30.12.2012



ANGELINA VICINI

PROGGRU - ZUSAMMENFASSUNG

Inhaltsverzeichnis

LEXIKALISCHE KONVENTIONEN 3

Zeichen-vorrat 3
Lexikalische Analyse 3
Lexikalische Einheiten 3
Gross- und Kleinschreibung 3
Kommentare 3
Konstanten 4
Enumerations 4

DATENTYPEN UND VARIABLEN 4

Typkonzept 5
Integertypen 5
Gleitpunkt-typen 5
Variablen 5
Typ-Attribute 5

POINTER ..... FEHLER! TEXTMARKE NICHT DEFINIERT.

Pointer-definition 6
NULL-Pointer 6
Wertzuweisung 6
Referenzieren, Dereferenzieren 6
Adresse einer Variable 6
Pointer auf void 6

ARRAY 7

Eindimensionaler 7
Array 7
Arraydefinition 7
Array durchlaufen 7
Buffer overflow 7
Zeichenketten 8
Stringver-arbeitung 8

FORTGESCHRITTENE PROGRAMMIERUNG MIT POINTERN 9

Pointer auf Array 10
Vergleichen von Arrays 10
Weitere Funktionen zur Speicher-bearbeitung 11
Arraynamen 11
Pointer-arithmetik 12
Initialisierung von Arrays 12
Mehrdimen-sionale Arrays 13
Übergabe von Arrays 13
Const bei Pointern 14
Array von Pointern 15
Pointer auf Pointer 15
Pointer auf Funktionen 16

ANWEISUNG, AUSDRÜCKE, OPERATOREN 17

Stelligkeit von Operatoren 17

Postfix- und Präfix-Operatoren 17
Ausdrücke und Anweisungen 17
Nebeneffekte 18
Auswertungs-reihenfolge 18
Assozia-tivitäten 19
L- und R-Werte 19
Operatoren 19
Typum-wandlung 21

KONTROLLSTRUKTUREN 21

Sequenz 22
Selektion 22
Iteration 23
Sprung-anweisungen 24

BLÖCKE UND FUNKTIONEN 25

Block 25
Gültigkeit 25
Sichtbarkeit 25
Lebensdauer 26
Funktionen 26

STRUKTUREN UND UNIONEN 28

Struktur 28
Operationen auf Struktur-variablen 28
Zugriff auf eine Struktur-variable 29
Lage im Speicher 29
Über- und Rückgabe einer Struktur-variable 30
Initialisierung einer Struktur-variablen 30
Unionen 30

REKURSION UND ITERATION 31

Definition 31
Beispiel 31
Stack 31

SPEICHERKLASSEN 32

Adressraum 32
Programm aus mehreren Dateien 32
Speicher-klasse extern 34
Speicher-klasse static 34
Speicher-klasse bei lokalen Variablen 34

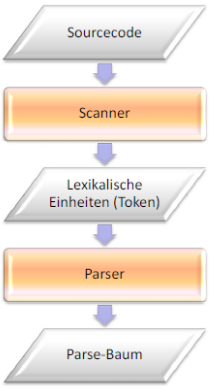
FRAGEN: 36

NICHTBEHANDELTE THEMEN: 38

Lexikalische Konventionen

Zeichen-vorrat
A – Z, a – z, 0 – 9, Leerzeichen, Tabulator, ; . () [] <> + - * / % ^ ~ & |= ! ?: ' " { } \ _
Keien Umlaute!

Lexikalische Analyse
Die lexikalische Analyse beschreibt die erste Phase des Compilielaufs die der Scanner durchführt. Am Anfang besteht ein Programm nur aus einzelnen Bytes. Der Scanner definiert Zeichngruppen indem nach den Leerzeichen gesucht wird. Diese Zeichengruppen werden als **lexikalische Einheit oder Token** beschrieben.
Im nächsten Schritt überprüft der **Parser** die Einhaltung des Syntax der Token.



Lexikalische Einheiten
Namen (Variablen, Funktionen, Marken usw.)
Reservierte Wörter, auch Schlüsselwörter (sizeof, int, if, void usw.)
Literale Konstanten (Ganzzahlige Konstanten vom Typ int usw.)
Konstante Zeichenketten (Strings)
Operatoren (+ - & [] usw.)
Satzzeichen (: [] , * = usw.)

Komplette Auflistung der lexikalischen Einheiten ab Kapitel 4.2.3 ab Seite 82

Gross- und Kleinschreibung
Gross- und Kleinschreibung: C unterscheidet Gross- und Kleinschreibung wodurch alpha und Alpha zwei unterschiedliche Namen sind. Reservierte Wörter sind immer klein geschrieben.

Kommentare
Kommentare dienen der Verständlichkeit von Programmen und werden vom Compiler nicht berücksichtigt.
Ein Komentar kann entweder so dargestellt werde: /* Ich bin ein Kommentar */ oder als Zeilenkommentar: // ich bin ein Zeilenkommentar. Der Zeilenkommentar endet am Ende der Zeile.

Konstanten

Symbolische Konstanten haben einen Namen der ihren Wert repräsentiert. Sie werden aus Grossbuchstaben und Underscores gebildet.
#define PI 3.14159

Da Symbolische Konstanten keinen Typ haben, sollen wen möglich enum's oder const's verwendet werden.

Literale Konstanten haben keinen Namen, werden aber durch ihren Wert repräsentiert.

Ganzzahlige Konstanten

```
254    035    0x3f    -34    14L    14U    14UL
dezimal  okt    hex            long   unsigned  unsigned long
```

Zeichenkonstanten

```
'c'    '\n'    '\x4a'    '\014'    '\\'    L'a'
                ASCII hex  ASCII okt          Double Byte
```

Gleitpunktkonstanten (default: double)

```
254.89    -13.0    3.45e23      4.65f      3.14159L
                   Exp-Schreibweise  float-Konstante  long double
```

Aufzählungskonstanten

Enumerations

Enumerations: Aufzählungskonstanten werden konstante ganzzahlige Werte vom Typ int zugesprochen. Das Etikett, hier test, ist optional.

```
enum test {alpha, beta, gamma};          //alpha = 0, beta = 1, gamma = 2
enum test {alpha = 5, beta = 3, gamma = 7};  //alpha = 5, beta = 3, gamma = 7
enum test {alpha = 4, beta, gamma = 3};      //alpha = 4, beta = 5, gamma = 3
```

Datentypen und Variablen

Typkonzept

Alle Variablen haben einen genau definierten, vom Programmierer festgelegten Typ (z.B. int, float). Einer Variablen können nur Werte dieses Typs zugewiesen werden.
Erfolgt eine Zuweisung von float zu int, wird der Typ umgewandelt und nur die Zahlen bis zum Dezimalpunkt übernommen.

Integertypen

- **Char:** Zeichen, (immer 8 Bit)
- **Int:** effizienteste Grösse ganzzahlige Werte (mind. 16 Bit)
- **Short oder short int:** kleine ganzzahlige Werte (mind. 16 Bit)
- **Long oder long int:** grosse ganzzahlige Werte (mind. 32 Bits)
- **Enum:** Aufzählungstyp

Signed und unsigned: bedeutet vorzeichenbehaftet bzw. nicht vorzeichenbehaftet. Gilt für ganzzahlige Datentypen. In den Grundeinstellungen sind alle ganzzahligen Datentypen ausser char signed. Bei char hängt es vom Compiler ab.

Gleitpunkt-typen

- **Float:** Gleitpunktzahl (1 Vorzeichenbit, 8 Exponentenbits, 23 Mantissenbits),
- **Double:** Gleitpunktzahl mit einer höheren Genauigkeit der Darstellung,
- **Long double:** Gleitpunktzahl mit einer noch höheren Genauigkeit der Darstellung

Übersicht über alle Standard-Datentypen mit Wertebereich: Kapitel 5.2, Seite 99

Variablen

Deklaration: Legt nur die Art und den Typ der Variable, bzw. die Schnittstelle der Funktion.
Definition: Reserviert zusätzlich Speicherplatz
Definition = Deklaration + Reservierung des Speicherplatzes

Globale Variablen: stehen allen Funktionen zur Verfügung, müssen ausserhalb einer Funktion definiert werden. Sind in allen Funktionen sichtbar. Werden zu Beginn eines Programms automatisch mit 0 initialisiert.
Lokale Variablen: stehen nur in der Funktion zur Verfügung, in der sie definiert sind. Werden nicht automatisch initialisiert.
Grundsatz: so lokal wie möglich

Variablendefinierung: datentyp name;

Typ-Attribute

Const: Einmaliges Initialisieren der Variable, ist anschliessend schreibgeschützt.
const double PI = 3.1415927;

Volatile: Der Compiler kann so die Variable aus Optimierungsgründen nicht woandershin schieben. Wird gebraucht wenn die Variablen stets an denselben Adressen stehen müssen.

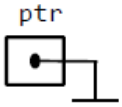
Pointer

Pointer-definition

Ein Pointer ist eine Variable, welche die Adresse einer im Speicher befindlichen Variablen oder Funktion aufnehmen kann.
Pointer können auf **Datenobjekte, Pointervariablen und Funktionen** zeigen.

NULL-Pointer

NULL ist vordefiniert (in <stddef.h>) und setzt den Pointer auf einen definierten Nullwert. Besser ist, statt NULL direkt 0 zu verwenden. Der effektive Wert des Nullpointers ist implementationsabhängig. Der Nullpointer ist einzig ein definierter Nullwert für Pointer, er zeigt aber nicht auf die Adresse 0.



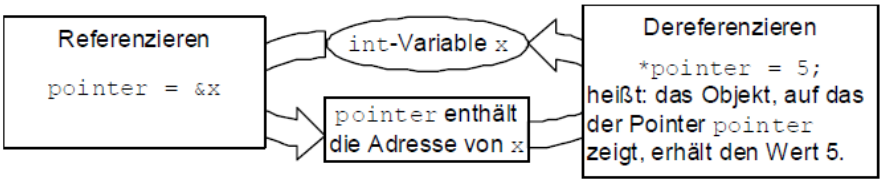

```
int * pointer = NULL;
int * pointer = 0;
```

Wertzuweisung

int alpha = 1;	Definieren einer int-Variable mit dem Wert 1
int * pointer1;	Definieren eines Pointers auf int
pointer1 = α	Zuweisung der Adresse von alpha auf den Pointer
int * pointer1 = &alpha ;	Die letzten zwei Zeilen können auch so geschrieben werden.
int * pointer2 = pointer1;	Definieren eines Pointers auf int der auf dieselbe Adresse zeigt wie pointer1.
*pointer1 = 2;	**Dereferenzierung:** Der Variable alpha wird der Wert 2 zugewiesen. Ist äquivalten zu alpha = 2;

Referenzieren, Dereferenzieren

& wird als **Adressoperator** bezeichnet. Mit ihm erhält man die **Adresse eines Objekts.**
* wird als **Dereferenzierungsoperator** bezeichnet. Mit ihm erhält man den **Inhalt des Objektes**, auf das ein Pointer zeigt.



* und & heben sich gegenseitig auf!

Adresse einer Variable

Ausgabe der Adresse einer Variable:

```
float zahl = 3.5;
printf(„Adresse von zahl: %p“, &zahl);
```

Pointer auf void

Der Pointer auf void ist ein **untypisierter** Pointer. Dieser ist zu allen anderen Pointertypen kompatibel und kann insbesondre in Zuweisungen mit typisierten Pointern gemischt werden. Ein Pointer auf void umgeht also bei einer Zuweisung die Typüberprüfung des Compilers.

Array

Eindimensionaler Array

Ein eindimensionaler Array ist eine Zusammenfassung von mehreren Variablen vom gleichen Typ. Der Compilierer erkennt ein Array an den eckigen Klammern. Die Anzahl der Elemente muss immer eine positive ganze Zahl sein. Sie kann gegeben sein durch eine Konstante, einen konstanten Ausdruck aber nicht durch eine Variabel.

Arraydefinition

```
#define groesse 3;          Konstante definieren (sollte immr so gemacht werden)

int alpha [groesse];        Arraydefinition

alpha [0] = 1;              Das 1. Element alpha [0] hat den Index 0
alpha [1] = 2;              Das 2. Element alpha [1] hat den Index 1
alpha [2] = 3;              Das 3. Element alpha [2] hat den Index 2
```

alpha[0] int	alpha[1] int	alpha[2] int
1	2	3

Man kann auch direkt den ganzen Array bestimmen (manuelle Initialisierung):
int alpha[5] = {1, 2, 3, 4, 5};

Array durchlaufen

```
#define GROESSE 5
int alpha[GROESSE];

int i;
for (i = 0; i < GROESSE; i++)
{
printf("%d \n", alpha[i]);
}
```

Buffer overflow

Beim Überschreiten des zulässigen Indexbereiches wird kein Kompilier- bzw. Laufzeitfehler erzeugt. C führt die Anweisung trotzdem durch.

```
int alpha [4];
alpha [4] = 6;          Die Speicherzelle nach dem letzten Element wird
                        beschrieben.
```

Zeichenketten

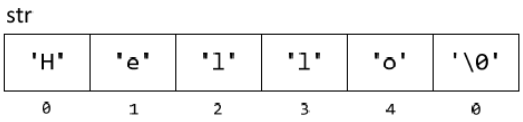
Eine Zeichenkette oder auch String ist ein **Array von Zeichen (char-Array)**. Dabei wird am Schluss ein zusätzliches Zeichen, das Zeichen **`\0`** angehängt um das Stringende zu markieren.

Direkte Initialisierung:
char str[20] = {'Z', 'e', 'i', 'c', 'h', 'e', 'n', 'k', 'e', 't', 't', 'e', '\0'};
char str[20] = "Zeichenkette";
char str[20] = {"Zeichenkette"};
char str[] = "Zeichenkette";

Ausgabe:
printf("%s\n", str);

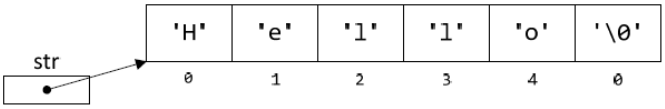
Vergleich von char-Array und Pointer auf Zeichenketten:
str entspricht konstanter Anfangsadresse des Arrays

```
char str[] = "Hello";
```



pointer kann auf andere Adresse zeigen (ist nicht konstant)
Zugriff auf "hello" könnte dadurch verloren gehen

```
char* str = "Hello";
```



Stringver-arbeitung

Manuelles Kopieren von Zeichenketten:
Es gibt einige Varianten einen String manuell zu kopieren, siehe:
Buch Kapitel 10.5, Seiten 273 / 274, (markieren!!)

Standardfunktionen zur Stringverarbeitung:
Funktionen für Strings und Speicher werden in **<string.h>** angeboten.

#include <string.h>

Funktionen, die mit **str** beginnen dienen der Stringverarbeitung, erkennen das '\0'-Zeichen. Funktionen, die mit **mem** beginnen dienen der Speicherverarbeitung, erkennen das '\0'-Zeichen nicht, deshalb muss die Bufferlänge in Byte ebenfalls als Parameter übergeben werden.

Strings kopieren strcpy():
char* strcpy(char* dest, const char* src);

Kopiert src nach dest, inklusive '\0', Rückgabewert ist dest.
dest muss auf einen Bereich zeigen, der genügend gross ist, ansonsten riskiert man einen Buffer overflow!

Strings zusammenfügen strcat():
char* strcat(char* dest, const char* src);

Hängt src an dest an, inklusive '\0'. Das ursprüngliche '\0' von dest wird überschrieben. Rückgabewert ist dest.
dest muss auf einen Bereich zeigen, der genügend gross ist, ansonsten riskiert man einen Buffer overflow!

Strings vergleichen strcmp() / strncmp():
int strcmp(const char* s1, const char* s2);
int strncmp(const char* s1, const char* s2, size_t n);

Vergleicht die beiden Strings s1 und s2 miteinander, bei strncmp() nur die ersten n char's.

Return:
< 0 : s1 ist lexikographisch kleiner als s2
== 0 : s1 und s2 sind gleich
> 0 : s2 ist lexikographisch grösser als s1

Stringlänge bestimmen strlen():
size_t strlen(const char* s);

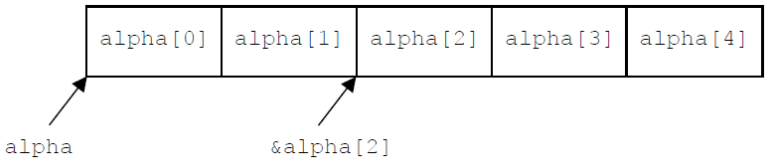
Bestimmt die Länge des Strings s, d.h. die Anzahl char's. Das Stringende-Zeichen '\0' wird dabei nicht mitgezählt!

Fortgeschrittene Programmierung mit Pointern

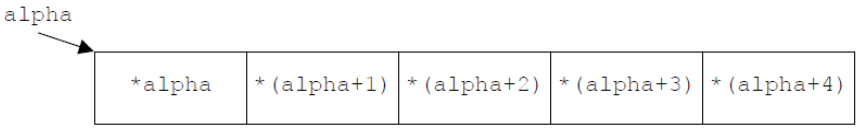
Pointer auf Array

int alpha [5];	Definieren des Arrays alpha
int * pointer;	Definition des Pointers pointer
pointer = &alpha[i-1];	Pointer zeigt auf das i-te Arrayelement

Der Name eines Arrays kann als konstanten Zeiger auf das erste Element des Arrays verwendet werden. **alpha ≙ &alpha[0]**
Deswegen wird bei Pointern auf ein Array oft kein & geschrieben!!!



Die Arraynotation ist äquivalent zu einer Pointernotation:
alpha[i] ≙ *(alpha+i)



Vergleichen von Arrays

Zwei Array können nicht so verglichen werden: **arr1 == arr2**. In diesem Fall werden die **Adressen miteinander verglichen** und die sind selten die gleichen.

Für das Vergleichen von Arrays bietet sich die Funktion **memcmp()** an:

Beispiel: **memcmp (string1, string2, sizeof (string1));**
Vergleicht die beiden Arrays string1 und string2 für die Anzahl Stellen, wie gross string1 ist.

Ausgabewerte:
< 0 wenn das erste Byte, das in beiden Puffern verschieden ist, vom ersten Array
einen kleineren Wert hat
= 0 wenn alle verglichenen Bytes gleich sind
> 0 wenn das erste Byte, das in beiden Puffern verschieden ist, vom ersten Array
einen grösseren Wert hat

Weitere Funktionen zur Speicher-bearbeitung

#include <string.h>

Speicherbereich kopieren memcpy():
void* memcpy(void* dest, const void* src, size_t n);

Kopiert n Bytes aus dem Puffer, auf den der Pointer src zeigt, in den Puffer, auf den dest zeigt. Wenn sich die Speicherbereiche überlappen, ist das Ergebnis undefiniert!
Rückgabewert ist dest.

Speicherbereich verschieben memmove():
void* memmove(void* dest, const void* src, size_t n);

Funktioniert gleich wie memcpy ausser dass bei überlappenden Speicherbereichen das korrekte Ergebnis erzielt wird.

Zeichen in Speicherbereich suchen memchr():
void* memchr(const void* s, int c, size_t n);

Durchsucht die ersten n Bytes des Puffers, auf den s zeigt, nach dem Wert c. Wird der Wert c gefunden, so wird ein Pointer auf das erste Vorkommen im Puffer zurückgegeben. Ist der Wert c in den ersten n –bytes nicht enthalten, so wird ein NULL-Pointer zurückgegeben.

Speicherbereich mit Wert belegen memset():
void* memset(const void* s, int c, size_t n);

Setzt die ersten n Bytes des Puffers, auf den der Pointer s zeigt, auf den Wert des Zeichens c. Rückgabewert ist der Pointer s.

Arraynamen

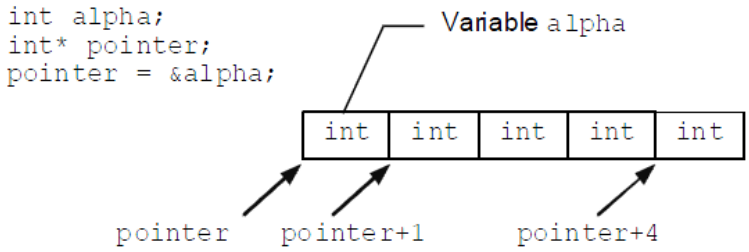
Der Arrayname ist ein nicht modifizierbarer L-Wert:
Der **Arrayname ist ein konstanter Pointer auf das erste Element des Arrays** und kann nicht verändert werden. Auf einen Arraynamen können nur die beiden Operatoren **sizeof und &** angewandt werden.
Einem Arraynamen kann kein Wert zugewiesen werden, einer Pointervariablen schon.

Pointer-arithmetik

Zuweisung: Pointer unterschiedlicher Datentypen dürfen einander nicht zugewiesen werden.
Einem Pointer eines bestimmten Typs dürfen Pointer dieses Typs oder void-Pointer zugewiesen werden.
Einem void-Pointer dürfen beliebige Pointer zugewiesen werden (nützlich aber gefährlich). Der NULL-Pointer ist gleichwertig wie der void-Pointer.

Addition und Subtraktion: Zu einem Pointer kann ein Pointer desselben Typ addiert bzw. subtrahiert werden. Ebenfalls kann zu einem Pointer eine ganze Zahl addiert bzw. subtrahiert werden.

Wenn ein Pointer um 1 erhört wird, bedeutet dies, dass er nun um die Grösse des Typs, auf den er weist weiter zeigt.



Vergleiche:
Pointer können mit den folgenden Operationen verglichen werden: ==, !=, <, >, >=, <=. **Es werden die Adressen verglichen!** Man bestimmt also ob zwei Pointer auf dieselbe Adresse zeigen, ob ein Pointer in einem Array „weiter vorn“ ist als der andere usw. Der Output ist entweder „wahr“ oder „falsch“ (also 1 oder 0).

Initialisierung von Arrays

Automatische Initialisierung: globale Arrays werden automatisch mit 0 initialisiert, lokale Arrays werden nicht automatisch initialisiert.

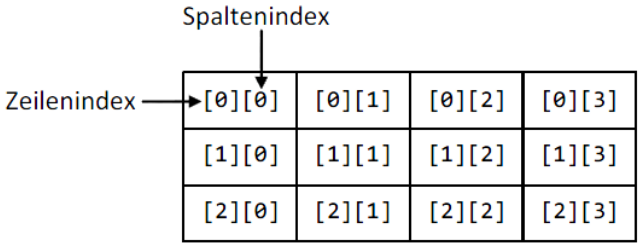
Manuelle Initialisierung: Initialisierung durch eine Liste von Initialisierungswerten. Es sind nur Ausdrücke oder Konstanten möglich, keine Variablen.
int alpha[5] = {1, 2, 3, 4, 5};

Spezialregel: Werden bei der Initialisierung von Arrays weniger Werte angegeben als der Array Elemente hat, so werden die restlichen Elemente mit dem Wert 0 belegt.

Implizierte Längenbestimmung: Hier werden die eckigen Klammern leergelassen. Der Compiler bestimmt die Grösse des Arrays durch Abzählen der Elemente in der geschweiften Klammer:
Int alpha [] = {1, 2, 3, 4};

Mehrdimensionale Arrays

int alpha [3] [4]



Zuweisung:
alpha [1] [3] = 8;

oder:

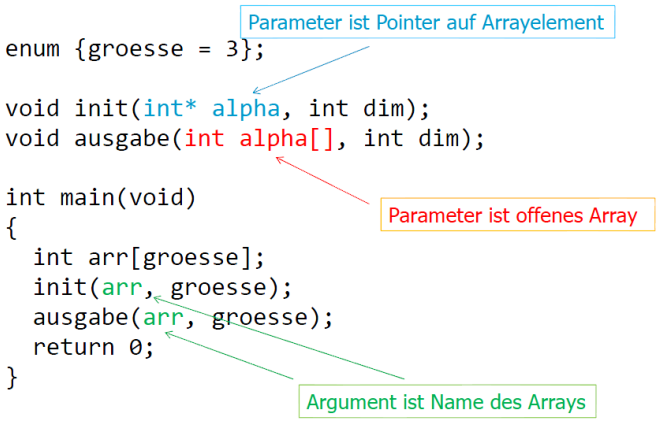
```
int alpha[3][4] = {
                    {1, 3, 5, 7},
                    {2, 4, 6, 8},
                    {3, 5, 7, 9}
                    };
```

1	3	5	7
2	4	6	8
3	5	7	9

Ist äquivalent zur folgende Definition:
int alpha[3][4] = {1, 3, 5, 7, 2, 4, 6, 8, 3, 5, 7, 9};

Übergabe von Arrays

Bei der Übergabe eines Arrays an eine Funktion wird als Argument der Arrayname übergeben.
Der formale Parameter für die Übergabe eines eindimensionalen Arrays kann ein offenes Array sein oder ein Pointer auf den Komponententyp des Arrays.
Die Grösse des Arrays muss immer explizit mitgegeben werden!



Const bei Pointern

const int arr[] = {14, -2, 456};
arr[0], arr[1] und arr[2] sind alle konstant und können somit nach der Initialisierung nicht mehr abgeändert werden.

Von rechts nach links lesen: "arr ist ein Array von int-Konstanten"

Konstanter String:
char str[] = "Ein String";
const char* text = str;

Der Pointer zeigt auf einen konstanten String, der durch den Pointer nicht geändert werden darf. "text ist ein Pointer auf eine char-Konstante"

```
char ch = text[1];
text[1] = 's';
text = "Ein anderer String";
str[4] = 'A';
```

Konstanter Pointer:
char str[] = "Ein String";
char* const text = str;
Der Pointer ist konstant und zeigt auf einen String. Die Position des Pointers kann nicht geändert werden. "text ist ein konstanter Pointer auf ein char"

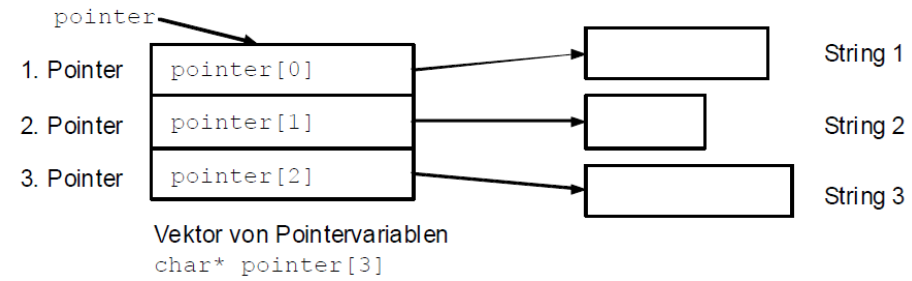
```
char ch = text[1];
text[1] = 's';
text = "Ein anderer String";
str[4] = 'A';
```

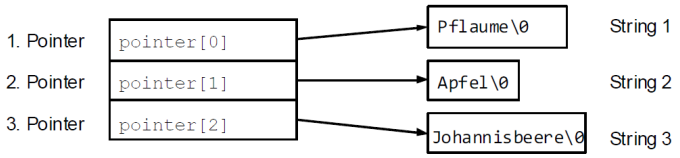
Konstanter Pointer auf konstanten String:
char str[] = "Ein String";
const char* const text = str;

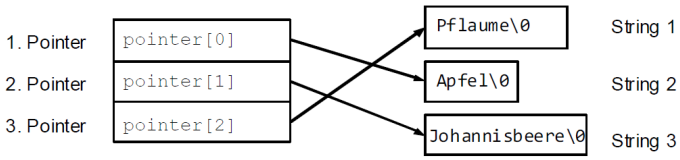
Der Pointer ist konstant uns zeigt auf einen konstanten String. Weder die Position des Pointers noch der String darf durch den Pointer geändert werden. "text ist ein konstanter Pointer auf eine char-Konstante"

```
char ch = text[1];
text[1] = 's';
text = "Ein anderer String";
str[4] = 'A';
```

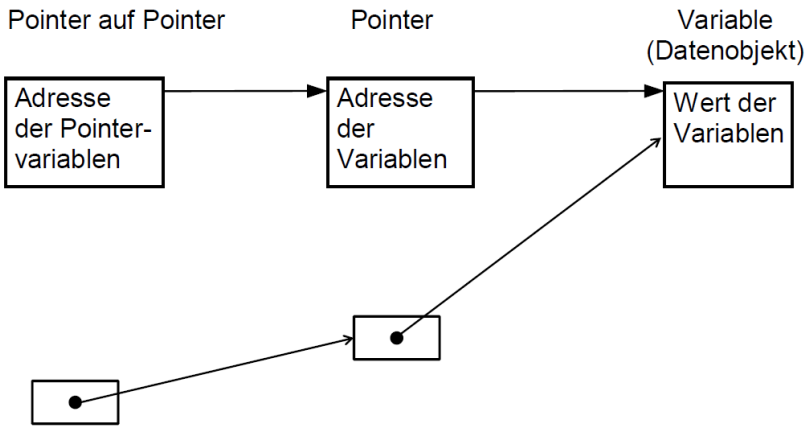
Array von Pointern







Pointer auf Pointer



Char ** text;
„text ist ein Pointer auf einen Pointer auf char“

Pointer auf Funktionen

Jede Funktion befindet sich an einer definierten Adresse im Codespeicher. Diese Adresse kann ebenfalls ermittelt werden.
Der Name der Funktion kann als Adresse auf den ersten Befehl der Funktion verwendet werden (analog Array).

```c
#include <stdio.h>

int foo(char ch)
{
  int i;
  for (i = 1; i <= 10; i++)
    printf("%c ", ch);
  printf("\n");
  return i;
}

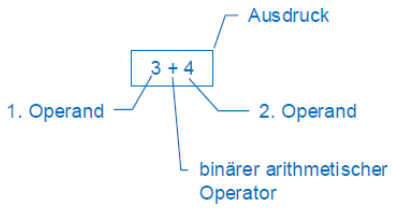
int main(void)
{
  int (*p)(char);   // Deklaration des Funktionspointers
  char c;
  int ret;
  printf("Buchstabe eingeben: ");
  scanf("%c", &c);
  p = foo;          // ermittle Adresse der Funktion foo()
  ret = p(c);       // Aufruf von foo() über Funktionspointer
  return 0;
}
```

Anweisung, Ausdrücke, Operatoren

Stelligkeit von Operatoren

Einstellige (unäre) Operatoren: Der Operator wirkt auf einen einzigen Operanden wie z.B. das Minus als Vorzeichenoperator.

Zweistellige (binäre) Operatoren: Ein Operator benötigt zwei Operanden für eine Verknüpfung.



Dreistelliger (ternärer) Operator: Davon gibt es nur einen, nämlich den Bedingungsoperator „?“.

Postfix- und Präfix-Operatoren

Postfix-Operatoren sind unäre Operatoren, die **hinter** ihrem Operanden stehen.
printf("%d", i**++**);
i wird auf den Bildschirm geschrieben, anschliessend inkrementiert (um eins erhöht).

Präfix-Operatoren sind unäre Operatoren, die **vor** ihrem Operatoren stehen.
printf("%d", **++**i);
i wird zuerst inkrementiert (um eins erhöht), dann auf den Bildschirm geschrieben.

Ausdrücke und Anweisungen

Ausdrücke haben immer einen Rückgabewert. Sie können damit Teil eines grösseren Ausdrucks sein.

3 + 4.5
In diesem Beispiel ist eine **implizierte Typumwandlung** nötig, da 3 vom Typ int und 4.5 vom Typ double ist. Der Rückgabewert ist somit double.

Anweisungen können keinen Rückgabewert haben. Sie können damit nicht Teil eines grösseren Ausdrucks sein.

while (a > 14)

In C kann jeder Ausdruck durch Anhängen eines „;“ eine Anweisung werden.

Nebeneffekte

Nebeneffekte ermöglichen eine schnelle und kurze Programmierschreibweise, sollten aber sparsam gebraucht werden. Somit ist es möglich Programmvariablen während der Auswertung eines Ausdrucks nebenbei zu verändern.

```
int i = 1;
int j;
j = i++;
```

ist äquivalent zu

```
int i = 1;
int j;
j = i;
i = i+1;
```

Auswertungs-reihenfolge

1. Wie in der Mathematik werden als erstes **Teilausdrücke in Klammern** ausgewertet.
2. Dann werden **Ausdrücke mit unären Operatoren** ausgewertet.
   Unäre Operatoren werden *von rechts nach links* angewendet, d.h.
   2.1 **zuerst** werden die **Postfix-Operatoren** auf ihre Operanden
   2.2 und **dann** die **Präfix-Operatoren** auf ihre Operanden angewendet.
3. Abschliessend werden Teilausdrücke mit mehrstelligen Operatoren gemäss der **Priorität** der Operatoren ausgewertet.
4. Bei gleicher Priorität der Operatoren entscheidet die **Assoziativität** (von links nach rechts oder von rechts nach links)

Prioritätentabelle siehe Kapitel 7.6.8 Seite 168

Beispiel:

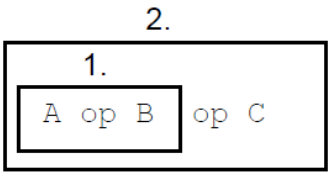
```
int zahl[4] = {4, 7, 66, 235};
int* p = &zahl[0];
```

Operation	*p	p
Start	4	0x22cca0
*p++	4	0x22cca4
*++p	66	0x22cca8
*(p++)	66	0x22ccac
(*p)++	234	0x22ccac
*p	235	0x22ccac

Assoziativitäten

Unter **Assoziativität** versteht man die Reihenfolge wie Opteratoren und Operanden **verknüpft** werden, wenn mehrstellige Operatoren der **gleichen Priorität** über ihre Operanden miteinander verkettet sind.
Die Reihenfolge der Verknüpfung hat mit der Reihenfolge der Auswertung der Operanden nichts zu tun!

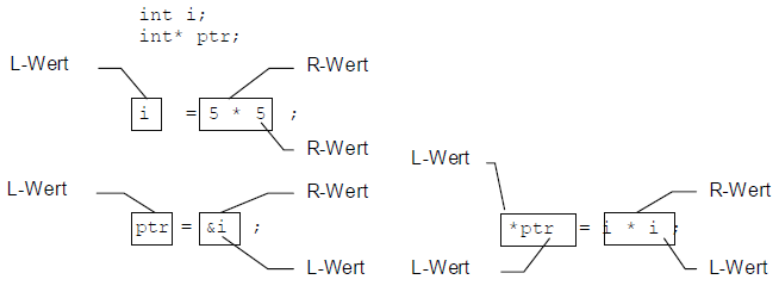
Beispiel:
Verknüpfungsreihenfolge bei einem linksassoziativen Operator op



L- und R-Werte

Ein Ausdruck stellt einen **L-Wert** (lvalue oder left value) dar, wenn er sich **auf ein Speicherobjekt bezieht**. Ein solcher Ausdruck kann **links (und rechts)** des Zuweisungsoperators stehen.

Ein Ausdruck, der sich **nicht auf ein Speicherobjekt bezieht**, kann nur **rechts** des Zuweisungsoperators stehen. Er wird als **R-Wert** (rvalue oder right value) bezeichnet. Einem R-Wert kann nichts zugewiesen werden.



Operatoren

Alle Operatoren sind im Buch im Kapitel 7.6 ab der Seite 145 detailiert beschrieben.

Unäre Operatoren:
Positiver Vorzeichenoperator: **+A**
Negativer Vorzeichenoperator: **-A**
Postfix-Inkrementoperator: **A++**
Präfix-Inkrementoperator: **++A**
Postfix-Dekrementoperator: **A--**
Präfix-Dekrementoperator: **--A**

Binäre Operatoren:
Additionsoperator: **A + B**
Subtraktionsoperator: **A - B**
Multiplikationsoperator: **A * B**
Divisionsoperator: **A / B**
Modulooperator: **A % B**

Zuweisungsoperator:
Zuweisungsoperator: =

Kann verkürzt dargestellt werden:
a = a / b;
a /= b;

Vergleichsoperatoren:
Gleichheitsoperator: **A == B**
Ungleichheitsoperator: **A != B**
Grösseroperator: **A > B**
Kleineroperator: **A < B**
Grössergleichoperator: **A >= B**
Kleinergleichoperator: **A <= B**

Logische Operatoren:
Logisch UND (AND): **A && B**
Logisch ODER (OR): **A || B**
Logisch NICHT (NOT): **!A**

Bitweise Operatoren:
Bitweises AND: **A & B**
Bitweises OR: **A | B**
Bitweises NOT (Inverter): **~A**
Bitweises XOR: **A ^ B**

Beispiel für AND:
0010 1110
1100 1101

0000 1100

Schiebe-Operatoren:
Mit den Schiebe-Operatoren werden Bits nach links bzw. rechts verschoben. Es darf nur um ganzzahlige positive Werte verschoben werden. Beim schieben gehen zwangsläufig Bits verloren. Auf der anderen Seite wird mit Nullen aufgefüllt.

Rechts-Shift um n Bits: **A >> n**
Links-Shift um n Bits: **A << n**

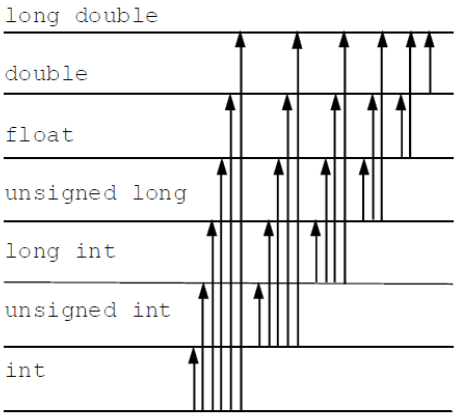
unsigned char a;
a = 8; 0000 1000 Bitmuster von 8
a = a >> 3; 0000 0001 Bitmuster von 1

Bedingungs-Operator (einziger ternäre Operator):
A ? B : C

Ist eine verkürzte Schreibweise für:
```
if (A)
  B;
else
  C;
```

Typum-wandlung

Implizite (automatische) Typumwandlung:
Wird vom Compiler automatisch ausgeführt.



Explizite Typumwandlung:
Mit Hilfe eines cast-Operators kann eine explizite Typumwandlung durchgeführt werden. So können implizite Typumwandlungen unterdrückt werden. Aber Achtung, dieses Vorgehen ist sehr fehlerträchtig.

Beispiel:
```
int a = (int)4.6;              // a == 4
double d = (double)a;      // d == 4.0
```

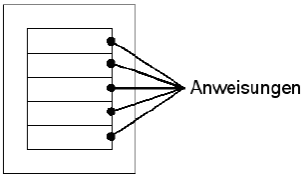
Erlaubte Typumwandlung :
Es kann nicht jeder Typ in einen beliebig anderen Typ umgewandelt werden. Erlaubt sind daher:
Umwandlungen zwischen skalaren Typen (Integer, Floating Points, Pointer) und das Umwandeln von skalarem Typ in void.

Kontrollstrukturen

Sequenz

Mehrere Anwendungen können mit geschweiften Klammern zu einem Block zusammengefasst wreden. Ein Block zählt syntaktisch als eine einzige Anwendung. Die Anweisungen zwischen den Blockbegrenzern werden sequentiell abgearbeitet.

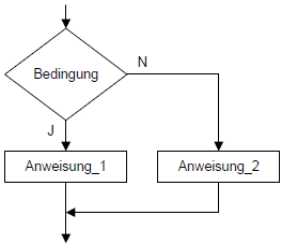
```
{
 Anweisung_1;
 Anweisung_2;
 Anweisung_3;
}
```

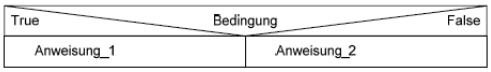


Selektion

if und else – einfache Alternative:

```
if (Ausdruck)
 Anweisung_1;
else
 Anweisung_2;
```





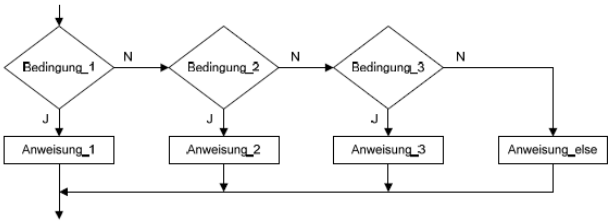
Der else-Zweig ist optional. Entfällt der else-Zweig, sp spricht man von einer bedingten Anweisung.

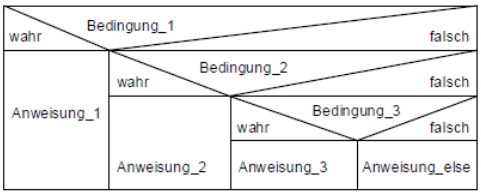
Nach einem Schlüsselwort nie ein „;“ ausser bei do while !!

if und else – mehrfache Alternative:

```
if (Ausdruck_1)
 Anweisung_1;

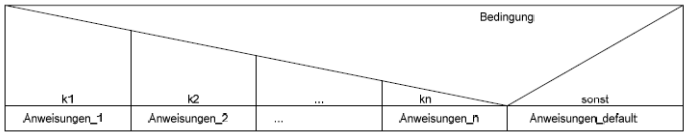
else if (Ausdruck_2)
 Anweisung_2;
else
 Anweisung_3;
```





switch – mehrfache Alternative:

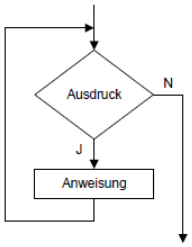
```
switch (Ausdruck)
{
 case k1:
   Anweisung_1;
   break;
 case k2:
   Anweisung_2;
   Break;
 default:
   Anweisung_3;
}
```



Mit break wird der switch verlassen.

Iteration **while – Schleife:**

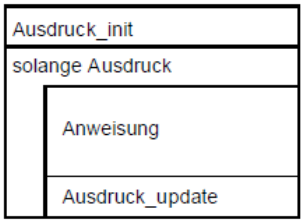
```
while
(Ausdruck)
{
 Anweisung_1;
 Anweisung_2;
}
```





for – Schleife

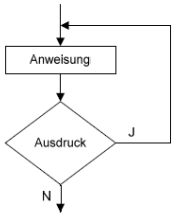
```
for ( Ausdruck_1; Ausdruck_2; Ausdruck_3)
{
 Anweisung_1;
 Anweisung_2;
}
```



Ausdruck_1: Initialisierung einer Laufvariable
Ausdruck_2: Prüfung der Schleifenbedingung
Ausdruck_3: gegebenfalls Ausführung von Anweisung und Erhöhung des Wertes der Laufvariable (wir erst nach dem Abarbeiten des Schleifenrumpfs durchgeführt)

do while – Schleife:

```
do
{
 Anweisung;
} while (Ausdruck);
```





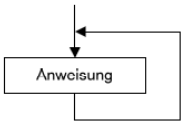
Zuerst wird die Anweisung ausgeführt und dann der Ausdruck ausgewertet.

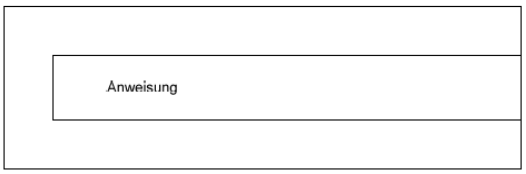
Endlosschleife:

```
for ( ; ; )
 Anweisung;

oder

while (1)
 Anweisung;
```





Sprung-anweisungen

break
do while –, while –, for – Schleife und switch – Anweisungen abbrechen

continue
in den nächsten Schleifendurchgang (Schleifenkopf) springen bei do while –, while – und for – Schleife

return
aus Funktion an aufrufende Stelle zurückspringen

goto (don't do it!)
innerhalb einer Funktion an eine Marke (Label) springen

Für Beispiele siehe im Buch Kapitel 8.4 Seiten 203 bis 206

Blöcke und Funktionen

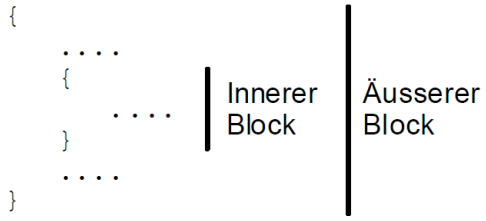
Block

Mit einem Block können **mehrere Anweisungen zusammengefasst** werden. Ein Block zählt dann syntaktisch als eine einzige Anweisung. Ein Block wird mit geschweiften Klammern eingefasst **{ … }**.
Sowohl die öffnende als auch die schliessende geschweifte Klammer werden nicht mit einem Strichpunkt abgeschlossen.
In C müssen die **Variablendefinitionen am Anfang eines Blocks** stehen (vor den Anweisungen).
Alle in einem Block getätigten Vereinbarungen (z.B. Variablendefinitionen) sind genau innerhalb dieses Blocks gültig und sichtbar, ausserhalb nicht.

Einsatzgebiete eines Blocks:
Der Rumpf einer Funktion ist ein Block.
Anweisungsfolgen können mit Hilfe von Blocks gegliedert werden.

Codierstil:
Den Inhalt eines jeden Blocks soll eingerückt werden (zwei Leerzeichen, keine Tabulatoren). Jede Zeile soll maximal 80 Zeichen lang sein.

Schachtelung:
Da eine Anweisung eines Blocks selbst wieder ein Block sein kann, können Blöcke geschachtelt werden.



Gültigkeit

Die Gültigkeit einer Variablen bedeutet, dass **an einer Programmstelle der Namen einer Variablen dem Compiler durch eine Vereinbarung bekannt ist.**

Sichtbarkeit

Die Sichtbarkeit einer Variablen bedeutet, dass man **von einer Programmstelle aus** die Variable sieht, das heisst, dass man **auf sie über ihren Namen zugreifen kann**.

Gültige Variable können für den Programmierer unsichtbar sein, wenn sie durch eine andere Variable desselben Namens verdeckt werden. Das heisst wenn eine lokale und eine globale Variable denselben Namen haben, überdeckt die lokale die globale Variable.

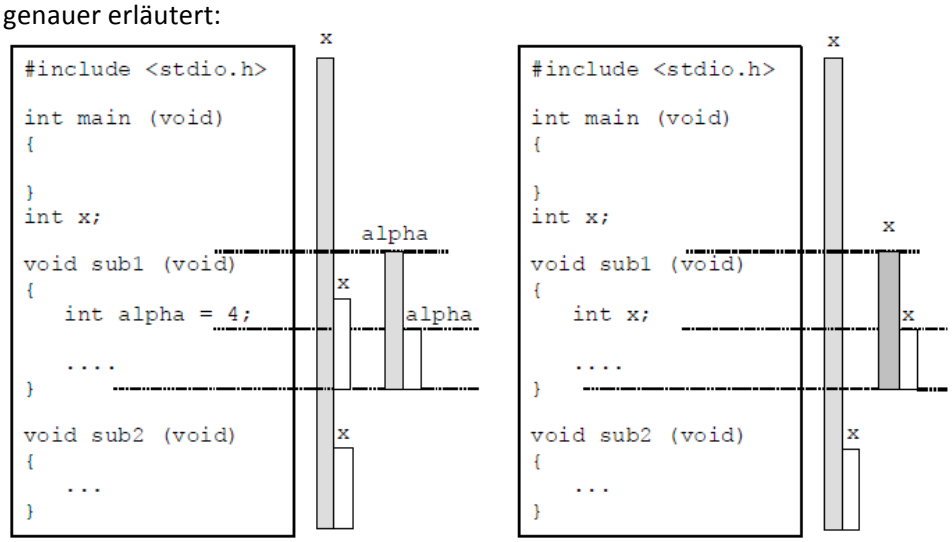
Variablen von inneren Blöcken sind nach aussen nicht sichtbar, aber Variablen in äusseren Blöcken und globale Variablen sind in inneren Blöcken sichtbar.

Lebensdauer

Die Lebensdauer ist die Zeitspanne, in der das Laufzeitsystem des Compilers der Variablen einen Platz im Speicher zur Verfügung stellt. **Also während ihrer Lebensdauer besitzt eine Variable einen Speicherplatz.**

Globale Variable leben so lange wie das Programm. Lokale Variablen werden beim Aufruf des Blockes angelegt und beim Verlassen des Blockes endet ihre Lebensdauer automatisch.

Im folgenden Beispiel wir die Lebensdauer (grau) und die Sichtbarkeit (weiss) genauer erläutert:



Funktionen

Funktionen haben die Aufgabe Teile eines Programmes, die funktional zusammengehören, unter einem Namen zusammen zu fassen. So können grosse Probleme in Teilprobleme unterteilt werden. Mit Hilfe seines Namens kann man dann den Programmteil aufrufen.

Der **Funktionskopf** legt die Aufrufschnittstelle der Funktion fest. Er besteht aus, **Rückgabetyp, Funktionsname und Parameterliste.**

Der **Funktionsrumpf** enthält die lokalen Vereinbarungen und Anweisungen innerhalb eines Blocks.

Eingabedaten sind die an die Parameterliste übergebenen Werte und die Werte der globalen Variablen.

Ausgabedaten sind die Rückgabewerte der Funktion und Änderungen an Variablen, deren *Adresse* über die Parameterliste an die Funktion übergeben wurde.

kein Rückgabewert parameterlos

```
void printGestrichelteLinie(void)
{
  printf("----------------------------");
}
...
printGestrichelteLinie();   // Aufruf
```

kein Rückgabewert Parameterliste

```
void printSumme(int a, int b)
{
  printf("%d", a + b);
}
...
int zahl = 14;
printSumme(zahl, 54);   // Aufruf
```

Typ des Rückgabewertes Parameterliste

```
int getSumme(int a, int b)
{
  return (a + b);
}
...
int zahl = 14;
int summe;
summe = getSumme(zahl, 54);   // Aufruf
```

Klammer ist optional

Funktionsprototyp:
Steht die Definition einer Funktion im Programmcode erst nach ihrem Aufruf, so muss mit einem **Funktionsprototyp** die Funktion vor ihrem Aufruf deklariert werden. Der Funktionsprototyp enthält Name der Funktion, den Typ ihres Rückgabewerts und deren Parameterliste.

```
void init(int* beta);    /* das ist der Funktionsprototyp   */

int main(void)
{
  int a;
  init(&a);
  printf("Der Wert von a ist %d", a);
  return 0;
}

void init(int* alpha)    /* das ist die Funktionsdefinition */
{
  *alpha = 10;
}
```

Parametername kann unterschiedlich sein, ist aber nicht zu empfehlen

Fehlt der Prototyp ganz, so wird die Funktion implizit (automatisch vom System) deklariert. Ihr Rückgabetyp wird als int angenommen, die Parameter werden nicht überprüft.

Strukturen und Unionen

Struktur

Daten, welche logisch zusammengehören, können zusammengenommen Werden. Die Struktur setzt sich aus verschiedenen Feldern zusammen, welche unterschiedlichen Datentypen angehören können. Diese Felder haben spezifische Namen.

Aufbau:
Es wird das Schlüsselwort **struct** verwendet.

```
struct StructName
{
FeldTyp1 feld1;
FeldTyp2 feld2;
FeldTyp3 feld3;
...
FeldTypN feldN;
};
```

Beispielstruktur für einen Angestellten:

```
struct Adresse
{
  char strasse[20];
  int hausnummer;
  int plz;
  char ort[20];
};
```

```
struct Angestellter
{
  int personalnummer;
  char name[20];
  char vorname[20];
  struct Adresse wohnort;
  struct Adresse arbeitsort;
  float gehalt;
};
```

Die erstellten Strukturen können jetzt mehrfach verwendet werden z.B. für mehrere Angestellte.
struct Angestellter mueller;
struct Angestellter bonderer;

struct Angestellter vertrieb[20];
Ist ein Array von 20 Strukturvariablen.

Operationen auf Struktur-variablen

Zuweisung
Liegen zwei Strukturvariablen a und b vom gleichen Strukturtyp vor, so kann der Wert der einen Variablen der anderen zugewiesen werden. Alle Komponentenwerte der Variablen b werden jenen der Variablen a zugewiesen.
a = b;

Ermittlung der Grösse der Struktur
Kann nicht aus den Grössen der einzelnen Komponenten bestimmt werden sondern muss mit dem Operator sizeof ermittelt werden.
sizeof (struct Angestellter);

Ermittlung der Adresse der Strukturvariablen
Adressen werden mit dem Adressoperator & ermittelt.

Zugriff auf eine Struktur-variable

Der Zugriff auf ein Feld einer Strukturvariablen erfolgt über den Namen der Strukturvariablen, gefolgt von einem Punkt und dem Namen des Feldes

```
mueller.personalnummer = 34259;
bonderer.wohnort.plz = 7208;
strcpy(mueller.vorname, "Fritz");
printf("%s\n", vertrieb[14].name);
```

Wenn der Zugriff über einen Pointer auf eine Strukturvariable erfolgt, über den Namen des Pointers, gefolgt von einem Pfeil (->) und dem Namen des Feldes

```
struct Angestellter* pMitarbeiter = &bonderer;
pMitarbeiter->personalnummer = 65433;
(*pMitarbeiter).personalnummer = 65433;
pMitarbeiter->arbeitsort.plz = 8640;
```

Lage im Speicher

Alignment:

Die Felder einer Strukturvariablen werden nacheinander gemäss der Definition in den Speicher platziert. Gewisse Datentypen verlangen unter Umständen, dass Sie auf eine Wortgrenze (gerade Adresse) gelegt werden. Dies nennt man **Alignment**. Durch das Alignment kann es vorkommen, dass einzelne Bytes nicht verwendet werden, d.h. im Speicher ausgelassen werden.

```
struct B
{
  int   wert;
  char  text[3];
  int   zahl;
};
```

Das int-Feld zahl muss auf einer geraden Adresse beginnen.
Deshalb wird die Adresse 7 hier nicht verwendet.

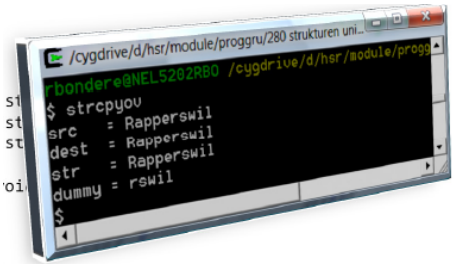
11	zahlLoLo
10	zahlLoHi
9	zahlHiLo
8	zahlHiHi
7	Not Used
6	text[2]
5	text[1]
4	text[0]
3	wertLoLo
2	wertLoHi
1	wertHiLo
0	wertHiHi

Buffer overflow:

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

int main(void)
{
 struct
 {
  char dest[5];
  char dummy[8];
 } s;
 char src[] = "Rapperswil";
 char* str;
 str = strcpy(s.dest, src);

 printf("src = %s\n", src);
 printf("dest = %s\n", s.dest);
 printf("str = %s\n", str);
 printf("dummy = %s\n", s.dummy);
 return 0;
}
```



dummy[7]	
dummy[6]	
dummy[5]	'\0'
dummy[4]	'l'
dummy[3]	'i'
dummy[2]	'w'
dummy[1]	's'
dummy[0]	'r'
dest[4]	'e'
dest[3]	'p'
dest[2]	'p'
dest[1]	'a'
dest[0]	'R'

Über- und Rückgabe einer Struktur-variable

Strukturvariablen können komplett an Funktionen übergeben werden. Der Rückgabetyp einer Funktion kann eine Struktur sein. Dabei wird die Strukturvariable direkt komplett übergeben. Zu beachten ist der Kopieraufwand bei der Übergabe, bzw. Rückgabe eines Wertes. In der Praxis wird deshalb häufig mit Pointern gearbeitet.

```
void foo(struct Angestellter a);                  // grosser Kopieraufwand, nicht ideal

void foo(struct Angestellter* pa);                // nur Pointerübergabe, effizient

void fooRead(const struct Angestellter* pa)       // nur Pointerübergabe, read only
                                                     durch const
```

Initialisierung einer Struktur-variablen

```
struct Angestellter maier =
{
 56321, // personalnummer
 "Maier", // name[20]
 "Hans", // vorname[20]
 {
  "Schillerplatz", // strasse [20]
  14, // hausnummer
  75142, // plz
  "Esslingen" // ort[20]
 }
};
```

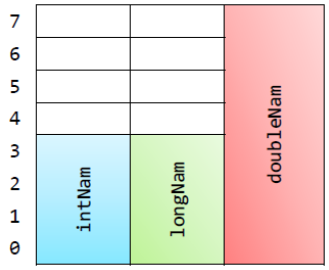
Unionen

Eine Union ist gleich aufgebaut wie eine Struktur, ausser dass der Speicherplatz vom Compiler so gross angelegt wird, dass der Speicher auch für die grösste Alternetive reicht. Somit beginnen alle Alternativen mit derselben Adresse.

Zu einem bestimmten Zeitpunkt ist jeweils nur eine einzige Komponente einer Reihe von Alternativen gespeichert. Deswegen muss der Programmierer verfolgen, welcher Typ jeweils in der Union gespeichert ist. Der Datentyp, der entnommen wird muss der sein, der zuletzt gespeichert wurde.

Es gelten dieselben Operationen wie für Strukturen.

```
union Vario
{
 int intNam;
 long longNam;
 double doubleNam;
};
```



Rekursion und Iteration

Definition

Rekursion:
Funktion enthält Abschnitte, in der sie selbst direkt oder indirekt wieder aufgerufen wird.

$0! = 1$
$n! = n \cdot (n - 1)!$

Iteration:
Algorithmus enthält Abschnitte, die innerhalb einer Ausführung mehrfach durchlaufen werden (Schleife).

$0! = 1$
$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n - 1) \cdot n$

Beispiel

Fakultät iterativ

```
unsigned long
faku(unsigned int n)
{
  unsigned long fak = 1L;
  unsigned int i;
  for (i = 2; i <= n; ++i)
  {
    fak = fak * i;
  }
  return fak;
}
```

Fakultät rekursiv

```
unsigned long
faku(unsigned int n)
{
  if (n > 1)
  {
    return n * faku(n-1);
  }
  else
  {
    return 1;
  }
}
```

Stack

Bei jedem Aufruf von faku() werden die Rücksprungadresse und weitere Verwaltungsinformationen auf einem Stack abgelegt, der durch das Laufzeitsystem verwaltet wird.
Auch die übergebenen Parameter (hier nur einer) werden auf diesem Stack abgelegt. Dabei wächst der Stack mit der Rekursionstiefe der Funktion. Der letzte Aufruf von faku() mit dem Parameter n = 1 bewirkt keine weitere Rekursion, da ja die Abbruchbedingung erfüllt ist.
Der Abbau des Stacks geschieht in umgekehrter Reihenfolge, wie aus dem folgenden Bild ersichtlich wird.

Speicherklassen

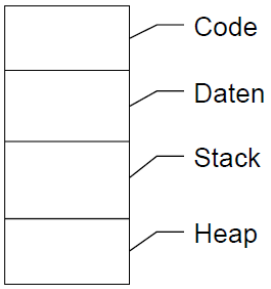
Adressraum

Der Adressraum eines ablauffähigen C-Programms – also einer Datei programm.exe – besteht aus den vier Segmenten Code, Daten, Heap und Stack.

Das Codesegment kann nur gelesen und ausgeführt werden. Es befindet sich im Flash EPROM oder im RAM. Das Datensegment kann gelesen und beschrieben werden. Es befindet sich im RAM und eventuell in Registern.

Code:
Hier liegt das Programm im Maschinencode. Häufig werden in diesem Segment auch Konstanten abgelegt.

Daten:
Hier sind die globalen (externen) und static Variablen zu finden.



Stack:
Hier befinden sich die lokalen Variablen, Parameter einer Funktion und Rücksprungadressen.

Heap:
Hier werden die dynamischen Variablen abgelegt.
(Speicherplatz wird erst zur Laufzeit angefordert)

Programm aus mehreren Dateien

C unterstützt eine **getrennte Kompilierung**. Dies bedeutet, dass es möglich ist, ein grosses Programm in Module zu zerlegen, die getrennt kompiliert werden. Ein Modul entspricht einer Datei.

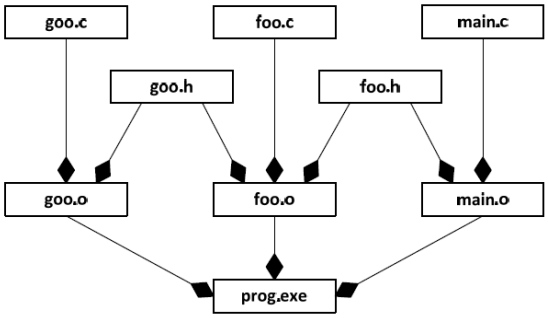
Übersetzungsvorgang:
Der Compiler erzeugt aus jeder Quelldatei (programm.c), welche getrennt übersetzt werden müssen, eine Object-Datei mit Maschinencode.
Eine Objectdatei ist nicht ausführbar, da die Adressen der Funktionen (und globalen Variablen), die in einer anderen Datei definiert sind, noch nicht bekannt sind.

gcc -c foo.c

Der Linker löst die noch offenen Referenzen auf, indem er alle zum ausführbaren Programm gehörenden Object-Dateien linkt.

gcc -o foo.exe foo.o goo.o hoo.o

Beispiel für eine mögliche Verknüpfung von Dateien:



Make-File:
Der Buildprozess enthält alle vorher beschriebenen Schritte. Es wäre mühsam, wenn diese Befehle jedesmal neu eingetippt werden müssten.
Deshalb wird in der Praxis oft ein Buildscript oder ein Buildtool eingesetzt, z.B. **make** oder **ant.**

Make-File zum gezeigten Beispiel:

```
# makefile
cc=gcc
CFLAGS=-g -c -Wall
LFLAGS=-g -Wall
OBJ=foo.o goo.o main.o
EXE=prog.exe

${EXE}: ${OBJ}
        ${cc} ${LFLAGS} -o ${@} ${OBJ}

foo.o: foo.h goo.h foo.c
        ${cc} ${CFLAGS} -o ${@} foo.c

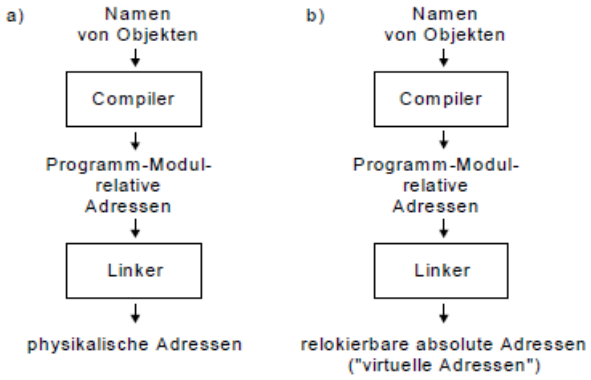
goo.o: goo.h goo.c
        ${cc} ${CFLAGS} -o ${@} goo.c

main.o: foo.h main.c
        ${cc} ${CFLAGS} -o ${@} main.c

clean:
        rm -f ${OBJ} ${EXE}
```

Der Befehl (gcc) wird nur dann ausgeführt, wenn sich an den Dateien, zu denen eine Abhängigkeit besteht, etwas geändert hat.

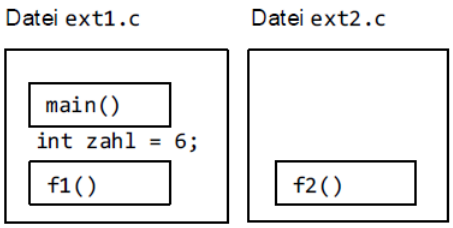
Feste vs. relokative (virtuelle) Adresse:
Legt der Linker noch keine **physikalischen**, sondern nur **virtuelle** Adressen fest, so ist das Programm im Arbeitsspeicher verschiebbar. Die Zuordnung von virtuellen zu physikalischen Adressen führt dann die Komponente **Memory Manager** des Betriebssystems durch.
Wird beim Linken bereits eine physikalische Adresse vergeben, so ist das Programm im Arbeitsspeicher nicht verschiebbar. Es wird an die vom Linker berechnete Adresse geladen.



Speicherklasse extern

Im folgenden Beispiel besteht das Programm aus zwei Dateien. Die Funktion f2() der Datei ext2.c möchte auf die Variable zahl der Datei ext1.c zugreifen. Dies ist möglich indem man in f2() die Variable zahl mit der Speicherklasse **extern** deklariert:

extern int zahl;



Eine externe Variable kann nur in einer einzigen Datei definiert werden und kann auch nur in dieser Datei manuell Initialisiert werden.

Header-Datei:
extern-Deklarationen werden üblicherweise in einer Headerdatei deklariert. Am Beginn der .c - Datei wird die Headerdatei mit **#include** eingefügt.

Speicherklasse static

Funktionen und globale Variablen, die als **static** definiert werden, sind nur in ihrer eigenen Datei sichtbar. Funktionen aus anderen Dateien können auf diese Funktionen und Variablen nicht zugreifen.

Wird eine Variable innerhalb eines Blocks als static definiert, ist sie auch nur in diesem Block gültig.

static int zahl = 6;

Achtung! static Variablen sind nur einmal vorhanden (auch in Multithreading-Umgebungen), d.h. ihr Wert wird erhalten, auch wenn die Funktion beendet ist. Beim nächsten Aufruf der Funktion geht es mit dem alten Wert weiter.

Speicherklasse bei lokalen Variablen

Automatische Variablen: Zu den automatischen Variablen zählen die Speicherklassen auto und register.
Automatische Variablen werden beim Verlassen des Blocks, in dem sie definiert wurden, vom System automatisch aus dem Speicher entfernt. Das ist das übliche Verhalten von lokalen Variablen. Der Wert einer automatischen Variablen wird für den nächsten Aufruf nicht beibehalten (logische und bekannte Folgerung)

auto: Variablen der Speicherklasse auto werden auf dem Stack angelegt.
Wenn bei einer Variablen keine Speicherklasse angegeben wird, so hat sie "automatisch" die Speicherklasse auto, d.h. auto ist der Default.
Die Speicherklasse auto wird praktisch nie explizit angegeben.

auto int a = 3;

register: Bei Angabe der Speicherklasse register wird dem Compiler empfohlen, für diese Variable ein Register (schnellste Speicherzelle) zu verwenden. Der Compiler wird dies versuchen, eine Garantie besteht aber nicht, da dem Compiler z.B. u.U. gar nicht genügend Register zur Verfügung stehen.

register long a;

Der Compiler optimiert meist gut, register soll deshalb üblicherweise nicht angegeben werden.

static: Wie bei allen Variablen der Speicherklasse static werden diese Variablen im globalen Datenbereich (nicht auf dem Stack) angelegt.
Das einzige spezielle von lokalen static Variablen gegenüber globalen static Variablen ist, dass die Sichtbarkeit auf den Block beschränkt ist, in welchem die Variable definiert ist.

Wichtige Funktionen

printf int printf (const char * format, …);
Steht in der Header-Datei stdio.h

Zeilenumbruch: \n

Formatelemente:
int: %d
char: %c, strings: %s
double / float: %lf, %e, %E, %f, %g, %G

Feldbreite:
%f: gibt eine Gleitpunktzahl im Default-Format des Compilers aus.
%5.2f: gibt eine Gleitpunktzahl mit insgesamt 5 Stellen aus, 2 hinter dem Punkt, 1 für den Punkt unnd 2 vor dem Punkt.
%5.0f: die Gleitpunktzahl wird mit 5 Stellen ausgegeben ohne Punkt und ohne Stelle nach dem Punkt.
%.3f: es sollen 3 Stellen hinter dem Punkt und der Punkt ausgegeben werden. Für die Zahl der Stellen vor dem Punkt erfolgt keine Anweisung.
%nd: Bei der Feldbreite von ganzen Zahlen zeigt n die Feldbreite an.

Beispiele:
printf(„Bitte eine Zahl eingeben: \n„);
printf(„Der Wert von a ist %d“, a);

double a = 0.000006;
printf(„\n%e \n%E \n%f \n%g \n%G“, a, a, a, a, a);

Ausgabe:
6.000000e-06
6.000000E-06
0.000006
6e-06
6E-06

float z = 12.3456f;
printf(„\n%f \n%5.2f \n%5.0f \n%.3f“, z, z, z, z);

Ausgabe:
12.345600
12.35
12
12.346

scanf Int scanf (const char * format, …);
Der Aufbau von scanf ist derselbe wie der von printf.

gets	Char * gets (char * s); Liest einen String in ein Array von Zeichen ein, auf das der Pointer s zeigt. Die Funktion wartet so lange bis eine Eingabezeile mit <RETURN> abgeschlossen wurde. An das letzte in das Array eingelesene Zeichen wird ein Stringende-Zeichen \0 angehängt.
getchar	Int getchar (void); Liest ein einzelnes Zeichen als unsigned char ein und wandelt es in int um. Die Funktion wartet so lange bis eine Eingabezeile mit <RETURN> abgeschlossen wurde.

Fragen:

- Wie werden Pointer zusammengezählt bzw. voneinander subtrahiert?
- Was genau ist der Vorteil von char * pointer = „hello“; gegenüber char buffer [] = „hello“;?
- Ist es richtig das die Speicherklasse extern auf den stack einer Datei zugreift?
- Wann wird getchar verwendet?

Nichtbehandelte Themen:

- Einführung in C
- Komplexere Datentypen, eigene Typnamen und Namensräume
- Input und Output
- Scope (Ausdrücke)
- Information Hiding Kapitel 19