C-Preprozessor

Kurzüberblick

- ❏ bevor eine C-Quelle dem C-Compiler übergeben wird, wird sie durch einen Makro-Preprozessor bearbeitet
- ❏ Anweisungen an den Preprozessor werden durch ein **`#`**-Zeichen am Anfang der Zeile gekennzeichnet
- ❏ die Syntax von Preprozessoranweisungen ist unabhängig vom Rest der Sprache
- ❏ Preprozessoranweisungen werden nicht durch **`;`** abgeschlossen!
- ❏ wichtigste Funktionen:

```
#define     Definition von Makros
#include    Einfügen von anderen Dateien
```


Makrodefinitionen

- ❏ Makros ermöglichen einfache textuelle Ersetzungen (parametrierbare Makros werden später behandelt)
- ❏ ein Makro wird durch die **`#define`**–Anweisung definiert
- ❏ Syntax:

  `#define` ***Makroname Ersatztext***

- ❏ eine Makrodefinition bewirkt, daß der Preprozessor im nachfolgenden Text der C-Quelle alle Vorkommen von ***Makroname*** durch ***Ersatztext*** ersetzt

Einfügen von Dateien

- ❏ durch die **`#include`**–Anweisung kann der Inhalt einer anderen Datei in eine C-Quelldatei eingefügt werden
- ❏ Syntax:

  `#include` **< Dateiname >**
  oder
  `#include` **"Dateiname "**

- ❏ mit **`#include`** werden *Header*-Dateien mit Daten, die für mehrere Quelldateien benötigt werden einkopiert
  - ➥ Deklaration von Funktionen, Strukturen, externen Variablen
  - ➥ Definition von Makros
- ❏ wird **Dateiname** durch **`<>`** geklammert, wird eine **Standard-Header-Datei** einkopiert
- ❏ wird **Dateiname** durch **`""`** geklammert, wird eine Header-Datei des Benutzers einkopiert (vereinfacht dargestellt!)

Zeiger(-Variablen)

- ❏ eine Zeigervariable ***(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
- ❏ Syntax:

  Typ `*`Name `;`

- ❏ Beispiel:

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

Adresse	Speicher
x: 1000	5
ip: 1004	1000
y: 1008	5

Adreßoperatoren

Adreßoperator `&`

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

Verweisoperator `*`

`*x` der unäre Verweisoperator `*` ermöglicht den Zugriff auf die Variable (das Objekt), auf die der Zeiger `x` verweist

Der Operator `[]`

`x[y]` `x` ist ein Basis-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

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*

… Zeiger als Funktionsargumente (2)

❑ Beispiel:

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

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

        tmp = *px;
        *px = *py;
        *py = tmp;
}
```


Arithmetik mit Adressen

❑ Folgende arithmetische Operatoren können zur Adreßarithmetik verwendet werden:

+, – Addition und Subtraktion von Zeigern und ganzzahligen Werten. Dabei wird immer die Größe des Objekttyps berücksichtigt.

– Subtraktion von zwei Zeigern liefert die Zahl der Objekte, die zwischen den Zeigern liegen. Nur sinnvoll, wenn beide Zeiger in den gleichen, zusammenhängenden Objektbereich verweisen.

++, -- Inkrement = nächstes Objekt
Dekrement = vorhergehendes Objekt

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

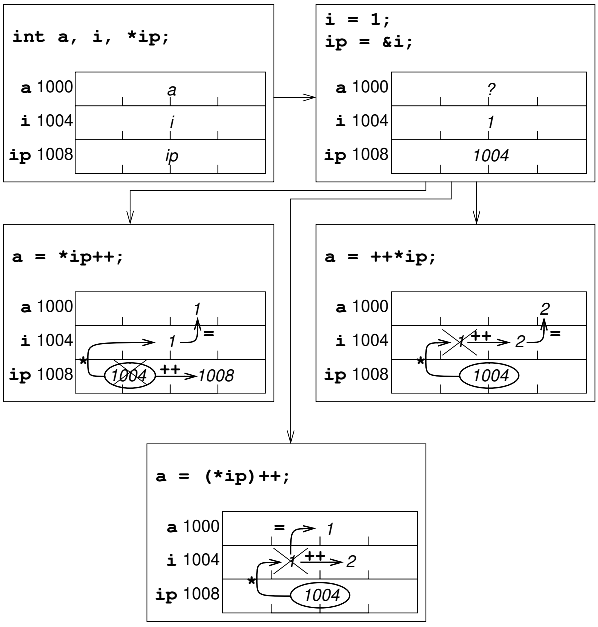
Vorrangregeln bei Operatoren

Table 3:

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

… Arithmetik mit Adressen (2)

❏ Beispiele



Vergleichsoperatoren und Adressen

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

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

Zeiger und Felder

❏ ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes

❏ im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden

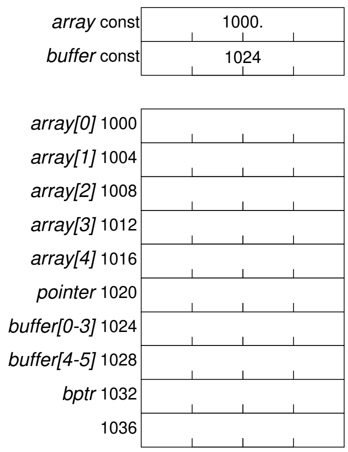
❏ es gilt:

```
int array[10];     /* → array = &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;
```


… Zeiger und Felder (2)

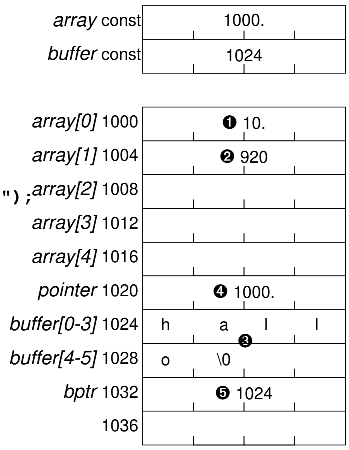
```
int array[5];
int *pointer;
char buffer[6];
char *bptr;
```



… Zeiger und Felder (3)

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

❶ array[0] = 10;
❷ array[1] = 920;
❸ strcpy(buffer,"hallo");
❹ pointer = array;
❺ bptr = buffer;
```



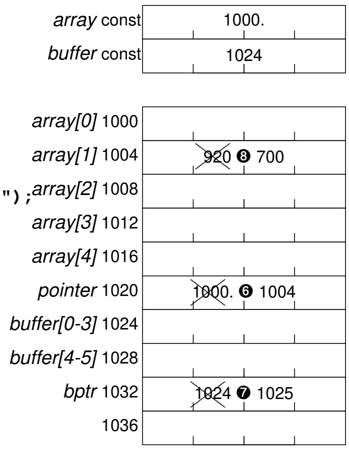
… Zeiger und Felder (4)

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

❶ array[0] = 10;
❷ array[1] = 920;
❸ strcpy(buffer,"hallo");
❹ pointer = array;
❺ bptr = buffer;

❻ pointer++;
❼ bptr++;
❽ *pointer = 700;

array++;
```



Felder als Funktionsparameter

Eindimensionale 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 stellt einen konstanten Adreßverweis dar, der als Argument eines Funktionsaufrufs *by-value* übergeben wird

❏ diese Kopie kann von der aufgerufenen Funktion – unabhängig von der Deklaration des formalen Parameters – wie eine Zeigervariable verwendet werden

❏ die beiden Deklarationen
int p2[] und **int *p2**
sind vollkommen äquivalent!

Mehrdimensionale Felder als Funktionsparameter

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

```
int a, b;
int wuerfel[10][10][10];
func(a, wuerfel, b);
…
int func(int p1, int p2[10][10][10], int p3);
oder:
int func(int p1, int p2[][10][10], int p3);
oder:
int func(int p1, int (*p2)[10][10], int p3);
```

❏ Bei der Deklaration eines mehrdimensionalen Feldes als formaler Parameter müssen die Dimensionen mit angegeben werden.
Lediglich die erste Dimension kann weggelassen werden.

❏ Funktionen können damit nur Felder gleicher fester Dimension verarbeiten.
(Der Programmierer kann jedoch die für den Zugriff auf Feldelemente erforderliche Adreßrechnung auch selbst vornehmen).

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

```
/* 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);
}

/* 3. Version */
int strlen(char *s)
{
        char *p = s;
        while (*(p++))
                ;
        return(p-s-1);
}
```


… 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

```
char amessage[] = "now is the time";
```

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

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

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

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

```
pmessage = amessage;
```

  weist dem Zeiger **pmessage** lediglich die Adresse der Zeichenkette **"now is the time"** zu

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

… Zeiger, Felder und Zeichenketten (3)

- ❏ 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++ )
                ;
}
```

- ❏ in ANSI-C können Zeichenketten in nicht-modifizierbaren Speicherbereichen angelegt werden

  ➥ Schreiben in Zeichenketten (Zuweisungen über dereferenzierte Zeiger) kann zu Programmabstürzen führen!
  strcpy("zu ueberschreiben", "reinschreiben");
  kann damit einen Segmentierungsfehler bewirken

Beispiel: dynamische Speicherverwaltung

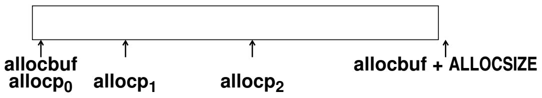
- ❏ die folgenden zwei Funktionen verwalten einen globalen, zusammenhängenden Speicherbereich,
  - aus dem Anforderungen von freiem Speicher befriedigt werden ***(alloc)***
  - und in den freigegebener Speicher wieder integriert wird ***(free)***
- ❏ diese Version einer dynamischen Speicherverwaltung 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
- ❏ Globale Definitionen

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

… Beispiel: dynamische Speicherverwaltung (2)

- ❑ Anfordern von Speicher

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



- ❑ Freigabe von Speicher

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


Felder von Zeigern

- ❑ Auch von Zeigern können Felder gebildet werden

- ❑ Deklaration - Beispiel:

```
int i;
int *pfeld[20];
```

- ❑ Zugriffe auf einen Zeiger des Feldes - Beipiel:

```
pfeld[5] = &i;
```

- ❑ Zugriffe auf das Objekt,
  auf das ein Zeiger des Feldes verweist - Beispiel:

```
i = *pfeld[7];
```

- ❑ Beispiel: Definition und Initialisierung eines Zeigerfeldes:

```
char *month_name(int n)
{
      static char *Monatsnamen[] = {
            "Unzulässiger Monat",
            "Januar",
      ...
            "Dezember"
      };

      return ( (n<0 || n>12) ?
            Monatsnamen[0] : Monatsnamen[n]
);
}
```


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)

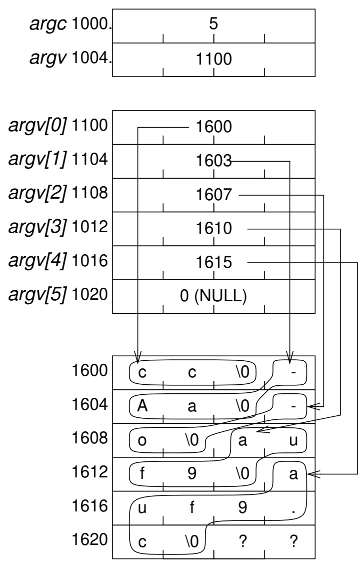
- ❑ der Kommandoname wird als erstes Argument übergeben (`argv[0]`)

… Argumente aus der Kommandozeile (2)

Kommando: `cc -Aa -o auf9 auf9.c`

Datei cc.c:

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



… Argumente aus der Kommandozeile (3)

Beispiel: Ausgeben aller Argumente

❏ die folgenden Versionen zeigen einige Anweisungen in **`main( )`**, die alle Argumente der Kommandozeile ausgeben (außer dem Kommandonamen)

1. Version

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

2. Version

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

3. Version

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


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;
```

Adresse	Speicher
x: 1000	5
ip: 1004	1000
ipp: 1008	1004

`void`–Zeiger (ANSI)

❏ Zeiger auf Daten, deren Typ nicht bekannt ist, werden als **`void`**–Zeiger definiert

❏ an **`void`**–Zeiger können beliebige Zeiger ohne **`cast`**–Operation zugewiesen werden

➥ Beispiel:

```
int a;
double b;
void *p;

p = &a;
p = &b;
```

❏ wird insbesondere bei der Übergabe von Parametern an Funktionen in Zusammenhang mit Funktionszeigern (siehe dort) benötigt

❏ K&R–C kennt keine **`void`**–Zeiger — es werden statt dessen **`char`**–Zeiger verwendet

– direkte Zuweisung anderer Zeigertypen an **`char`**–Zeiger führen zu Fehlermeldungen oder Warnungen

Zeiger auf Funktionen

❏ Ein Funktionsname ist ein konstanter Zeiger auf die Funktion

❏ dieser Wert kann als Argument an eine Funktion übergeben werden oder in einer Zeigervariablen gespeichert werden

❏ die Adresse einer Funktion ist ihr ***entry point***, an dem die Ausführung beginnt

❏ Definition eines Funktionszeigers
Syntax:

```
Typname (*Funktionszeiger) ( );
```

❏ Funktionszeiger als Funktionsparameter
Syntax der Funktionsdefinition: (ANSI–C)

```
Typname Funktionsname (…, Typ (*Parameterfunktion)(), …)
```

❏ Aufruf über Funktionszeiger
Syntax:

```
Typname (* Funktionszeiger) ( );
...
[Ergebnis =] (*Funktionszeiger) ( … );
oder einfach (nur ANSI-C!):
[Ergebnis =] Funktionszeiger( … );
```

❑ Beispiel: Shell Sort

```
int main(void)
{
   int buffer[100];
   int n;

   void sort(void *el[], int nel,
            int (*cmp)(void *, void *),
            void (*exc)(void *, void *));

   int (*compare) (void *, void *);
   void (*exchange)(void *, void *);

   int numcmp (int *, int*);

   compare = numcmp;
   …
   sort(buffer, n, compare, exchange);
   …
}
```

```
void sort (void *p[], int n,
      int (* comp)(void *, void *),
      void (*exch)(void *, void *))
{
   int gap, i, j;

   for (gap = n/2; gap > 0; gap /= 2)
      for (i = gap; i < n; i++)
         for (j = i-gap; j >= 0; j-= gap) {
            if ((*comp)(p[j], p[j+gap]) <= 0)
               break;
            (*exch)(&p[j], &p[j+gap]);
         }
      }
   }
}
```


Strukturen

❑ Variablen gleichen oder unterschiedlichen Typs können zu einer **Struktur** zusammengefaßt werden

❑ Definition - **Syntax:**

```
struct [ Strukturtypname ] {
    { Komponenten-Deklaration }*
} { Strukturname }* ;
```

❑ Strukturtypname
- bezeichnet den durch die Definition erzeugten Datentyp
- dieser Typname kann in nachfolgenden Vereinbarungen benutzt werden

❑ Strukturname
- bezeichnet das erzeugte Objekt und
- ermöglicht den Zugriff auf dessen Komponenten

Deklaration einer Struktur

❑ ein Strukturtyp mit einem bestimmten Namen **(Strukturtypname)** wird vereinbart und kann in anderen Vereinbarungen (Deklarationen und Definitionen) verwendet werden

❑ **Syntax:**

```
struct Strukturtypname {
     Komponenten-Deklaration(en)
} ;
```

Definition einer Struktur

❑ ein Struktur-Objekt des angegebenen Typs wird mit dem Namen **Strukturname** erzeugt

❑ Syntax:

```
struct {
     Komponenten-Deklaration(en)
} Strukturname(n) ;

oder:
struct Strukturtypname Strukturname(n) ;
```


Beispiele

❑ Deklaration:

```
struct datum {
     char name[12];
     int tag;
     int monat;
     int jahr;
};
```

❑ Definition:

```
struct datum meindatum;
```

❑ Deklaration und Definition:

```
struct datum {
     char name[12];
     int tag;
     int monat;
     int jahr;
} meindatum, deindatum;
```

Zugriffsoperatoren

Der `.` –Operator

x.y Zugriff auf die Komponente **y** des Struktur-Objekts **x**

Der `–>` –Operator

x–>y Zugriff auf die Komponente **y** über den Strukur-Zeiger **x**

Zugriff auf Strukturkomponenten

❑ Syntax:

Strukturname . Komponentenname

Strukturzeiger–>Komponentenname

Initialisierung von Strukturen:

❑ Syntax:

struct Strukturtypname Strukturname =
{ Initialisierungsliste } ;

❑ Beispiel:

```
struct datum {
      char name[12];
      int tag;
      int monat;
      int jahr;
} meindatum = { "Mai", 14, 5, 1917 };

struct datum *dp = &meindatum;
dp->tag = 15;     /* → meindatum.tag == 15 */
oder: (*dp).tag = 15;

meindatum.jahr = 1937;
```

Felder von Strukturen

❑ Syntax:

struct Strukturtypname {
Komponenten-Deklaration(en)
} Strukturname [Konstante] ;

❑ Beispiel:

```
struct datum geburtstage[9];
geburtstage[8].tag = 19;
…
```


© Jürgen Kleinöder Universität Erlangen Nürnberg, IMMD IV, 1996
Reproduktion jeder Art oder Verwendung dieser Unterlagen zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig
U1-C.doc 08.11.96 17:24

Zeiger und Strukturen

❑ eine Komponente einer Struktur kann ihrerseits eine Struktur sein (geschachtelte Struktur)

❑ eine direkte Rekursion bei der Vereinbarung von Strukturen ist nicht möglich

❑ eine Komponente einer Struktur kann aber ein Zeiger auf die Struktur selbst sein (**rekursive Strukturen**)

❑ rekursive Strukturen ermöglichen die Realisierung dynamischer Datenstrukturen, z. B. lineare Listen oder Bäume

❑ Beispiel:

```
struct node {
    char *word;
    int count;
    struct node *left;
    struct node *right;
}

struct node root, lnode, rnode;
root.left = &lnode;
root.right = &rnode;
```


© Jürgen Kleinöder Universität Erlangen Nürnberg, IMMD IV, 1996
Reproduktion jeder Art oder Verwendung dieser Unterlagen zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig
U1-C.doc 08.11.96 17:24

Compilerabhängige Eigenschaften von Strukturen

❑ Zuweisungen

- ANSI-C– und neuere K&R-C–Compiler erlauben Zuweisungen von ganzen Strukturen:

```
struct datum mein_geburtstag, sein_geburtstag;
sein_geburtstag = mein_geburtstag;
```

- erlaubt der Compiler diese Art von Zuweisung nicht, müssen die Elemente einzeln kopiert werden:

```
deindatum.tag = meindatum.tag;
deindatum.monat = meindatum.monat;
deindatum.jahr = meindatum.jahr;
for (i=0; i<12; i++)
    deindatum.name[i] = meindatum.name[i];
```

❑ Namen der Strukturelemente — Konflikte

- sehr alte K&R–C-Compiler unterscheiden Namen von Strukturelementen unterschiedlicher Strukturen nicht
- die Deklaration von:

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

führt dann zu einem **Compilerfehler**:
a,b declared twice

© Jürgen Kleinöder Universität Erlangen Nürnberg, IMMD IV, 1996
Reproduktion jeder Art oder Verwendung dieser Unterlagen zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig
U1-C.doc 08.11.96 17:24