auf Funktionen

Beispiel

Erstellung einer Wertetabelle einer beliebigen Funktion mit einem *float* Argument und Ergebnistyp *float*:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

float funk1(float x)
{
    if (x != -1.0)
        return 1.0/(1.0+x);
    else
        return 0;
}

float MySin(float x)
{
    return sin(x);
}

void tabelle(float von, float bis, float delta,
             float (*f) (float))
{
    float x;
    printf("  x-Wert      Funktion\n");
    x = von;
    do
    {
        printf("%8f %11f\n", x, f(x));
        x += delta;
    } while (x <= bis);
    printf("\n");
}

int main(void)
{
    float pi = 3.1415926;
    float (*funk) (float x);

    funk = funk1;
    Tabelle(0.0, 10.0, 1.0, funk);

    Tabelle(-pi/2, pi/2, pi/10, MySin);
    exit(0);
}
```

Funktionszeiger als Funktionswert

Suche beste Sortierfunktion aus:

```
void (*best_sort(float list1[])) (float list2[])
{
    extern void quick_sort(float list[]),
              merge_sort(float list[]),
              heap_sort(float list[]);

    /* Analysiere hier die Daten in list1 */

    /* Falls Quicksort das beste ist */
    return quick_sort;

    /* Anderenfalls, wenn Mergesort das beste ist */
    return merge_sort;

    /* Anderenfalls, wenn Heapsort das beste ist */
    return heap_sort;
}
```

Um ein Array zu sortieren würde man dann aufrufen:

```
void sort_array(float list[])
{
    extern void (*best_sort(float list1[])) (float list2[]);
    ...
    (best_sort(list)) (list);
}
```

Die obige Funktionsdeklaration versteht man besser, wenn man sie unter Zuhilfenahme einer *typedef*-Deklaration in zwei Teile zerlegt:

```
void (*best_sort(float list1[])) (float list2[])
...

typedef void *FP (float list2[]);
FP best_sort(float list1[])
...
```

Die main()-Funktion

Die Funktion *main()* bekommt vom Betriebssystem zwei Parameter übergeben:

- Der erste (*argc*) ist ein *int* und enthält die Anzahl der Parameter der Kommandozeile, mit der das Programm aufgerufen wurde,
- der zweite (*argv*) ist ein Array von Zeigern auf die Kommandozeilenparameter.

Das folgende Programm verwendet *argc* und *argv[]*, um die Liste der Kommandozeilenparameter auszugeben:

```
#include <stdio.h>
#include <stdlib.h>

/* Echo command line arguments */

int main( int argc, char *argv[])
{
    while(--argc > 0) /* Vermeide durch Prefix 1. Parameter */
        printf( "%s ", *++argv);          /* Ebenfalls */
    printf( "\n" );
    exit( 0 );
}
```

Die Kommandozeilenparameter werden immer als Strings übergeben. Sollen sie als Zahlen interpretiert werden, so müssen sie konvertiert werden:

```
/* Berechne arg1 hoch arg2 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( int argc, char *argv[])
{
    float x, y;
    if (argc < 3)
    {
        printf( "Usage: power <number> <number>\n" );
        printf( "Yields arg1 to arg2 power\n" );
        return;
    }
    x = atof( *++argv ); /* Konvertiere String zu float */
    y = atof( *++argv );
    printf( "%f\n", pow( x, y ) );
}
```

Komplexe Deklarationen

Die folgende Deklaration vereinbart x als

- einen Zeiger auf eine Funktion,
- die einen Zeiger auf ein 5-elementiges Array bestehend aus
- Zeigern auf *ints*

zurückliefert:

```
int *( *(*x) (void) ) [5];
```

Solch komplizierte Deklarationen kann man vermeiden, indem man als Zwischenschritte verschiedene *typedefs* vereinbart:

<code>typedef int *AP[5]</code>	5-elementiges Array aus Zeigern auf int
<code>typedef AB *FP(void)</code>	Funktion, die Zeiger auf obiges Array liefert
<code>FP *x</code>	Zeiger auf obige Funktion

Ursache für das kryptische Aussehen

- der Pointer-Operator ist ein *Prefix*-Operator
- Array- und Funktionsoperatoren sind *Postfix*-Operatoren

Weiterhin ist zu beachten

- Array- und Funktionsoperatoren (`[]` und `()`) besitzen eine *stärkere Bindung (precedence)* als der Pointer-Operator (`*`)
- die Array- und Funktionsoperatoren gruppieren sich von links her während der Pointer-Operator sich von rechts her gruppiert.

Die Bezeichner werden dazwischen gepackt. Um eine solche Deklaration aufzubauen oder auch nur zu verstehen, muß man also von *innen nach außen* vorgehen.

Entschlüsselung komplexer Deklarationen

Beispiel 1

```
char *x[ ];
```

würde wie folgt interpretiert:

- `x[]` ist ein **Array** (`[]` hat stärkere Bindung als der `*`)
- `*x[]` ist ein **Array** von **Zeigern**
- `char *x[]` ist ein **Array** von **Zeigern** auf *chars*

Beispiel 2

Klammerung spielt entscheidende Rolle:

```
int (*x[ ]) (void);
```

- `x[]` ist ein **Array**
- `(*x[])` ist ein **Array** von **Zeigern**
- `(*x[]) (void)` ist ein **Array** von **Zeigern** auf **Funktionen**
- `int (*x[]) (void)` ist ein **Array** von **Zeigern** auf **Funktionen** die als Wert *ints* zurückgeben

Ohne die Klammer

```
int *x[ ] (void);
```

wäre das

- ein **Array** von **Funktionen** die jeweils einen **Zeiger** auf einen *int* zurückgeben (was eine illegale Deklaration ist, da Arrays von Funktionen nicht erlaubt sind!)

Zusammensetzen komplexer Deklarationen

Folgende Deklaration ist zu konstruieren:

- Ein **Zeiger** auf ein **Array** von **Zeigern** auf **Funktionen**, die als Funktionswert **Zeiger** auf ein **Array** von **Strukturen** mit dem Etikett S zurückgeben
- (*x)
ist ein **Zeiger**
- (*x[])
ist ein **Zeiger** auf ein **Array**
- (*(x[]))
ist ein **Zeiger** auf ein **Array** von **Zeigern**
- (*(x[])) (void)
ist ein **Zeiger** auf ein **Array** von **Zeigern** auf **Funktionen**
- (* (*(x[])) (void))
ist ein **Zeiger** auf ein **Array** von **Zeigern** auf **Funktionen** die **Zeiger** zurückliefern
- (* (*(x[])) (void)) []
ist ein **Zeiger** auf ein **Array** von **Zeigern** auf **Funktionen** die **Zeiger** auf **Arrays** zurückliefern
- struct S (* (*(x[])) (void)) []
ist ein **Zeiger** auf ein **Array** von **Zeigern** auf **Funktionen** die **Zeiger** auf **Arrays** bestehend aus **Strukturen** mit dem Etikett S zurückliefern

Jedesmal, wenn ein neuer Pointer-Operator hinzukommt, wird dieser zur stärkeren Bindung mit Klammern umgeben.

Beispiele komplexer Deklarationen (1/3)

In der folgenden Tabelle sind legale und illegale mehr oder weniger komplexe Deklarationen wiedergegeben. Funktionsparameter wurden zur besseren Lesbarkeit weggelassen.

<code>int i</code>	Ein int
<code>int *p</code>	Ein Zeiger auf ein int
<code>int a[]</code>	Ein Array von ints
<code>int f()</code>	Eine Funktion mit Funktionswert int
<code>int *pp</code>	Ein Zeiger auf einen Zeiger auf ein int
<code>int (*pa) []</code>	Ein Zeiger auf ein Array von ints
<code>int (*pf) ()</code>	Ein Zeiger auf eine Funktion mit Funktionswert int
<code>int *ap []</code>	Ein Array von Zeigern auf ints
<code>int aa [] []</code>	Ein Array von Arrays auf ints
<code>int af [] ()</code>	Ein Array von Funktionen mit Funktionswert int (ILLEGAL)
<code>int *fp ()</code>	Eine Funktion mit einem Zeiger auf int als Funktionswert
<code>int fa () []</code>	Eine Funktion mit einem Array von ints als Funktionswert (ILLEGAL)
<code>int ff () ()</code>	Eine Funktion mit einer Funktion als Funktionswert die selbst ints als Funktionswert liefert (ILLEGAL)

Beispiele komplexer Deklarationen (2/3)

<code>int ***ppp</code>	Ein Zeiger auf einen Zeiger auf einen Zeiger auf ein int
<code>int (**ppa) []</code>	Ein Zeiger auf einen Zeiger auf ein Array von ints
<code>int (**ppf) ()</code>	Ein Zeiger auf einen Zeiger auf eine Funktion mit ints als Funktionswert
<code>int *(*pap) []</code>	Ein Zeiger auf ein Array von Zeigern auf ints
<code>int (*ppa) [] []</code>	Ein Zeiger auf ein Array von einem Array von ints
<code>int (*paf) [] ()</code>	Ein Zeiger auf ein Array von Funktionen mit ints als Funktionswert (ILLEGAL)
<code>int *(*pfp) ()</code>	Ein Zeiger auf eine Funktion, mit einem Zeiger auf int als Funktionswert
<code>int (*pfa) () []</code>	Ein Zeiger auf eine Funktion mit einem Array von ints als Funktionswert (ILLEGAL)
<code>int (*pff) () ()</code>	Ein Zeiger auf eine Funktion mit einer Funktion als Funktionswert die selbst ints als Funktionswert liefert (ILLEGAL)
<code>int **app []</code>	Ein Array von Zeigern auf Zeiger auf ints
<code>int (*apa []) []</code>	Ein Array von Zeigern auf Arrays von ints
<code>int (*apf []) ()</code>	Ein Array von Zeigern auf Funktionen mit ints als Funktionswert
<code>int *aap [] []</code>	Ein Array von Arrays von Zeigern auf ints
<code>int aaa [] [] []</code>	Ein Array von Arrays von Arrays von ints
<code>int aaf [] [] ()</code>	Ein Array von Arrays von Funktionen mit ints als Funktionswert (ILLEGAL)

Beispiele komplexer Deklarationen (3/3)

<code>int *afp [] ()</code>	Ein Array von Funktionen mit Zeigern auf ints als Funktionswert
<code>int afa [] () []</code>	Ein Array von Funktionen mit einem Array von ints als Funktionswert (ILLEGAL)
<code>int aff [] () ()</code>	Ein Array von Funktionen mit einer Funktion als Funktionswert die selbst ints als Funktionswert liefert (ILLEGAL)
<code>int **fpp ()</code>	Eine Funktion, mit einem Zeiger auf einen Zeiger auf ints als Funktionswert
<code>int (*fpa ()) []</code>	Eine Funktion, mit einem Zeiger auf ein Array von ints als Funktionswert
<code>int (*fpf ()) ()</code>	Eine Funktion, mit einem Zeiger auf eine Funktion mit int als Funktionswert
<code>int *fap () []</code>	Eine Funktion mit einem Array von Zeigern auf ints als Funktionswert (ILLEGAL)
<code>int faa () [] []</code>	Eine Funktion mit einem Array von Arrays von ints als Funktionswert (ILLEGAL)
<code>int faf () [] ()</code>	Eine Funktion mit einem Array von Funktionen als Funktionswert, die selbst ints als Funktionswerte liefern (ILLEGAL)
<code>int *ffp () ()</code>	Eine Funktion mit einer Funktion als Funktionswert, die selbst einen Zeiger auf int als Funktionswerte liefert (ILLEGAL)

12. Rekursion

12.1. Funktionsweise und Arten der Rekursion

12.2. Realisierung und klassische Beispiele

12.3. Nicht-lineare Rekursion

Rekursion

Was genau ist Rekursion?

- Es ist eine Technik, bei der eine Aufgabe *A* dadurch gelöst wird, daß man eine andere Aufgabe *A'* löst.
- Diese Aufgabe *A'* ist *von genau derselben Art* wie die Aufgabe *A*!
Lösung von *A*:
Löse Aufgabe *A'*, was von derselben Art wie Aufgabe *A* ist.
- Obwohl *A'* von derselben Art wie *A* ist, ist die Aufgabe *A'* doch in einer gewissen Art *kleiner*!

Beispiel

Binäres Suchen in einem Wörterbuch:

```
Suche(Wörterbuch, Wort)
  IF Wörterbuch besteht aus einer Seite THEN
    suche das Wort auf dieser Seite
  ELSE
    BEGIN
      Öffne das Wörterbuch in der Mitte
      Stelle fest, in welcher Hälfte das gesuchte Wort liegt
      IF Wort liegt in der ersten Hälfte THEN
        Suche(erste Wörterbuchhälfte, Wort)
      ELSE
        Suche(zweite Wörterbuchhälfte, Wort)
    END
```

Strategie der Rekursion: „Teile und erobere“.

Jeder Schritt wird etwas einfacher, bis man bei dem *degenerierten Fall* (*Terminierungsbedingung*) ankommt, der die sofortige Lösung bietet.

Rekursion

Vier grundsätzliche Fragen, um eine rekursive Lösung zu konstruieren:

1. Wie kann man das Problem als ein kleineres Teilproblem derselben Art umdefinieren?
2. Wie verringert jeder rekursive Aufruf die Größe des Problems?
3. Welcher Fall des Problems kann als degenerierter Fall dienen?
4. Wird dieser degenerierte Fall auch erreicht?

Standard-Beispiel: Fakultät

- Berechnung der mathematischen Funktion **Fakultät**.
- Iterative Lösung wäre allerdings in diesem Fall effizienter!
- Die Funktion läßt sich folgendermaßen darstellen:

$$\begin{array}{ll} n! := n * (n-1)! & \text{für } n > 0 \\ 0! := 1 & \text{degenerierter Fall} \end{array}$$

- Danach wird zur Berechnung von $n!$ die Berechnung von $(n-1)!$ benötigt.
- Weiterhin gilt nach obiger Definition:
$$(n-1)! = (n-1) * (n-2)!$$
- Also wird die Berechnung von $(n-2)!$ benötigt, usw.
- Also ruft die Funktion *Fakultät* immer wieder sich selbst auf.
- Der Parameter n wird in jedem Schritt um eins reduziert.
- Schließlich landet man bei $0!$. Dann ist die Rekursion zu Ende.

Standard-Beispiel: Fakultät

Fakultät(4)

$$4! = 4 \cdot 3!$$

$$3! = 3 \cdot 2!$$

$$2! = 2 \cdot 1!$$

$$1! = 1 \cdot 0!$$

$$0! = 1$$

- Damit haben wir den degenerierten Fall als Terminierungsbedingung der Rekursion erreicht.
- Die Rekursion ist zu Ende. Aber wo ist die Lösung?
- Aus unseren bisher erarbeiteten Erkenntnissen können wir nun - von hinten beginnend - die Lösungen jedes Schrittes in den vorhergehenden einsetzen, usw.
- Schließlich gelangen wir zum ersten Schritt, der die ganze Rekursion ausgelöst hat:

$$\text{Da } 0!=1, \text{ ist } 1! = 1 \cdot 1 = 1$$

$$\text{Da } 1!=1, \text{ ist } 2! = 2 \cdot 1 = 2$$

$$\text{Da } 2!=2, \text{ ist } 3! = 3 \cdot 2 = 6$$

$$\text{Da } 3!=6, \text{ ist } 4! = 4 \cdot 6 = 24$$

Funktionsweise der Rekursion

- Soll ein rekursiver Algorithmus endlich sein, so muß es eine Terminierungsbedingung geben (s.o. bei der Fakultät).
- Bei Nichterfüllung wird ein Parameter gezielt verändert.
- Dabei wird das Problem „kleiner,,.
- Dies geht solange, bis einmal die Terminierungsbedingung (degenerierter Fall) erfüllt ist.
- Jedesmal wird eine neue Rekursionsebene eröffnet, die jedoch - solange die Terminierungsbedingung noch nicht erfüllt ist - nicht sofort gelöst werden kann: zunächst wird nur die nächste Ebene geöffnet.
- Am Ende wird in umgekehrter Reihenfolge eine Ebene nach der anderen abgearbeitet.

Funktionsweise der Rekursion

Bei der Abarbeitung einer rekursiven Funktion $f(x)$ mit dem Parameter x können also zwei Fälle auftreten:

1. x erfüllt die Terminierungsbedingung ($0!=1$ in obigem Beispiel). Dann erhält $f(x)$ einen bestimmten Wert $bed(x)$.
2. x erfüllt die Terminierungsbedingung nicht. Um diese zu erreichen muß reduziert werden. Dies geschieht nach dem Algorithmus $red(x)$. Der nächste Selbstaufruf lautet dann $f(red(x))$.

Schematisch:

$$f(x) = \begin{cases} bed(x), & \text{falls Bedingung erfüllt} \\ f(red(x)), & \text{sonst} \end{cases}$$

Die Durchführbarkeit der Reduktion ist gegeben, wenn die Kette der einzelnen Reduktionen $red(x)$ nach endlichen Schritten abbricht.

Die Komplexität der Reduktion hängt von der Komplexität der Funktion $red(x)$ ab, aber auch - evtl. noch stärker - ob in einem Schritt nur ein Selbstaufruf erfolgt oder vielleicht noch mehrere.

Rekursionsarten

Man unterscheidet:

- *Lineare Rekursion* wie beim Beispiel *Fakultät*. Es erfolgt in jedem Schritt nur ein Selbstaufruf.
- *Nichtlineare Rekursionen* entstehen, wenn die Rekursionen in jedem Schritt *vervielfältigt* werden.
- *Binäre Rekursion*: Hier enthält jeder Schritt *zwei* Selbstaufrufe.
- *Indirekte Rekursionen* entstehen durch – oft ungewollte – gegenseitige Aufrufe von Unterprogrammen:

```
void a(void)      void b(void)
{
    ...
    b();
}
{
    ...
    a();
}
```

Dies kann auch über mehrere Zwischenstationen geschehen. Es führt in der Regel zu einer Endlos-Schleife und sollte daher vermieden werden!

Realisierung und klassische Beispiele der Rekursion

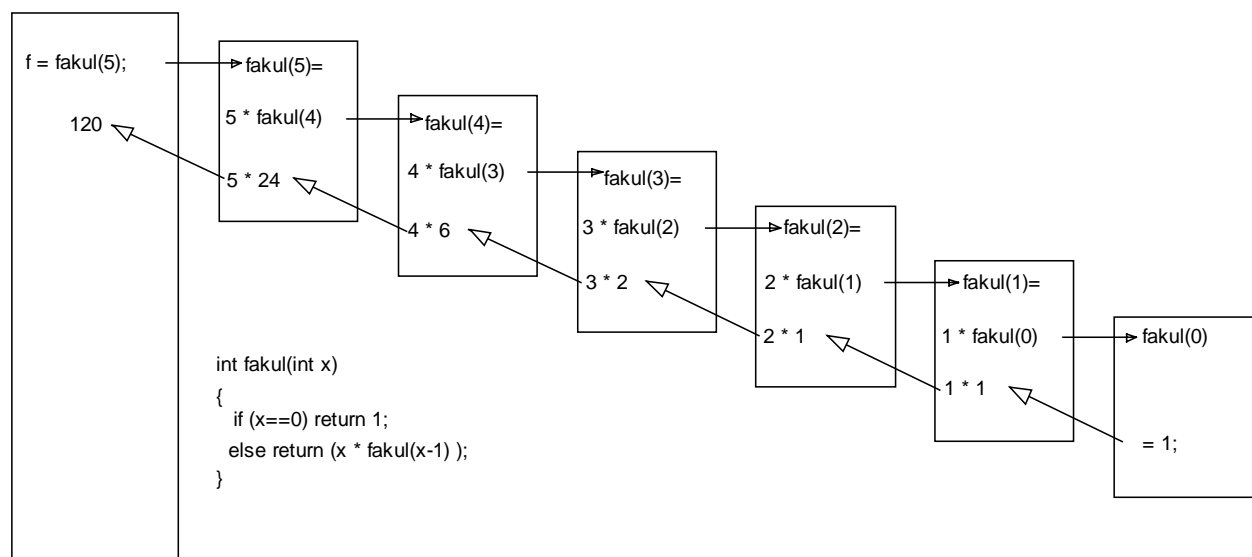
Fakultät

```
int fakultaet(int x)
{
    if (x==0)
        return 1;
    else
        return ( x * fakultaet(x-1) );
}
```

Bei jedem Aufruf von *fakultaet* wird ein weiterer Speicherplatz für den Werteparameter angelegt (übrigens auch für alle evtl. benötigten lokalen Parameter!).

Berechnet man z.B. $5!$, so existieren zum Schluß 6 verschachtelte Werteparameter (und lokale Umgebungen der Funktion).

Graphische Darstellung:



Realisierung und klassische Beispiele der Rekursion

Anwendung eines *Stacks (Stapel)*, im Deutschen *Kellerspeicher (LIFO-Prinzip)*.

Je nachdem, wie tief der Keller belegt ist, spricht man bei Rekursionen auch von *Rekursionstiefe*.

Ein anderes einfaches Beispiel

```
int spiegel(void);
{
    char x;

    scanf("%c", x);
    if (x != ' ') spiegel;
    printf("%c", x);
}
```

Was macht dieses Programm?

Nicht-lineare Rekursion

Beispiel einer binären Rekursion

Berechnung der *Fibonacci-Zahlen* nach folgender Definition:

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2) & \text{für } n > 1 \\ 1 & \text{für } n = 1 \\ 0 & \text{für } n = 0 \end{cases}$$

$fib(n)$ gibt für eine natürliche Zahl n die Summe der beiden vorangehenden Fibonacci-Zahlen der natürlichen Zahlen $n-2$ und $n-1$ an.

Weiteres klassisches Beispiel sind die *Türme von Hanoi*

13. Dateien und Dateiverarbeitung

- 13.1. Streams
- 13.2. Datenpufferung
- 13.3. Die <stdio.h> Header Datei
- 13.4. Fehlerbehandlung
- 13.5. Öffnen und Schließen von Dateien
- 13.6. Lesen und Schreiben von Daten
- 13.7. Wahl der Ein-/Ausgabe Methode
- 13.8. Ungepufferte Ein-/Ausgabe
- 13.9. Wahlfreier Zugriff
- 13.10. Beispielprogramme
- 13.11. Zusammenfassung der I/O-Funktionen

Dateien und Dateiverarbeitung

- Alle bisher kennengelernten Datentypen sind interner Natur, existieren nur während der Laufzeit des Programms. Ist das Programm zu Ende, sind die Daten „weg“.
- Will man die Daten auch über längere Zeit zu Verfügung haben, muß man sie irgendwie permanent speichern. Dies kann auf Disketten, auf der Festplatte oder auf anderen Datenträgern (Lochkarte, Lochstreifen, Magnetband, CD-ROM, etc.) geschehen. Zuvor müssen die Daten jedoch in eine geeignete Struktur gebracht werden und zu einer logischen Einheit zusammengefaßt werden. Eine solche logische Einheit nennt man *Datei*, im Englischen *File* (d.h. Ordner oder Karteikasten), bekannt z.B. von DOS.
- Im Gegensatz zu dem bisher kennengelernten Datentyp *Array* ist es bei einer Datei *nicht* notwendig, von *vornherein die Anzahl* der Komponenten festzulegen. Man kann in eine Datei so viele Sätze schreiben, wie das Speichermedium (oder der Systemadministrator) Platz zur Verfügung stellt.

Die Datei ist die allgemeinste und mächtigste Ablageform von Daten auf einem Speichermedium. Durch geeignete interne Struktur und Verknüpfung mehrerer Dateien erhält man Datenbanken.

Streams

- Ziel: Überwindung der Abhängigkeiten vom Betriebssystem.
- Die ANSI-konforme C-Laufzeitbibliothek: ca. 40 Funktionen zur Ein- und Ausgabe zur Verfügung. Sie verwenden alle den sog. *buffered I/O*, verwenden also einen internen Zwischenpuffer zur Ein- und Ausgabe.
- Kein Unterschied bezüglich des möglichen Ein-/Ausgabegerätes gemacht. Jegliche Ein-/Ausgabe verläuft über sog. *streams* (etwa: Textstrom), die mit dem Gerät oder der Datei verknüpft sind.

Streams

Ein *stream* ist eine geordnete Sequenz von Bytes, quasi ein 1-dimensionales Array von Charactern. Lesen und Schreiben bedeutet Lesen und Schreiben von bzw. in den *stream*.

Bevor eine Ein-/Ausgabe-Operation durchgeführt werden kann, muß ein *stream* mit der Datei oder dem Gerät verknüpft werden. Dazu muß ein Zeiger auf eine Struktur vom Datentyp FILE deklariert werden. Diese Struktur ist in *stdio.h* definiert. Sie enthält mehrere Komponenten, z.B. für den Dateinamen oder die Art des Zugriffes sowie einen Zeiger auf das nächste Zeichen im Datenstrom (*file position indicator*).

Der Zeiger auf diese Datenstruktur, der sog. *file pointer*, ist der einzige Angriffspunkt für jegliche Ein-/Ausgabe. Er wird durch die Funktion *fopen()* mit einem Wert belegt. Ein Programm kann mehrere *streams* zur gleichen Zeit offen haben.

Der *file position indicator* gibt an, von welcher Stelle in dem *stream* das nächste Zeichen gelesen oder an welche Stelle das nächste Zeichen geschrieben wird.

Standard Streams

Drei *streams* sind standardmäßig in jedem C-Programm geöffnet: *stdin*, *stdout* und *stderr*. Normalerweise sind sie mit Tastatur bzw. Bildschirm verbunden.

printf() schreibt nach *stdout*, *scanf()* liest von *stdin*. Mit den Funktionen *fprintf()* und *fscanf()* kann man mit Dateien oder externen Geräten kommunizieren.

Streams

Text- und Binärformat

- Auf Daten kann in zwei unterschiedlichen Formaten zugegriffen werden, als Text oder binär (*text* oder *binary*).
- Ein Textstrom besteht aus einer Kette von Zeichen aufgeteilt in *Zeilen*. Das Ende einer Zeile wird durch einen *newline* Character dargestellt.
- Physikalisch können die Daten anders gespeichert sein. Für den Benutzer stellt sich jedoch immer das Zeilenende in Form des *newline* Zeichens dar.
- Portabilität nicht immer hundertprozentig.
- Die oben erwähnten drei Standard *streams* werden als Textströme geöffnet.
- Im Binärformat werden die Daten so geschrieben, wie sie im Rechner (z.B. in einer Datenstruktur) gespeichert sind.

Datenpufferung (engl.: *buffering*)

- Reduktion der physikalischen Zugriffe auf das Ein-/Ausgabegerät. Wird von allen Betriebssystemen durchgeführt. Zugriff auf 512 oder 1024 Bytes große Blöcke.
- Die C-Laufzeitbibliothek schiebt eine weitere Pufferungsebene dazwischen. Es gibt zwei Ausprägungen: *Zeilenpufferung* und *Blockpufferung*.
- Im ersten Fall dient der *newline* Character als Pufferungsgrenze, im zweiten Fall wird immer mit einer festen Blockgröße gearbeitet.
- Man kann mit Hilfe der Funktion *fflush()* erzwingen, dass der Ausgabepuffer zu dem Gerät hin geleert wird.
- Durch setzen bestimmter Parameter in der Laufzeitbibliothek kann die Pufferung weiter beeinflusst werden. Z.B. ist es möglich durch Setzen der Puffergröße auf 0 die Pufferung ganz auszuschalten (*unbuffered I/O*).

Die <stdio.h> Header Datei

Hier liegen alle wesentlichen I/O-bezogenen Definitionen:

- Die FILE Struktur,
- nützliche Makrokonstanten, wie *stdin*, *stdout* und *stderr*,
- die Definition EOF, die von vielen I/O-Funktionen zurückgegeben wird, wenn das Ende der Datei (*End-of-File*) erreicht ist.

Früher war hier auch der Null-Pointer Wert NULL definiert. Nach ANSI ist dieser Wert jetzt in *stddef.h* definiert.

Fehlerbehandlung bei Ein-/Ausgabe

Jede I/O-Funktion liefert im Fehlerfall einen besonderen Wert zurück (Siehe Dokumentation).

Zwei Funktionen erlauben es, die End-of-File- und Error-Flags eines *streams* abzufragen:

- *feof()* und
- *ferror()*.

Bei Initialisieren eines *streams* werden diese Flags zurückgesetzt, später aber nie wieder automatisch. Dazu dient die Funktion

- *clearerr()*.

Ein Beispiel

```
/* Return stream status flags.
 * Two flags are possible: EOF and ERROR
 */

#include <stdio.h>
#define EOF_FLAG 1
#define ERR_FLAG 2

char stream_stat( FILE *fp )
{
    char stat = 0;
    if (ferror( fp ))
        stat |= ERR_FLAG; /* Bitwise inclusive OR */
    if (feof( fp ))
        stat |= EOF_FLAG; /* Bitwise inclusive OR */
    clearerr(fp);
    return stat;
}
```

Als schlimmes Rudiment aus der Unix-Welt gibt es die globale(!) Integer-Variable *errno*, definiert in *errno.h*, die von einigen (wenigen) I/O-Funktionen benutzt wird (mehr von mathematischen Funktionen); s. Dokumentation.

Öffnen von Dateien

Öffnen mit der Funktion *fopen()*.

Zwei Parameter:

- Dateiname
- Zugriffsmodus

Es gibt zwei Sätze von Zugriffsmodi,

- einen für Textströme und
- einen für binäre Ströme,

wobei die Binären Modi sich lediglich durch ein nachgestelltes *b* unterscheiden.

fopen() Text-Modi

Kürzel	Bedeutung
"r"	Öffne existierende Textdatei zum Lesen. Lesen beginnt am Anfang der Datei.
"w"	Erzeuge eine neue Textdatei zum Schreiben. Existiert die Datei bereits, wird sie auf Länge 0 abgeschnitten. Der <i>file position indicator</i> steht zunächst am Anfang der Datei.
"a"	Öffne existierende Textdatei im <i>append</i> Modus. Es kann nur ab dem Ende der Datei weiterschreiben.
"r+"	Öffne existierende Textdatei zum Lesen und Schreiben. Der <i>file position indicator</i> steht zunächst am Anfang der Datei.
"w+"	Erzeuge eine neue Textdatei zum Schreiben und Lesen. Existiert die Datei bereits, wird sie auf Länge 0 abgeschnitten. Der <i>file position indicator</i> steht zunächst am Anfang der Datei.
"a+"	Öffne existierende Textdatei oder erzeuge eine neue im <i>append</i> Modus. Es kann von überall gelesen werden, es kann nur ab dem Ende der Datei geschrieben werden.

Öffnen von Dateien

Auswirkungen der verschiedenen Modi

	r	w	a	r+	w+	a+
Datei muß existieren	*			*		
Alte Datei wird auf Länge 0 beschnitten		*			*	
Strom kann gelesen werden	*			*	*	*
Strom kann geschrieben werden		*	*	*	*	*
Strom kann nur am Ende geschrieben werden			*			*

fopen() gibt als Funktionswert den *file pointer* zurück.

Beispiel

```
#include <stddef.h>
#include <stdio.h>

/* ---- Open file named "test" with read access ---- */

FILE *open_test() /* Returns a pointer to opened FILE */
{
    FILE *fp;

    fp = fopen ("test", "r");
    if (fp == NULL)
        fprintf(stderr, "Error opening file test\n");
    return fp;
}
```

Das Öffnen und Testen etwas C-typisch knapper:

```
if ((fp = fopen ("test", "r")) == NULL)
    fprintf(stderr, "Error opening file test\n");
```

Öffnen von Dateien

Allgemeinere Funktion

Öffnen einer beliebigen Datei in einem beliebigen Modus:

```
#include <stddef.h>
#include <stdio.h>

FILE *open_file(char *file_name, char *access_mode)
{
    FILE *fp;

    if ((fp = fopen(file_name, access_mode)) == NULL)
        fprintf(stderr, "Error opening file %s with access"
                    "mode %s\n", file_name, access_mode );
    return fp;
}
```

Achtung:

Die doppelte Klammerung um ((fp = fopen(...)) ist notwendig, da der „==“-Operator einen höheren Vorrang als = hat!

Zugehörige *main*-Funktion

```
#include <stddef.h>
#include <stdio.h>

int main(void)
{
    extern FILE *open_file(char *file_name, char *access_mode)

    if (open_file("test", "r") == NULL)
        exit(1);
    ...
}
```

Schließen einer Datei

```
fclose(fp);
```

Man sollte dies auch möglichst immer tun.

Lesen und Schreiben von Daten

Lesen und Schreiben ist in 3 verschiedenen Granularitäten möglich:

- Zeichen,
- Zeile und
- Block.

Die folgenden Beispiele realisieren eine Funktion, die eine Datei in eine andere kopiert, auf diese verschiedenen Weisen.

Zeichenweises I/O

Vier Funktionen

<code>getc()</code>	Makro, der ein Zeichen aus dem <i>stream</i> liest
<code>fgetc()</code>	Dasselbe als Funktion
<code>putc()</code>	Makro, der ein Zeichen in den <i>stream</i> schreibt
<code>fputc()</code>	Dasselbe als Funktion

Vorsicht mit Makros! Aber oft schneller.

Beispiel

```
#include <stddef.h>
#include <stdio.h>
#define FAIL 0
#define SUCCESS 1

int copyfile(char *infile, char *outfile)
{
    FILE *fp1, *fp2;

    if ((fp1 = fopen( infile, "rb" )) == NULL)
        return FAIL;
    if ((fp2=fopen( outfile, "wb" )) == NULL)
    {
        fclose( fp1 );
        return FAIL;
    }

    while (!feof( fp1 ))
        putc( getc( fp1 ), fp2 );
    fclose( fp1 );
    fclose( fp2 );
    return SUCCESS;
}
```

Zeilenweises I/O

Zwei Funktionen: *fgets()* und *fputs()*.

Der Prototyp von *fgets()*:

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

Mit

<i>s</i>	Zeiger auf das erste Element eines Arrays, in das die Zeile geschrieben wird
<i>n</i>	Maximale Anzahl zu lesender Zeichen
<i>stream</i>	Input stream Zeiger

Liest so viele Zeichen bis

- ein *newline* Zeichen gefunden,
- EOF oder
- die maximale Anzahl der zu lesenden Zeichen erreicht

ist.

fgets() speichert auch den *newline* Character in dem Ziel-Array.

fgets() fügt an das Ende der Zeile ein Null-Character an. Das Array sollte also um ein Element größer sein, als die spezifizierte maximale Anzahl.

fgets() gibt NULL zurück, wenn EOF gefunden wurde, sonst denselben Zeiger wie das erste Argument.

Zeilenweises I/O

Beispiel

```
#include <stddef.h>
#include <stdio.h>

#define FAIL 0
#define SUCCESS 1
#define LINESIZE 100

int copyfile( char *infile, char *outfile )
{
    FILE *fp1, *fp2;
    char line[LINESIZE];

    if ((fp1 = fopen( infile, "r" )) == NULL)
        return FAIL;
    if ((fp2 = fopen( outfile, "w" )) == NULL)
    {
        fclose( fp1 );
        return FAIL;
    }
    while (fgets( line, LINESIZE-1, fp1 ) != NULL)
        fputs( line, fp2 );
    fclose( fp1 );
    fclose( fp2 );
    return SUCCESS;
}
```

Blockweises I/O

Zwei Funktionen: *fread()* und *fwrite()*.

Der Prototyp von *fread()*:

```
size_t fread(void *ptr, size_t size, size_t nmemb,  
              FILE stream);
```

Mit

<i>ptr</i>	Zeiger auf ein Array, das den Block aufnehmen soll
<i>size</i>	Größe jedes Elements des Arrays in Bytes
<i>nmemb</i>	Anzahl von zu lesenden Elementen
<i>stream</i>	Input stream Zeiger

fread() gibt die tatsächlich gelesene Anzahl von Elementen zurück.

Dies sollte derselbe Wert wie der des 3. Parameters sein, falls nicht EOF erreicht wurde oder ein Fehler auftrat.

fwrite() besitzt dieselbe Parameterversorgung, aber schreibt natürlich in den *stream*.

Blockweises I/O

Beispiel

```
#include <stddef.h>
#include <stdio.h>

#define FAIL 0
#define SUCCESS 1
#define BLOCKSIZE 512

typedef char DATA;

int copyfile(char *infile, char *outfile)
{
    FILE *fp1, *fp2;
    DATA block[BLOCKSIZE];
    int num_read;

    if ((fp1 = fopen( infile, "rb" )) == NULL)
    {
        printf( "Error opening file %s for input.\n",
                infile );
        return FAIL;
    }

    if ((fp2 = fopen( outfile, "wb" )) == NULL)
    {
        printf( "Error opening file %s for output.\n",
                outfile );
        fclose( fp1 );
        return FAIL;
    }

    while ((num_read = fread( block, sizeof(DATA),
                             BLOCKSIZE, fp1 )) > 0)
        fwrite( block, sizeof(DATA), num_read, fp2 );
    fclose( fp1 );
    fclose( fp2 );

    if (ferror( fp1 ))
    {
        printf( "Error reading file %s\n", infile );
        return FAIL;
    }
    return SUCCESS;
}
```

Keine saubere Überprüfung, ob weniger als die angeforderten Zeichen geliefert wurden. So ist jedoch der Code kompakter.

Wahl der Ein-/Ausgabe Methode

Von der Geschwindigkeit her sind die Makros *putc()* und *getc()* normalerweise am schnellsten.

Oft besitzen Betriebssysteme jedoch direkte Schnittstellen für Block I/O, die sehr effizient sind. Diese sind jedoch üblicherweise nicht in die C-Laufzeitbibliothek integriert. Ggf. muß man sie direkt aufrufen.

Ein anderer Aspekt ist die Einfachheit und Lesbarkeit des Quelltextes.

Will man z.B. eine zeilenweise Auswertung einer Datei durchführen, dann sind die (langsameren) Funktionen *fgets()* und *fputs()* die bessere Wahl:

```
#include <stdio.h>
#include <stddef.h>
#define MAX_LINE_SIZE 120

/* ----- Zähle die Zeilen einer Datei ----- */

int lines_in_file(FILE *fp)
{
    char buf[MAX_LINE_SIZE];
    int line_num = 0;

    rewind(fp); /* Moves the file position indicator
                 * to the beginning of the file.
                 */
    while (fgets(buf, MAX_LINE_SIZE, fp) != NULL)
        line_num++;

    return line_num;
}
```

Diese Funktion mit zeichen- oder blockorientiertem I/O zu programmieren würde eine wesentlich schlechter lesbare Funktion erzeugen.

Das dritte Kriterium ist Portabilität. Textdateien sollten im Textmodus geöffnet werden, Dateien mit binären Daten im binären Modus.

Wahlfreier Zugriff

„*Random Access*„

Für Anwendungen, wo von einer bestimmten Stelle der Datei gelesen werden soll, gibt es die Funktionen *fseek()* und *ftell()*.

fseek() bewegt den *file position indicator* zu einem bestimmten Zeichen im Strom:

```
int fseek(FILE *stream, long int offset, int whence);
```

Die Argumente sind:

<i>stream</i>	der File Pointer
<i>offset</i>	ein positiver oder negativer Offset in Characters
<i>whence</i>	von wo aus wird der Offset gerechnet

Mögliche Werte für *whence* sind

<i>SEEK_SET</i>	vom Anfang der Datei
<i>SEEK_CUR</i>	von der aktuellen Stelle des <i>file position indicators</i>
<i>SEEK_END</i>	vom Ende der Datei (EOF Position)

Wahlfreier Zugriff

Der Aufruf

```
status = fseek(fp, 10, SEEK_SET);
```

positioniert den *file position indicator* zum Zeichen 10 im Strom. Auch hier wird ab 0 gezählt. Es wird 10 Zeichen ab dem Zeichen 0 vorwärts positioniert, also auf das 11. Zeichen.

Wenn alles o.k. ist, gibt *fseek()* eine 0 zurück!

Für binäre Ströme kann *offset* ein positiver oder negativer Wert sein, der nicht aus dem Bereich der Datei herausführt.

Für Textströme muß *whence* den Wert *SEEK_SET* haben und *offset* 0 sein oder ein Wert besitzen, der von *ftell()* zurückgegeben wurde.

ftell() hat nur den File Pointer als Argument und gibt den aktuellen Wert des *file position indicators* zurück. Hiermit kann man sich eine Position in der Datei merken, zu der man später wieder hinpositionieren will:

```
cur_pos = ftell(fp);  
if (search(string) == FAIL)  
    fseek(fp, cur_pos, SEEK_SET);
```


Schreiben und Lesen einer binären Datei (1/2)

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
/*-----+
 * Dieses Programm liest eine Zahl von Datensätzen von der
 * Tastatur-Eingabe und schreibt sie auf eine Datei im
 * Laufwerk a:. Nach beendeter Eingabe werden zuerst die
 * männlichen Einträge, dann die weiblichen auf dem Bild-
 * schirm gelistet.
 *-----*/
{
    typedef struct {
        char name[20];
        int geschlecht;
    } PERSON;

    FILE *datei;
    PERSON satz;
    char eingabe[5], antwort[5];

    if ((datei = fopen("a:\\person.dat", "wb")) == NULL)
    {
        printf("Fehler beim Öffnen der Datei");
        exit(1);
    }

    do
    {
        printf("Name: ");
        scanf("%s", satz.name);
        printf("Männlich oder weiblich (M/W)? ");
        scanf("%s", eingabe);

        switch (eingabe[0]) /* Wandle Eingabe in Zieltyp um */
        {
            case 'M':
            case 'm': satz.geschlecht = 1;
                     break;
            case 'W':
            case 'w': satz.geschlecht = 2;
                     break;
        }

        if (fwrite(&satz, sizeof(satz), 1, datei) != 1)
        {
            printf("Fehler beim Schreiben\n");
            exit(2);
        }
        printf("\nWeiteren Satz eingeben (J/N)? ");
        scanf("%s", antwort);
    }
```

```
} while ((antwort[0] == 'J') | (antwort[0] == 'j'));  
fclose(datei);
```

Schreiben und Lesen einer binären Datei (2/2)

```
/* ----- Ausgabe 'männlich' ----- */
printf("\nMännlich:\n");
if ((datei = fopen("a:\\person.dat", "rb")) == NULL)
{
    printf("Fehler beim 1. Öffnen der Datei zum Lesen");
    exit(3);
}
while (fread(&satz, sizeof(satz), 1, datei) == 1)
{
    if (satz.geschlecht == 1)
        printf("%s\n", satz.name);
}
fclose(datei);

{ ----- Ausgabe 'weiblich' ----- }
printf("\nWeiblich:\n");
if ((datei = fopen("a:\\person.dat", "rb")) == NULL)
{
    printf("Fehler beim 2. Öffnen der Datei zum Lesen");
    exit(4);
}
while (fread(&satz, sizeof(satz), 1, datei) == 1)
{
    if (satz.geschlecht == 2)
        printf("%s\n", satz.name);
}
fclose(datei);
exit(0);
}
```

Zeichenweises Schreiben einer neuen Textdatei

```
...
FILE    *datei;
char    c;

if ((datei = fopen("...", "w")) == NULL)
{
    printf("Fehler beim Anlegen der Datei");
    exit(1);
}

while (!fertig)
{
    while (!zeile_fertig)
    {
        lies_ein_zeichen(c);
        putc(c, datei);
    }
    putc('\n', datei); /* End-of-Line */
    ...
}
```

Zeichenweises Lesen einer Textdatei

```
...
FILE    *datei;
char    c;

if ((datei = fopen("...", "r")) == NULL)
{
    printf("Fehler beim Öffnen der Datei");
    exit(1);
}

while (!feof(datei))
{
    while ( (c=getc(datei)) != '\n')
        verarbeite_ein_zeichen(c);

    verarbeite_zeilenende();
}
```

I/O mit *stdin* oder *stdout*

<i>Funktion</i>	<i>Beschreibung</i>
<code>getchar()</code>	Liest nächstes Zeichen von <i>stdin</i> . Identisch mit <i>getc(stdin)</i>
<code>gets()</code>	Liest Zeichen von <i>stdin</i> bis <i>newline</i> oder <i>eof</i> angetroffen
<code>printf()</code>	Gibt ein oder mehrere Werte entsprechend Formatierungsangaben des Anwenders nach <i>stdout</i> aus
<code>putchar()</code>	Gibt ein Zeichen nach <i>stdout</i> aus. Identisch mit <i>putc(stdout)</i>
<code>puts()</code>	Gibt einen String von Zeichen nach <i>stdout</i> aus. Fügt ein <i>newline</i> ans Ende
<code>scanf()</code>	Liest ein oder mehrere Werte von <i>stdin</i> , wobei jeder gemäß den Formatierungsregeln interpretiert wird

Fehlerbehandlungsroutinen

<i>Funktion</i>	<i>Beschreibung</i>
<code>clearerr()</code>	Fehlerflag (<i>errno</i>) und EOF-Flag des entsprechenden <i>streams</i> werden zurückgesetzt
<code>feof()</code>	Prüft, ob während der vorigen I/O-Operation EOF gefunden wurde
<code>ferror()</code>	Gibt einen Fehlerwert zurück (Wert von <i>errno</i>), falls zuvor ein fehler aufgetreten ist, sonst 0

Dateimanagement Funktionen

<i>Funktion</i>	<i>Beschreibung</i>
<code>remove()</code>	Löscht eine datei
<code>rename()</code>	Benennt eine Datei um
<code>tmpfile()</code>	Erzeugt eine temporäre binäre Datei
<code>tmpname()</code>	Erzeugt einen Namen für eine temporäre Datei

Ein-/Ausgabe mit Dateien

<i>Funktion</i>	<i>Beschreibung</i>
<code>fclose()</code>	Schließt eine Datei
<code>fflush()</code>	Leert den Puffer des <i>streams</i> . Datei bleibt geöffnet
<code>fgetc()</code>	wie <code>getc()</code> , aber als Funktion statt Makro implementiert
<code>fgets()</code>	wie <code>gets()</code> , aber von beliebigem <i>stream</i> ; weiterhin kann die maximale Anzahl der zu lesenden Zeichen angegeben werden
<code>fopen()</code>	Öffnet, evtl kreieert, Datei und verknüpft sie mit einem <i>stream</i>
<code>fprintf()</code>	wie <code>printf()</code> , jedoch nach beliebigem <i>stream</i>
<code>fputc()</code>	wie <code>putc()</code> , aber als Funktion statt Makro implementiert
<code>fputs()</code>	wie <code>puts()</code> , aber nach beliebigem <i>stream</i> ; weiterhin wird kein <i>newline</i> in den <i>stream</i> geschrieben
<code>fread()</code>	Liest einen Block von Daten von einem <i>stream</i>
<code>freopen()</code>	Schließt einen <i>stream</i> und öffnet ihn mit einer neuen Datei (z.B. Umdefinition von <i>stdin</i>)
<code>fscanf()</code>	wie <code>scanf()</code> , jedoch von beliebigem <i>stream</i>
<code>fseek()</code>	Positioniere wahlfrei in Datei
<code>ftell()</code>	Liefert Wert des <i>file position indicators</i> zurück
<code>fwrite()</code>	Schreibt einen Block von Daten in einen <i>stream</i>
<code>getc()</code>	Liest ein Zeichen von einem <i>stream</i>
<code>putc()</code>	Schreibt ein Zeichen in einen <i>stream</i>
<code>ungetc()</code>	Schreibt ein Zeichen in einen <i>stream</i> zurück

14. Der C-Präprozessor

14.1. Makro-Definition und Substitution

14.2. Bedingte Compilation

14.3. Includierung von Source-Code

Der C-Präprozessor

Präprozessor:

- Ein Programm, das vor dem eigentlichen Compiler den Quellcode bearbeitet und dadurch den Sprachumfang von C erweitern kann.

Hauptaufgaben des Präprozessors:

- Verarbeitung von Makros (Definition und Substitution)
- Bedingte Compilation von Teilen eines C-Quell-Files in Abhängigkeit von bestimmten Ausdrücken
- Includierung von zusätzlichen C-Quell-Files

Präprozessor-Direktiven:

- beginnen mit # als erstem Zeichen ungleich Leerzeichen,
- enden mit dem Newline-Character,
- zur Verlängerung über mehrere Zeilen wird ein \ unmittelbar vor dem Newline benutzt.

Beispiele:

```
#include <stdio.h>
#define LONG_MACRO "This is a very long macro\
that spans two lines."
```


Makro-Definition und -Substitution

Ein Makro ist ein Name mit einem zugeordneten Text-String, dem Makro-Körper.

Der Name des Makros sollte nach den C-Konventionen nur aus Großbuchstaben und dem Underscore-Character `_` bestehen. Dies macht die Unterscheidung zwischen Variablennamen (bestehend aus Kleinbuchstaben) und Makronamen leichter.

Makro-Definition

Geschieht mittels der Präprozessor-Direktive `#define`:

```
#define BUFFLEN 512
```

Makro-Substitution

Wenn ein Makroname außerhalb seiner Definition erscheint, wird er durch den Makro-Körper ersetzt. Während der Präprozessor-Verarbeitung wird aus dem Text

```
char buf[BUFFLEN];
```

der Text

```
char buf[512];
```

Makro-Definition und -Substitution

Parametrisierung von Makros

Bei Makros können, ähnlich wie bei Funktionen, Parameter übergeben werden, z. B.

```
#define MUL_BY_TWO(a)  (2*(a))
```

Dann wird

```
j = MUL_BY_TWO(5);
```

expandiert zu

```
j = (2*(5));
```

Sind die Klammern notwendig? Man betrachte die Anweisung

```
j = MUL_BY_TWO(k-1);
```

Dies führt zu

```
j = (2*(k-1));
```

Ohne die inneren Klammern, also beim Makro-Körper $(2*a)$ ergäbe sich

```
j = (2*k-1);
```

Auch die äußeren Klammer sind notwendig wegen der Schachtelung von Makros.

Weitere Beispiele:

```
#define TO_LOWER(c)  ( (c) - ('a' - 'A'))  
#define getchar()  getc(stdin)  
#define feof(f)  ((f)->_flag & _IOEOF)  
#define PI  3.141592654
```

Makro-Definition und -Substitution

Makro versus Funktion

Makro

```
#define min(a, b) ((a) < (b)? (a) : (b))
```

Funktion

```
int min(int a, int b) {return a < b ? a : b;}  
float ...  
double ...
```

Löschen einer Makro-Definition

```
#undef MUL_BY_TWO  
#undef PI
```

Vordefinierte Makros

ANSI C kennt fünf Makro-Namen, die fest im Präprozessor eingebaut sind. Die beginnen und enden mit zwei Underscore-Charactern:

__LINE__

__FILE__

__TIME__

__DATE__

__STDC__ ... expandiert zu 1, falls der Compiler ANSI-C-konform ist.

Makro-Definition und -Substitution

String-Producer

```
#define str(s) #s
```

Das Statement

```
printf( str( This is a string ) );
```

expandiert zu

```
printf( "This is a string" );
```

Beispiel: ASSERT-Makro

```
#define ASSERT( b ) if (!b) {\nprintf( "The following condition failed: %s\\n",\n#b);\\ exit(1); }
```

```
ASSERT( array_ptr < array_start + array_size );
```

Falls der Ausdruck falsch ist, druckt das Programm die folgende Meldung und endet mit exit.

```
The following condition failed:\narray_ptr < array_start + array_size
```

Makro-Definition und -Substitution

Token Pasting

Der Operator `##` dient zum Verketteten von Strings:

```
#define FILENAME( extension ) test_ ## extension
```

Die Zeichenfolge

```
FILENAME( bak )
```

expandiert zu

```
test_bak
```

Fehlermöglichkeiten

Makro-Definition mit Semikolon beendet:

```
#define SIZE 10;
```

Benutzung von `=` in Makro-Definition:

```
#define MAX = 100
```

Leerzeichen zwischen linker Klammer und Makroname:

```
#define neg_a_plus_f (a) - (a) + f
```

Seiteneffekte bei Makro-Parametern:

```
#define min(a, b) ((a) < (b)? (a) : (b))  
...  
a = min( b++, c );
```

Mangelnde Klammerung von Parametern:

```
#define SUM(a, b) a + b
```

Bedingte Compilation

Syntaktischer Aufbau :

```
#if Conditional expression
    C-Quellcode
[ #elif Conditional expression
    C-Quellcode ]
[ #else
    C-Quellcode ]
#endif
```

Beispiele:

```
#if __STDC__
    extern int foo(char a, float b);
    extren char *goo(char *string);
#else
    extern int foo( );
    extern char *goo( );
#endif
```

```
#if DEBUG
    if (exp_debug) {
        printf("lhs = ");
        print_value(result);
        printf("rhs = ");
        print_value(&rvalue);
        printf("\n");
    }
#endif
```

Bedingte Compilation

#ifdef, #ifndef - Direktiven

Mit Hilfe von `#ifdef` bzw. `#ifndef` kann man die Existenz bzw. Nichtexistenz einer Makrodefinition testen:

```
#ifdef TEST
    printf("This is a test.\n");
#else
    printf("This is not a test.\n");
#endif
```

Includierung von Source-Code

#include - Direktive

Sie hat die Formen:

1. `#include <filename>` Suche in Standard-Verzeichnissen wie z.B. `/usr/include` bei UNIX
2. `#include "filename"` Suche im Arbeitsverzeichnis

Beispiele:

```
#include <stdio.h>
#include <sys/stat.h>
#include "myheader.h"
#include "new/myfile.c"
```