

H Zeiger(-Variablen)

H.1 Einordnung

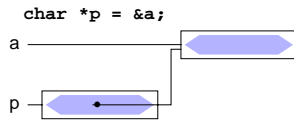
- **Konstante:**
Bezeichnung für einen Wert

'a' ≡ 0110 0001

- **Variable:**
Bezeichnung einer Speicherstelle, die einen Wert enthält



- **Zeiger-Variabe (Pointer):**
Bezeichnung einer Speicherstelle, die als Wert die Adresse einer Speicherstelle enthält, die dann schließlich einen Wert enthält



1 Definition von Zeigervariablen

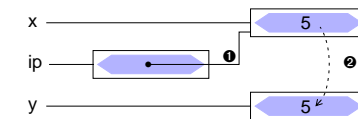
[H.2 Überblick](#)

- Syntax:

Typ *Name ;

2 Beispiele

```
int x = 5;  
int *ip;  
int y;  
ip = &x; ❶  
y = *ip; ❷
```



H.2 Überblick

- Eine Zeigervariabe (**pointer**) enthält als Wert die Adresse einer anderen Variablen
 - ↳ der Zeiger verweist auf die Variable
- Über diese Adresse kann man **indirekt** auf die Variable zugreifen
- Daraus resultiert die große Bedeutung von Zeigern in C
 - ↳ Funktionen können ihre Argumente verändern (**call-by-reference**)
 - ↳ dynamische Speicherverwaltung
 - ↳ effizientere Programme
- Aber auch Nachteile!
 - ↳ Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variable zugreifen?)
 - ↳ häufigste Fehlerquelle bei C-Programmen

H.3 Adreßoperatoren

1 Adreßoperator &

&x der unäre Adreß-Operator liefert die Adresse der Variablen (des Objekts) x

2 Verweisoperator *

*x der unäre Verweisoperator * ermöglicht den Zugriff auf den Inhalt der Variablen (des Objekts), auf die der Zeiger x verweist

3 Der Operator []

x[y] x ist ein Zeiger auf das erste einer Reihe von Objekten (z. B. ein Feld)
y ist ein Index (*offset*), der bestimmt, auf welches der Objekte zugegriffen wird
– der Zugriff erfolgt durch eine implizite Addition, bei der die Größe des Objekts berücksichtigt wird

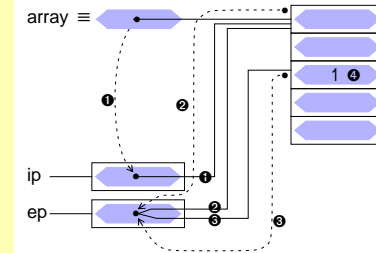
H.4 Zeiger als Funktionsargumente

- Parameter werden in C *by-value* übergeben
- die aufgerufene Funktion kann den aktuellen Parameter beim Aufrufer nicht verändern
- auch Zeiger werden *by-value* übergeben, d. h. die Funktion erhält lediglich eine Kopie des Adreßverweises
- über diesen Verweis kann die Funktion jedoch mit Hilfe des ***-Operators auf die zugehörige Variable zugreifen und sie verändern
↳ *call-by-reference*

H.5 Zeiger und Felder (1)

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];
int *ip = array; ❶
int *ep;
ep = &array[0]; ❷
ep = &array[2]; ❸
*ep = 1; ❹
```



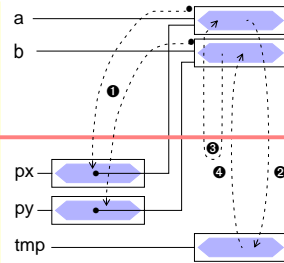
H.4 ... Zeiger als Funktionsargumente (2)

- Beispiel:

```
main(void) {
int a, b;
void swap (int *, int *);
...
swap(&a, &b); ❶
}
```

```
void swap (int *px, int *py)
{
int tmp;

tmp = *px; ❷
*px = *py; ❸
*py = tmp; ❹
}
```

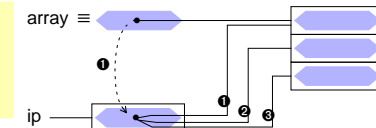


H.6 Arithmetik mit Adressen

[H.6 Arithmetik mit Adressen](#)

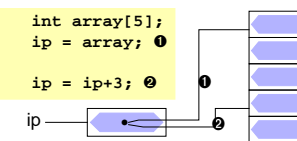
- ++ -Operator: Inkrement = nächstes Objekt

```
int array[3];
int *ip = array; ❶
ip++; ❷
ip++; ❸
```



- -- -Operator: Dekrement = vorheriges Objekt

```
int array[5];
ip = array; ❶
ip = ip+3; ❷
```



Dabei wird immer die Größe des Objekttyps berücksichtigt!

- !!! **Achtung:** Assoziativität der Operatoren beachten !!

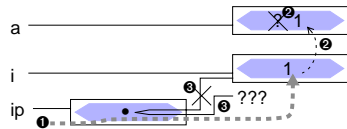
1 Vorrangregeln bei Operatoren

Operatorklasse	Operatoren	Assoziativität
primär	() Funktionsaufruf []	von links nach rechts
unär	! ~ ++ -- + - * &	von rechts nach links
multiplikativ	*/%	von links nach rechts
...		

2 Beispiele

```
int a, i, *ip;
i = 1;
ip = &i;
```

```
a = *ip++;
(1) a = *ip++;
(2) a = *ip++;
```



3 Zeigerarithmetik und Felder

- Ein Feldname ist eine Konstante, für die Adresse des Feldanfags
 - Feldname ist ein ganz normaler Zeiger
 - Operatoren für Zeiger anwendbar (*, [])
 - aber keine Variable → keine Modifikationen erlaubt
 - keine Zuweisung, kein ++, --, +=, ...

es gilt:

```
int array[5]; /* → array ist Konstante für den Wert &array[0] */
int *ip = array; /* ≡ int *ip = &array[0] */
int *ep;

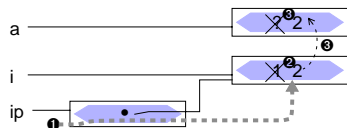
/* Folgende Zuweisungen sind äquivalent */
array[i] = 1;
ip[i] = 1;
*(ip+i) = 1; /* Vorrang ! */
*(array+i) = 1;

ep = &array[i]; *ep = 1;
ep = array+i; *ep = 1;
```

2 Beispiele (2)

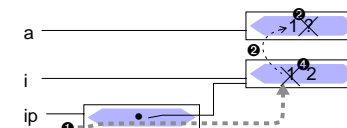
```
int a, i, *ip;
i = 1;
ip = &i;
```

```
a = ++*ip;
(1) a = ++*ip;
(2) a = ++*ip;
```



```
int a, i, *ip;
i = 1;
ip = &i;
```

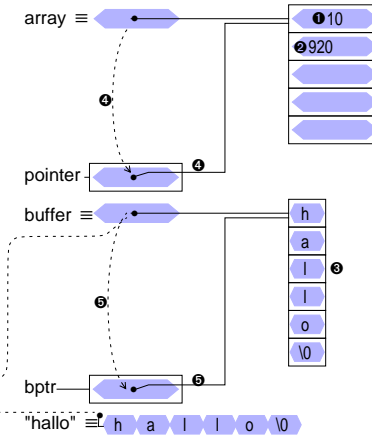
```
a = (*ip)++;
(1) a = (*ip)++;
(2) a = (*ip)++;
```



3 Zeigerarithmetik und Felder

```
int array[5];
int *pointer;
char buffer[6];
char *bptr;

(1) array[0] = 10;
(2) array[1] = 920;
(3) strcpy(buffer, "hallo");
(4) pointer = array;
(5) bptr = buffer;
```



Formale Parameter
der Funktion strcpy

```
ziel ← [ ]
quelle ← [ ]
```

"hallo" = h a l l o \0

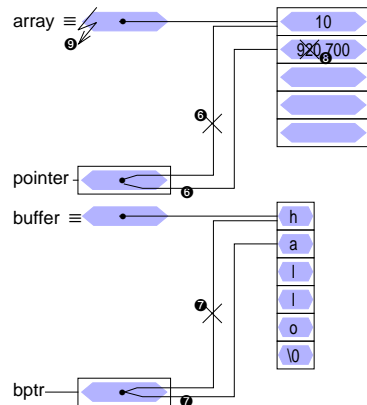
3 Zeigerarithmetik und Felder

H.6 Arithmetik mit Adressen

```
int array[5];
int *pointer;
char buffer[6];
char *bptr;
```

```
1 array[0] = 10;
2 array[1] = 920;
3 strcpy(buffer, "hallo");
4 pointer = array;
5 bptr = buffer;
```

```
6 pointer++;
7 bptr++;
8 *pointer = 700;
9 array++;
```



C-Ing

Einführung in die Programmierung für Ingenieure — C
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

H-Zeiger.doc 1998-07-22 14.22

H.13

Reproduktion jeder Art oder Vervielfältigung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

H.7 Felder als Funktionsparameter

- Funktionsaufruf und Deklaration der formalen Parameter am Beispiel eines `int`-Feldes:

```
int a, b;
int feld[20];
func(a, feld, b);
...
int func(int p1, int p2[], int p3);
oder:
int func(int p1, int *p2, int p3);
```

- Der Feldname ist eine Konstante Adresse, die als Argument eines Funktionsaufrufs *by-value* übergeben wird (= Funktion erhält Kopie!)
- Diese Kopie kann von der aufgerufenen Funktion – unabhängig von der Deklaration des formalen Parameters – wie eine Zeigervariable verwendet werden
- die Parameter-Deklarationen `int p2[]` und `int *p2` sind vollkommen äquivalent!

C-Ing

Einführung in die Programmierung für Ingenieure — C
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

H-Zeiger.doc 1998-07-22 14.22

H.15

Reproduktion jeder Art oder Vervielfältigung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Vergleichsoperatoren und Adressen

H.6 Arithmetik mit Adressen

- Neben den arithmetischen Operatoren lassen sich auch die Vergleichsoperatoren auf Zeiger (allgemein: Adressen) anwenden:

- < kleiner
- <= kleiner gleich
- > größer
- >= größer gleich
- == gleich
- != ungleich

C-Ing

Einführung in die Programmierung für Ingenieure — C
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

H-Zeiger.doc 1998-07-22 14.22

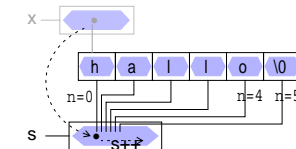
H.14

Reproduktion jeder Art oder Vervielfältigung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

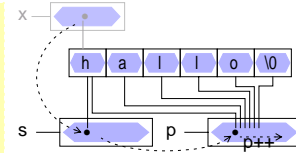
H.8 Zeiger, Felder und Zeichenketten

- Zeichenketten sind Felder von Einzelzeichen (`char`), die in der internen Darstellung durch ein `'\0'`-Zeichen abgeschlossen sind
- Beispiel: Länge eines Strings ermitteln — Aufruf `strlen(x)`;

```
/* 1. Version */
int strlen(char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
```



```
/* 2. Version */
int strlen(char *s)
{
    char *p = s;
    while (*p != '\0')
        p++;
    return(p-s);
}
```



C-Ing

Einführung in die Programmierung für Ingenieure — C
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

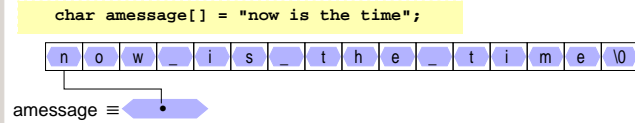
H-Zeiger.doc 1998-07-22 14.22

H.16

Reproduktion jeder Art oder Vervielfältigung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

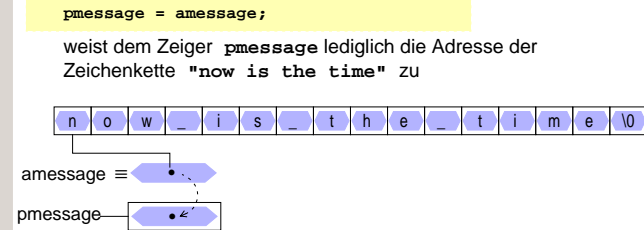
H.8 ... Zeiger, Felder und Zeichenketten (2)

- wird eine Zeichenkette zur Initialisierung eines `char`-Feldes verwendet, ist der Feldname ein konstanter Zeiger auf den Anfang der Zeichenkette



H.8 ... Zeiger, Felder und Zeichenketten (4)

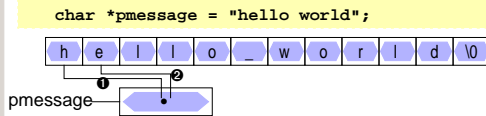
- die Zuweisung eines `char`-Zeigers oder einer Zeichenkette an einen `char`-Zeiger bewirkt kein Kopieren von Zeichenketten!



- wird eine Zeichenkette als aktueller Parameter an eine Funktion übergeben, erhält diese eine Kopie des Zeigers

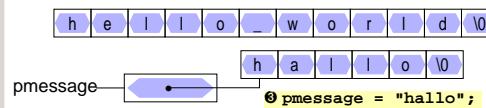
H.8 ... Zeiger, Felder und Zeichenketten (3)

- wird eine Zeichenkette zur Initialisierung eines `char`-Zeigers verwendet, ist der Zeiger eine Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird



```
pmessage++;  
printf("%s", pmessage); /*gibt "ello world" aus*/
```

- wird dieser Zeiger überschrieben, ist die Zeichenkette nicht mehr adressierbar!



H.8 ... Zeiger, Felder und Zeichenketten (5)

- Zeichenketten kopieren

```
/* 1. Version */  
void strcpy(char s[], t[])  
{  
    int i=0;  
    while ( (s[i] = t[i]) != '\0' )  
        i++;  
}  
  
/* 2. Version */  
void strcpy(char *s, *t)  
{  
    while ( (*s = *t) != '\0' )  
        s++, t++;  
}  
  
/* 3. Version */  
void strcpy(char *s, *t)  
{  
    while ( *s++ = *t++ )  
        ;  
}
```

H.8 ... Zeiger, Felder und Zeichenketten (6)

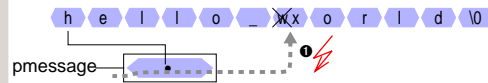
- in ANSI-C können Zeichenketten in nicht-modifizierbaren Speicherbereichen angelegt werden (je nach Compiler)

→ Schreiben in Zeichenketten
(Zuweisungen über dereferenzierte Zeiger)
kann zu Programmabstürzen führen!

- Beispiel:

```
strcpy("zu ueberschreiben", "reinschreiben");
```

```
char *pmessage = "hello world";
```



```
pmessage[6] = 'x';
```

aber!

```
char amessage[] = "hello world";
```



```
amessage[6] = 'x';
```

 ok!

C-Ing

Einführung in die Programmierung für Ingenieure — C

© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

H-Zeiger.doc 1998-07-22 14.22

H.21

Reproduktion jeder Art oder Vervielfältigung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

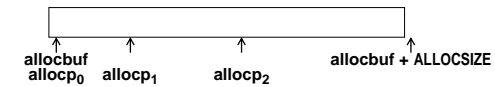
H.9 ... Beispiel: dynamische Speicherverwaltung (2)

- Globale Definitionen

```
#define NULL (char *)0
#define ALLOCSIZE 1000
static char allocbuf[ALLOCSIZE]
static char *allocp = allocbuf;
```

- Anfordern von Speicher

```
char *alloc(int n)
{
    if (allocp+n <= allocbuf+ALLOCSIZE) {
        allocp += n;
        return (allocp - n);
    } else
        return (NULL);
}
```



C-Ing

Einführung in die Programmierung für Ingenieure — C

© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

H-Zeiger.doc 1998-07-22 14.22

H.23

Reproduktion jeder Art oder Vervielfältigung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

H.9 Beispiel: dynamische Speicherverwaltung

- die folgenden zwei Funktionen verwalten einen globalen, zusammenhängenden Speicherbereich,
 - aus dem freier Speicher angefordert werden kann (**alloc**)
 - und in den freigegebenen Speicher wieder integriert wird (**free**)
- diese Version einer dynamischen Speicherverwaltung ist rudimentär und soll lediglich einen Eindruck von einer möglichen Realisierung geben
- in der Regel werden für diese Aufgabenstellung die Funktionen **malloc()** und **free()** aus der C-Bibliothek verwendet

C-Ing

Einführung in die Programmierung für Ingenieure — C

© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

H-Zeiger.doc 1998-07-22 14.22

H.22

Reproduktion jeder Art oder Vervielfältigung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

H.9 ... Beispiel: dynamische Speicherverwaltung (3)

- Freigabe von Speicher

```
void free(char *p)
{
    if (p >= allocbuf &&
        p < allocbuf+ALLOCSIZE)
        allocp = p;
}
```

C-Ing

Einführung in die Programmierung für Ingenieure — C

© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

H-Zeiger.doc 1998-07-22 14.22

H.24

Reproduktion jeder Art oder Vervielfältigung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

H.10 Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden

- Deklaration

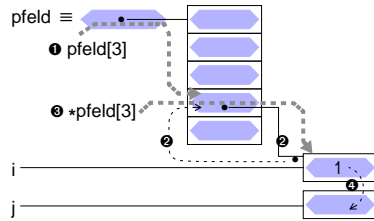
```
int *pfeld[5];
int i = 1;
int j;
```

- Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i;
```

- Zugriffe auf das Objekt, auf das ein Zeiger des Feldes verweist

```
j = *pfeld[3];
```



H.11 Argumente aus der Kommandozeile

- beim Aufruf eines Kommandos können normalerweise Argumente übergeben werden
- der Zugriff auf diese Argumente wird der Funktion `main()` durch zwei Aufrufparameter ermöglicht:

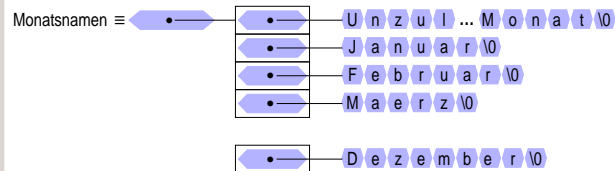
```
int main (int argc, char *argv[])
{
  ...
}
oder
int main (int argc, char **argv)
{
  ...
}
```

- der Parameter `argc` enthält die Anzahl der Argumente, mit denen das Programm aufgerufen wurde
- der Parameter `argv` ist ein Feld von Zeiger auf die einzelnen Argumente (Zeichenketten)
- der Kommandoname wird als erstes Argument übergeben (`argv[0]`)

H.10 Felder von Zeigern (2)

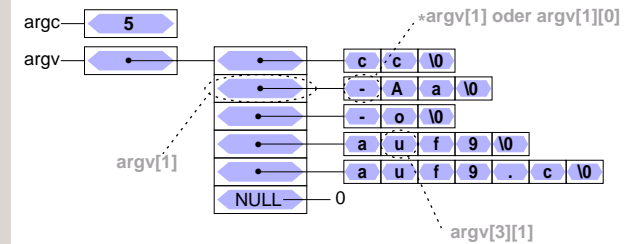
- Beispiel: Definition und Initialisierung eines Zeigerfeldes:

```
char *month_name(int n)
{
  static char *Monatsnamen[] = {
    "Unzulaessiger Monat",
    "Januar",
    ...
    "Dezember"
  };
  return ( (n<0 || n>12) ?
    Monatsnamen[0] : Monatsnamen[n] );
}
```



1 Datenaufbau H.11 Argumente aus der Kommandozeile

```
Kommando: cc -Aa -o auf9 auf9.c
Datei cc.c:
...
main(int argc, char *argv[]) {
...
}
```



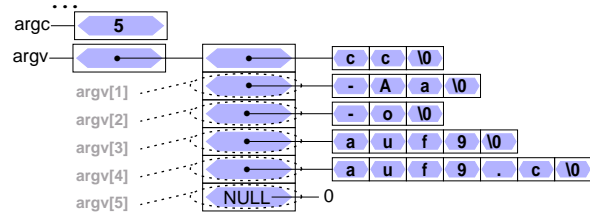
2 Zugriff H.11 Argumente aus der Kommandozeile

Beispiel: Ausgeben aller Argumente (1)

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```
int
main (int argc, char *argv[])
{
    int i;
    for ( i=1; i<argc; i++) {
        printf("%s%c", argv[i],
            (i < argc-1) ? ' ':'\n' );
    }
    ...
}
```

1. Version

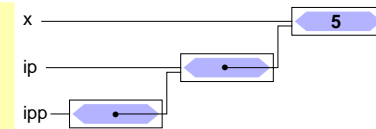


H.12 Zeiger auf Zeiger

- ein Zeiger kann auf eine Variable verweisen, die ihrerseits ein Zeiger ist

```
int x = 5;
int *ip = &x;

int **ipp = &ip;
/* -> **ipp = 5 */
```



- wird vor allem bei der Parameterübergabe an Funktionen benötigt, wenn ein Zeiger "call bei reference" übergeben werden muß (z. B. swap-Funktion für Zeiger)

2 Zugriff H.11 Argumente aus der Kommandozeile

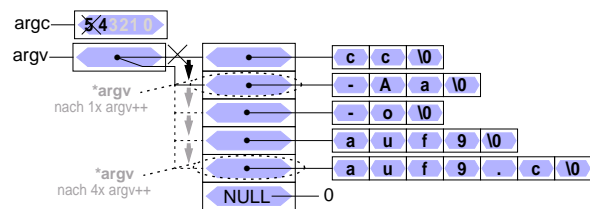
Beispiel: Ausgeben aller Argumente (2)

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```
int
main (int argc, char **argv)
{
    while (--argc > 0) {
        *argv++;
        printf("%s%c", *argv, (argc>1) ? ' ':'\n' );
    }
    ...
}
```

2. Version

linksseitiger Operator: erst dekrementieren, dann while-Bedingung prüfen -> Schleife läuft für argc=4,3,2,1



H.13 sizeof-Operator

- In manchen Fällen ist es notwendig, die Größe (in Byte) einer Variablen oder Struktur zu ermitteln
 - z. B. zum Anfordern von Speicher für ein Feld (→ malloc)

- Syntax:

```
sizeof x liefert die Größe des Objekts x in Bytes
sizeof (Typ) liefert die Größe eines Objekts vom Typ Typ in Bytes
```

- Das Ergebnis ist vom Typ `size_t` (\equiv `int`) (`#include <stddef.h>`!)

- Beispiel:

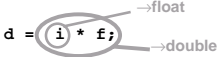
```
int a; size_t b;
b = sizeof a; /* => b = 2 oder b = 4 */
b = sizeof(double) /* => b = 8 */
```

H.14 Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck (vgl. Abschnitt D.11)

Beispiel:

```
int i = 5;
float f = 0.2;
double d;
```

 $d = (i * f);$
 

- In manchen Fällen wird eine explizite Typumwandlung benötigt (vor allem zur Umwandlung von Zeigern)

◆ Syntax:

(Typ) Variable

Beispiele:

```
(int) a    (int *) a
(float) b  (char *) a
```

◆ Beispiel:

```
char *malloc(int); /* Funktion zum dynamischen Anfordern von Speicher */
int *array; /* Zeiger auf Anfang eines dyn. anzufordernden Feldes */
int i, n = 20;

array = (int *)malloc(n * sizeof(int)); /* Feld mit n Integer-Werten */
/* dynamisch anfordern */
for (i=0; i<n; i++) array[i] = 1; /* alle Feld-Werte auf 1 setzen */
```

C-Ing

Einführung in die Programmierung für Ingenieure — C
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

H-Zeiger.doc 1998-07-22 14.22

H.33

Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

H.15 Zeiger auf Strukturen (2)

- Zugriff auf Strukturkomponenten über einen Zeiger

- Bekannte Vorgehensweise

- *-Operator liefert die Struktur
- .-Operator zum Zugriff auf Komponente
- Operatorvorrang beachten

↳ `(*pstud).best = 'n';` unleserlich!

- Syntaktische Verschönerung

↳ `->-Operator`

```
pstud->best = 'n';
```

C-Ing

Einführung in die Programmierung für Ingenieure — C
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

H-Zeiger.doc 1998-07-22 14.22

H.35

Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

H.15 Zeiger auf Strukturen

- Konzept analog zu "Zeiger auf Variablen"

- Adresse einer Struktur mit &-Operator zu bestimmen
- Name eines Feldes von Strukturen = Zeiger auf erste Struktur im Feld
- Zeigerarithmetik berücksichtigt Strukturgröße

- Beispiele

```
struct student stud1;
struct student gruppe8[35];
struct student *pstud;
pstud = &stud1; /* => pstud -> stud1 */
pstud = gruppe8; /* => pstud -> gruppe8[0] */
pstud++; /* => pstud -> gruppe8[1] */
pstud += 12; /* => pstud -> gruppe8[13] */
```

- Besondere Bedeutung zum Aufbau

- ↳ rekursiver Strukturen

C-Ing

Einführung in die Programmierung für Ingenieure — C
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

H-Zeiger.doc 1998-07-22 14.22

H.34

Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

H.16 Rekursive Strukturen

- Strukturen in Strukturen sind erlaubt — aber

- ◆ die Größe einer Struktur muß vom Compiler ausgerechnet werden können

- Problem: eine Struktur enthält sich selbst

```
struct liste {
    struct student stud;
    struct liste rest;
};
```

 falsch!

- ◆ die Größe eines Zeigers ist bekannt (meist 4 Byte)

- eine Struktur kann einen Zeiger auf eine gleichartige Struktur enthalten

```
struct liste {
    struct student stud;
    struct liste *rest;
};
```

- ↳ Programmieren rekursiver Datenstrukturen

C-Ing

Einführung in die Programmierung für Ingenieure — C
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

H-Zeiger.doc 1998-07-22 14.22

H.36

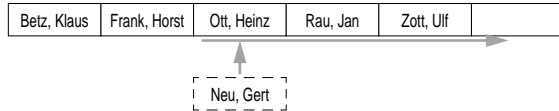
Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

H.16 Rekursive Strukturen (2)

- Problem:
 - es sollen beliebig viele Studentendaten eingelesen werden und in sortierter Form im Programm verwaltet werden

Lösung 1: Feld

- wie groß machen? — und was, wenn es nicht reicht?
- Einsortieren = richtige Position suchen + Rest nach oben verschieben + eintragen



H.16 Rekursive Strukturen (4)

- Realisierung von Lösung 2 (Skizze):

```

struct eintrag {
    struct student stud;
    struct eintrag *naechster;
}
struct eintrag leer = { {"", ""}, NULL }; /* Leeres Listenelement */
struct eintrag *stud_liste; /* Zeiger auf Listen-Anfang */
struct eintrag *akt_eintrag; /* aktuell bearbeiteter Eintrag */
struct eintrag *einfuege_pos; /* Einfuegeposition */

int eintrag_lesen(struct student *);

/* erstes Listen-Element anfordern */
akt_eintrag = (struct eintrag *)malloc(sizeof (struct eintrag));
stud_liste = &leer; /* Listenanfang auf leeres Element setzen (dadurch
                    Vermeidung von Sonderbehandlung für Listenanfang) */

while (eintrag_lesen(&akt_eintrag->stud) != EOF) {
    einfuege_pos = ... /* Eintrag, hinter dem einzufügen ist irgendwie suchen */
    akt_eintrag->naechster = einfuege_pos->naechster;
    einfuege_pos->naechster = akt_eintrag;

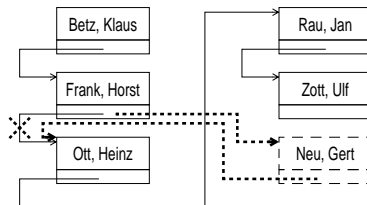
    /* neues Listen-Element anfordern */
    akt_eintrag = (struct eintrag *)malloc(sizeof (struct eintrag));
}
    
```

H.16 Rekursive Strukturen (3)

- Problem:
 - es sollen beliebig viele Studentendaten eingelesen werden und in sortierter Form im Programm verwaltet werden

Lösung 2: verkettete Liste von dynamisch angeforderten Strukturen

- Speicher für jeden Eintrag mit malloc() anfordern
- Einsortieren = richtige Position suchen + zwei Zeiger setzen



H.17 Zusammenfassung

- Variable


```
int a;
a — 5
```
- Zeiger


```
int *p = &a;
a — 5
p — • —> a
```
- Feld


```
int a[3];
a ≡ [•] [•] [•]
```
- Feld von Zeigern


```
int *p[3];
p ≡ [•] [•] [•]
      [•] [•] [•]
      [•] [•] [•]
```
- Struktur


```
struct s {int a; char c;};
struct s s1 = {2, 'a'};
s1 — [2]
      [a]
```
- Zeiger auf Struktur


```
struct s *sp = &s1;
s1 — [2]
      [a]
sp — • —> s1
```
- Feld von Strukturen


```
sa ≡ [•] [•] [•]
      [•] [•] [•]
      [•] [•] [•]
struct s sa[3];
```