dbis
Institut für Informatik
Humboldt-Universität zu Berlin

Die Programmiersprache C

3. Zeiger, Adressarithmetik

Vorlesung des Grundstudiums
Prof. Johann-Christoph Freytag, Ph.D.
Institut für Informatik, Humboldt-Universität zu Berlin
SoSe 2018

Ein legales C-Programm?

```
main(t,_,a)
char *a;
{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86, 0, a+1 )+a)):1,t<_?main(t+1, _, a ):3,main ( -94, -27+t, a
)&&t == 2 ?_<13 ?main ( 2, _+1, "%s %d %d\n" ):9:16:t<0?t<-72?main(_,
t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+\
,/+#n+,/#;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/\
+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# ){n\
l]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#\
n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
:t<-50?_==*a ?putchar(a[31]):main(-65,_,a+1):main((*a == '/')+t,_,a\
+1 ):0<t?main ( 2, 2 , "%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc \
i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

Union-Typen

- ein Union-Typ ist eine Struktur,
  von dem eine Variable (zu verschiedenen Zeiten) Werte
  unterschiedlichen Typs und Größe speichern kann

- jeder Komponente des Typs wird derselbe Speicherplatz zugeordnet

- Speicherplatzgröße wird durch die »längste« Komponente bestimmt

Union-Typen (Forts.)

Beispiel:

```
union operand {     /* Struktur,                                  */
                    /* die Bitmuster als Werte besitzt, die als   */
                    /* integer-, long integer oder double         */
                    /* interpretiert werden können                */
    int     i;
    long    l;
    double  d;
} o;
```

deklariert eine union-Struktur mit Namen *operand* und eine Instanz namens *o*

- auf Elemente kann wie folgt zugegriffen werden:

  ```
  printf("%ld \n", o.l);
  ```

- die jeweils aktuelle korrekte Interpretation des Speicherplatzes muss in einem separaten Diskriminator [zumeist von enum-Typ] explizit verwaltet werden

Union-Typ- Beispiel

```
enum optype=
        {INT, LONG, FLOAT};

struct operand {
    enum optype mytype; /* Diskrim.*/
    union variante {
            int i;
            float f;
            long d;
    } value;
} op;
```

```
... // Initialisierung von op
...
switch (op.mytype) {
    case INT: printf("%d", op.value.i);
                break;
    case FLOAT: printf("%f", op.value.f);
                break;
    case LONG: printf("%ld", op.value.l);
                break;
    default: printf ("%s", "ERROR\n");
                break;
}
```

Längenoperator

Operator **sizeof**

- ...liefert
  Speicherplatzgröße (Byteanzahl) als Wert vom Typ *size_t*
  (zumeist unsigned long)

- ...gibt es in zwei syntaktischen Formen

  (1) sizeof (<ausdruck>), muss Konstantenausdruck sein
  Wert wird zur Compilierungszeit berechnet, Klammern dürfen fehlen

```
int a[10], n;
n= sizeof(a);   /* n= sizeof a; */
```

Empfehlung: immer Klammern!

  (2) sizeof (<typ_spezifikation>) wird auch zur Compilierungszeit berechnet
  Klammern dürfen NICHT fehlen

```
...= sizeof(operand);
```

Längenoperator (2)

- Achtung!

```
#include <stdio.h>
#include <stdlib.h>
const char arr[] = "hello";
const char *cp = arr;
main(){
   printf("Size of arr %lu\n", (unsigned long) sizeof(arr));
   printf("Size of *cp %lu\n", (unsigned long) sizeof(*cp));
   exit(EXIT_SUCCESS); }
```

- Erster printf-Befehlt druckt Größe des Arrays
- Zweiter printf-Befehlt druckt 1 (warum??)

Typkonvertierung (Type Casting)

- Typkonvertierung »Überführen« eines Wertes von einem Typ in einen anderen

```
int integernumber;
float floatnumber= 9.87;
integernumber=(int)floatnumber;      // Abschneiden ab Dezimalpunkt
```

- weiteres Beispiel:

```
int integernumber= 10;
float floatnumber;
floatnumber=(float)integernumber;
```

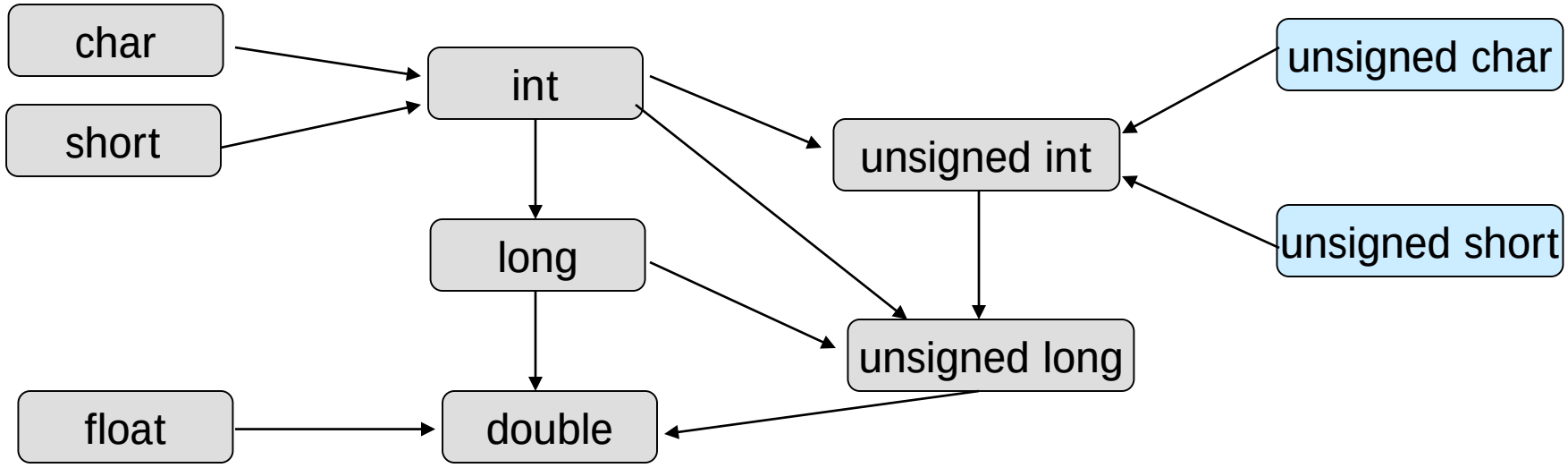
- Typkonvertierung kann auf jeden einfachen Datentyp angewandt werden, einschließlich char:

```
int integernumber; char letter='A';
integernumber=(int)letter;
```

weist den Wert 65 (ASCII-Code für 'A') der Variablen integernumber zu

Achtung: manchmal (z.B. in allen oben gezeigten Fällen ☹) führt C Typkonvertierung automatisch aus!

Konvertierung in arithm. Ausdrücken



automatische Typanpassung in Ausdrücken
(z.B. 2-stelliger Operator:

Operanden werden immer in Richtung des umfassenderen Typs konvertiert)

Typkonvertierung (Forts.)

- Cast-Operationen (explizite und implizite) können immer Quelle von Portabilitätsproblemen sein !!!

- Beispiel: Division wie erwartet...

```
floatnumber =  (float) internumber / (float) anotherint;
```

stellt sicher, dass eine »floating point« Division ausgeführt wird

Bem: es gibt auch sinnvolle Casts

Bezugsrahmen von Variablen

jede Variable hat ihren speziellen Bezugsrahmen:

- genau ein File: global
- mehrere Files (in nur einem File definiert und in anderen als „**extern**“ deklariert): programmglobal
- Block (speziell Funktionsblock): lokal

Lokale Variablen

dbis
INSTITUT FÜR INFORMATIK
HUMBOLDT-UNIVERSITÄT ZU BERLIN

Speicherverwaltung für lokale Funktionsvariablen

- Speicherplatzreservierung für lokale Fkts.variablen erfolgt zum Zeitpunkt des Funktionsaufrufs im Speicher auf dem Programm-Stack (gehört zum Aktivierungsbereich der Funktion)

- eine Initialisierung ist vom Nutzer vorzusehen,

  Achtung: eine nicht-initialisierte Variable erhält vorheriges (zufälliges) Bitmuster, d.h. der Wert ist undefiniert

- am Ende der Funktion wird Aktivierungsbereich freigegeben (Werte sind verloren)
  Achtung: eine Adresse einer lokalen Variablen niemals außerhalb des Bezugsrahmens verwenden (später)

Globale Variablen

dbis
Institut für Informatik
Humboldt-Universität zu Berlin

Speicherverwaltung für globale Variablen

- Speicherplatzreservierung für globale Variablen erfolgt zum Zeitpunkt des Ladens des (übersetzten und verbundenen) Programms,
  Compiler hat bereits die Größe eines zusammenhängenden Speicherbereiches für sämtliche globale Variablen berechnet
- der ausgefasste Speicherbereich ist NICHT immer vorinitialisiert
- falls nutzerdefinierte Initialisierungen vorgesehen sind, erfolgen diese vor Ausführung der main()-Funktion
- Speicherplatzfreigabe erfolgt mit Beendigung des Programms

Programmglobale Variablen

dbis
INSTITUT FÜR INFORMATIK
HUMBOLDT-UNIVERSITÄT ZU BERLIN

Speicherverwaltung für programmglobale Variablen

- wie bei globalen Variablen

Statische Variablen

- Bezugsrahmen von statische Variablen :
  - Funktionsblock {...} oder
  - verschachtelter Block {...{...}...}

- **Eigenschaften**
  - eine statische Variable wird zwar lokal für eine bestimmte Funktion (bzw. für den entsprechenden Bezugsrahmen) deklariert,

    wird jedoch nur ein Mal angelegt und initialisiert
    (nämlich beim ersten Aufruf der Funktion bzw. beim ersten Betreten des Bezugsrahmens)

  - beim Verlassen der Funktion bleibt der Wert der Variablen erhalten;
    beim nächsten Aufruf der Funktion (bzw. ...) hat die Variable denselben Wert wie zuvor beim Verlassen der Funktion (bzw. ...)

  - um eine statische Variable zu deklarieren, wird der Deklaration das Schlüsselwort »static« vorangestellt

Statische Variablen (2)

dbis
Institut für Informatik
Humboldt-Universität zu Berlin

Speicherverwaltung für statische Variablen

- Speicherplatzreservierung für statische Variablen erfolgt bereits zum Zeitpunkt des Ladens des Programms im globalen Datenbereich (außerhalb vom Aktivierungsbereich der Funktion)
  (Compilier hat bereits Vorkehrungen getroffen)

- ist damit global (aber nur für die Funktion/Block bzw. verschachtelte Blöcke sichtbar)

- Achtung: bei Nichtinitialisierung sind sie implizit mit Null vorinitialisiert (wie alle globalen Variablen)

Statische Variablen (3)

- Beispiel:

```
void stat(); /* Funktionsprototyp */

int main()   {
      int i;
      for (i=0; i<5; ++i) stat();
}

void stat()  {
      int auto_var = 0;
      static int static_var = 0;
      printf(" auto = %d, static = %d  \n", auto_var,  static_var);
      ++auto_var;
      ++static_var;
}
```

Statische Variablen (4)

- Ausgabe des Beispiels:

```
auto_var = 0, static_var = 0
auto_var = 0, static_var = 1
auto_var = 0, static_var = 2
auto_var = 0, static_var = 3
auto_var = 0, static_var = 4
```

- Beobachtung
  - Die Variable *auto_var* wird mit jedem Aufruf neu erzeugt
  - Die Variable *static_var* wird einmal initialisiert, dann bleibt der Wert bis zum nächsten Aufruf (trotz des Verlassens der Funktion) erhalten

Zeiger, allgemeine Bemerkungen

dbis
INSTITUT FÜR INFORMATIK
HUMBOLDT-UNIVERSITÄT ZU BERLIN

Zeiger (Pointer)

- Zeigerkonzept ist ein wesentlicher Teil der Sprache C

- In C werden ***Zeiger*** intensiv genutzt. **Warum?**:
  1. manchmal die einzige Möglichkeit (z.B. dynamische Variablen, siehe malloc/free)
  2. erzeugt kompakten und effizienten Code
  3. C kann mit mächtigen Werkzeugen kombiniert werden

- C nutzt Zeiger in Zusammenhang mit Feldern, Strukturen, Funktionen

Was ist ein Zeiger?

- Ein Zeiger ist eine Variable, die eine Adresse einer anderen Variablen (als Wert) speichert.

  Zeiger gibt es in C für Variablen beliebigen Typs

  Beispiel für die Deklaration eines Zeigers:

```
int *pointer;       /* auch möglich: int*pointer; oder int * pointer; */
int* p1, p2, p3;    /* ACHTUNG: nur ein Zeiger, zwei integer */
```

- der monadische Adressoperator „**&**“ gibt die Adresse einer Variablen zurück

- Der Dereferenzierungsoperator „*****“ gibt den „Wert eines Objektes zurück“, auf den der Zeiger zeigt

Was ist ein Zeiger? (2)

dbis
Institut für Informatik
Humboldt-Universität zu Berlin

Bemerkung:

ein Zeiger muss mit einem bestimmten Typ assoziiert werden

Begründung am Beispiel:
ein Zeiger auf ein

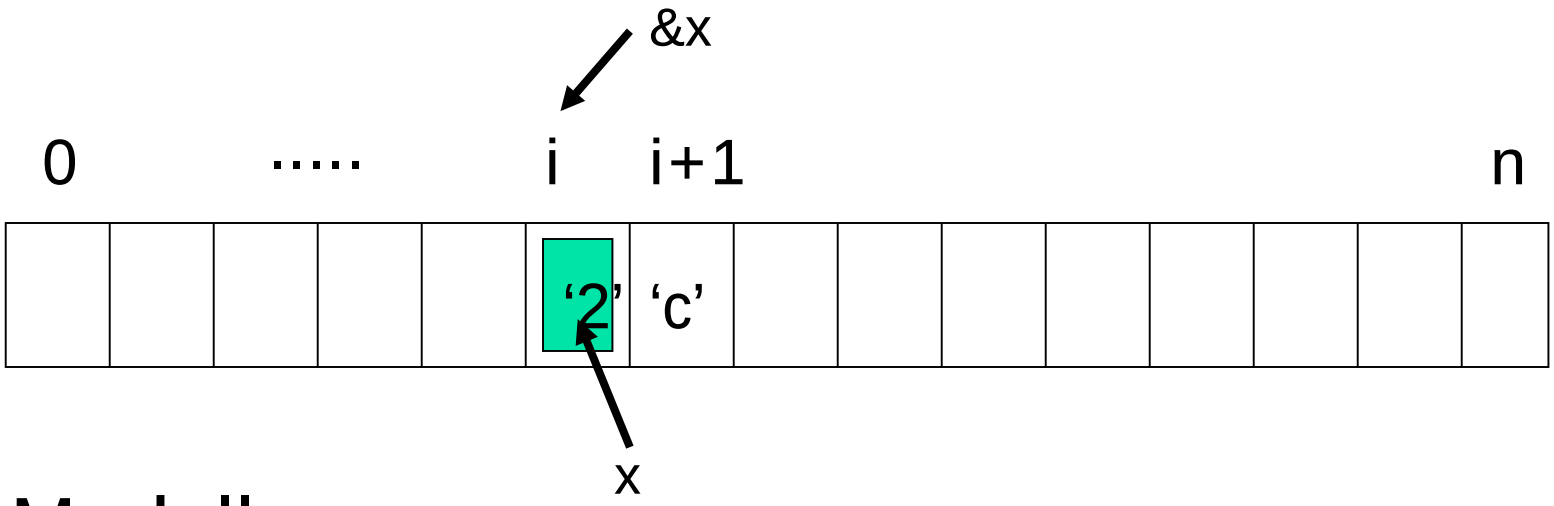
short int

kann nicht durch eine Zeigervariable auf ein

long int

verwaltet werden

Hauptspeichermodell



- Modell:
  - Speicher = lineare Sequenz von Speicherzellen
  - Adressierung von Bytes
  - Bemerkung: C-Adressierung ist typabhängig!!

Beispiel

Effekt des folgenden Programms:

```
int x = 1, y = 2; /* zwei initialisierte integer-Variablen */
int *ip;          /* nicht-initialisierte Zeigervariable für integer */
ip = &x;          /* Bildung eines Adresswertes und Zuweisung */
y = *ip;          /* Bildung eines Zeigerinhalts u. Zuweisung  an Variable*/
x = ip;           /* sollte verwarnt werden, weil nicht portabel : in C++ Fehler*/
*ip = 3;          /* Änderung eines Zeigerinhalts per Zuweisung */
```

Bemerkung

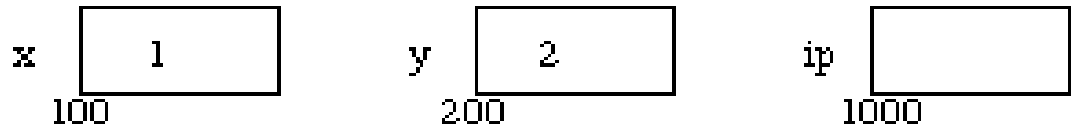
- ein Zeiger ist eine Variable
- sein Wert muss irgendwo im Speicher gespeichert werden

Annahme (s. Bild 3.23):

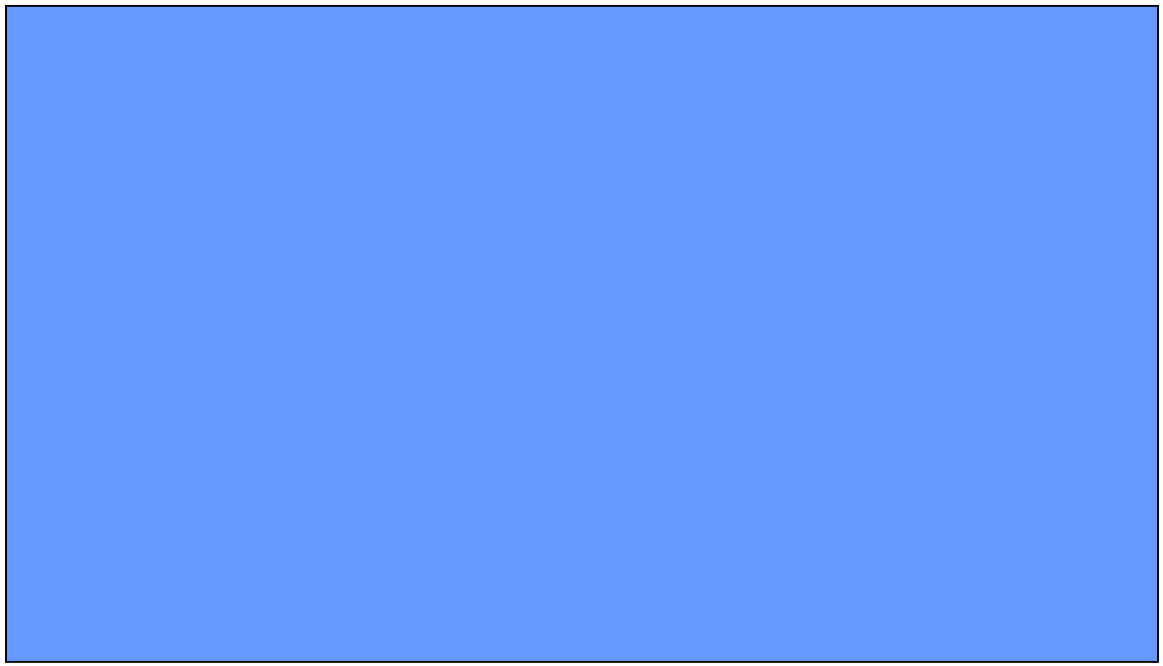
- Wert von x ist im Speicher in der Zelle 100 gespeichert
- y an Adresse 200 und ip an Adresse 1000

Beispiel (2)

```
int x = 1, y =2;
int *ip;

ip = &x;
```



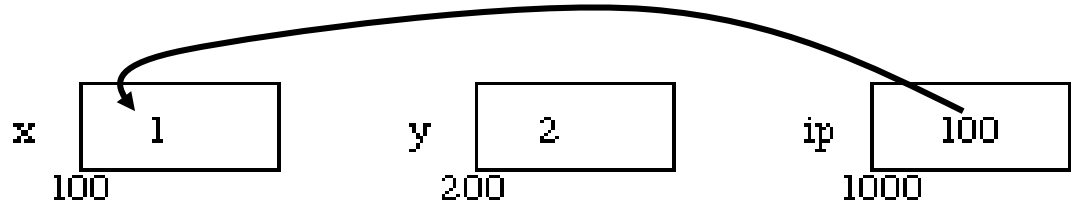

```
y = *ip;
```



```
x = ip;

*ip = 3
```

Beispiel (3)

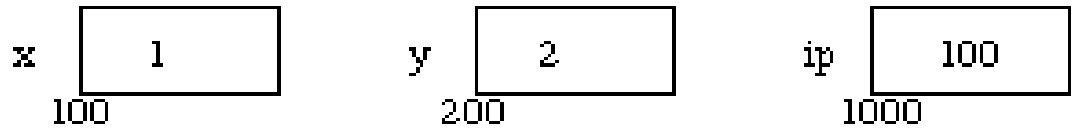
```
int x = 1, y =2;
int *ip;

ip = &x;
```



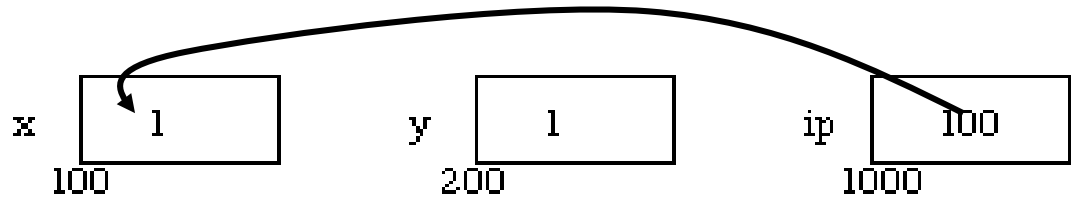

```
y = *ip;

x = ip;

*ip = 3
```

Beispiel (4)

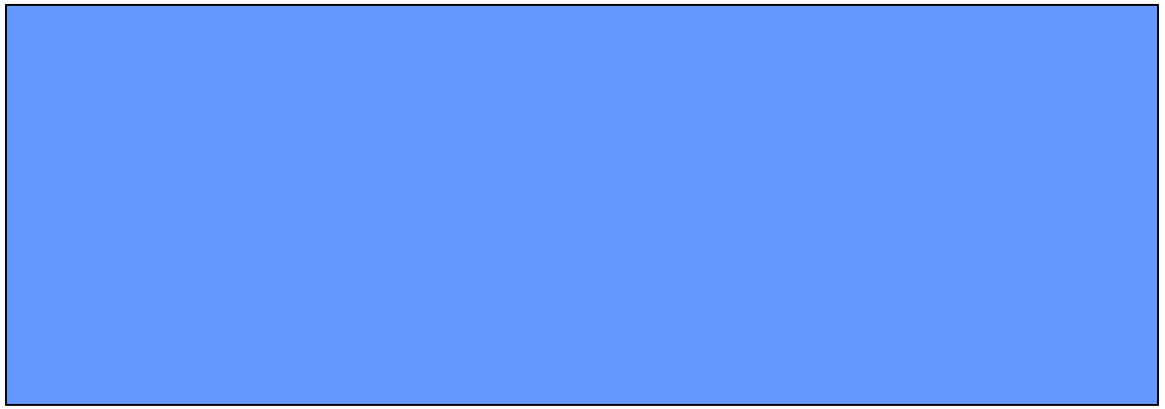
```
int x = 1, y =2;
int *ip;

ip = &x;
```




```
y = *ip;
```

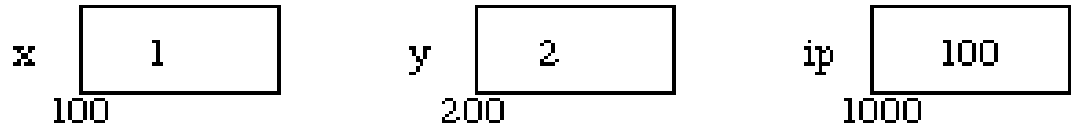



```
x = ip;

*ip = 3
```

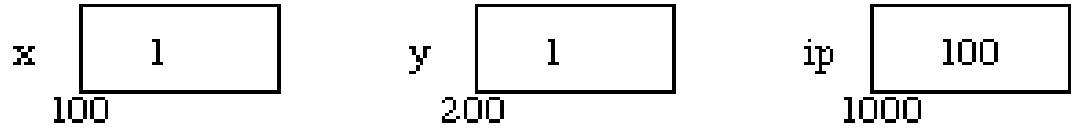


Beispiel (5)

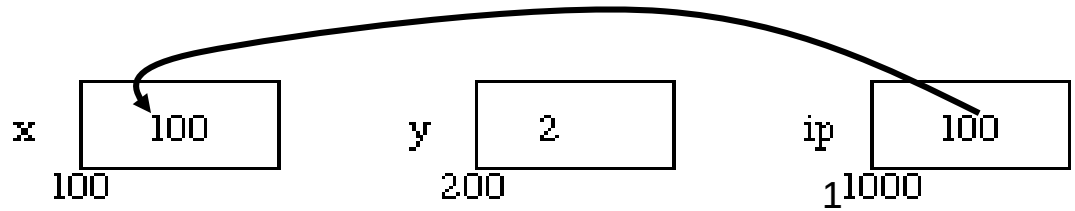
```
int x = 1, y =2;
int *ip;

ip = &x;
```



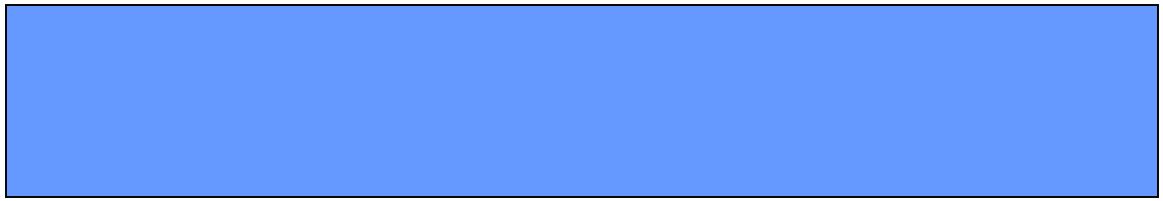

```
y = *ip;
```




```
x = ip;
```

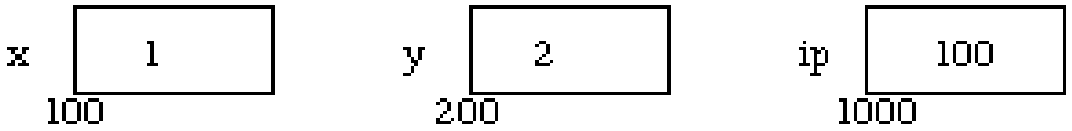



```
*ip = 3
```

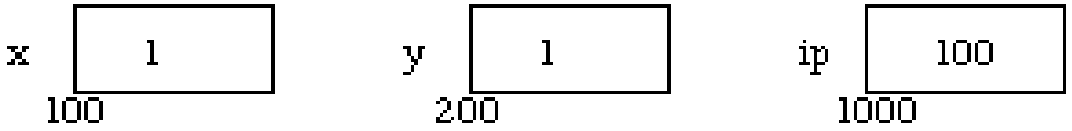


Beispiel (6)

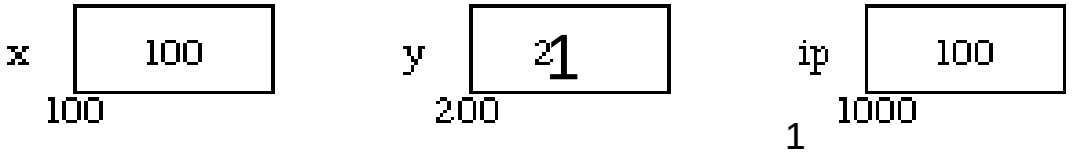
```
int x = 1, y =2;
int *ip;

ip = &x;
```



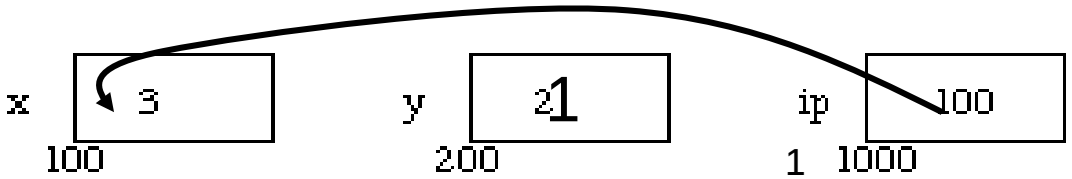

```
y = *ip;
```




```
x = ip;
```



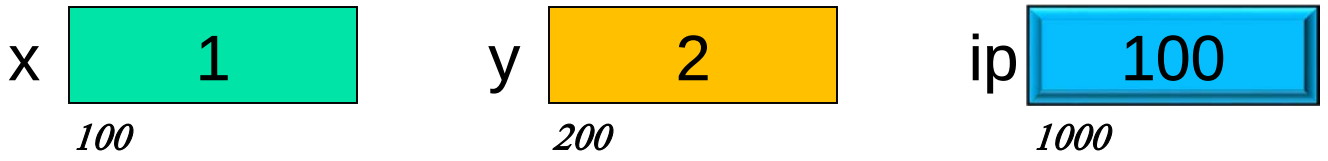

```
*ip = 3
```



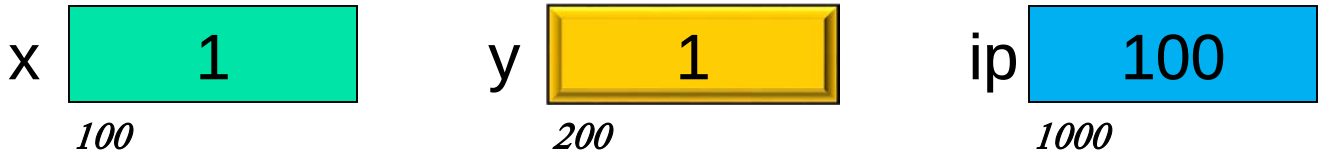
Beispiel (6)

int x = 1, y =2;int *ip;

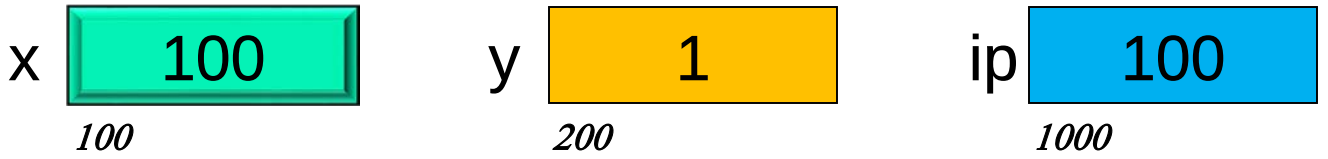
ip = &x;



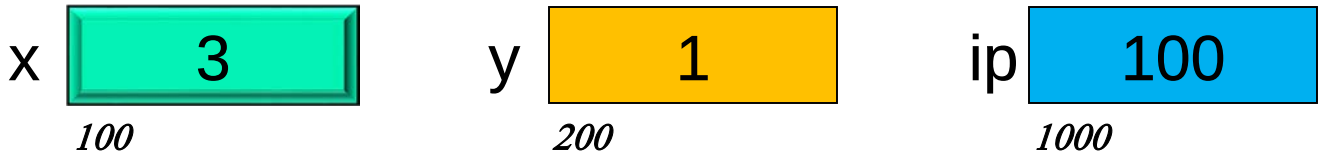
y = *ip;



x = ip



*ip = 3



Fazit

- Wichtig:
  - wird ein Zeiger deklariert, zeigt er zunächst irgendwo hin
    (nur globale Zeiger sind mit Null initialisiert)
  - Zeiger muss also initialisiert werden

- somit produziert

```
int *ip;   *ip = 100;
```

undefiniertes Verhalten (z.B. Abbruch des Programms oder unbemerktes Überschreiben des Wertes an der (zufälligen) Adresse, die in ip steht oder Formatieren der Festplatte ...).

- Korrektur:

```
int *ip; int x;
ip = &x; *ip = 100;
```

Adressarithmetik

Anwendung arithmetischer Operationen auf Zeiger:

```
int *x, *y;
*x = *x + 10;      /* Vorsicht: x nicht initialisiert */
++*x;  (*x)++;  /* Vorsicht: *x ++ ist auch gültig, aber andere Semantik!!!*/
y = x;
```

Bemerkung:

- ein Zeiger ist eine Adresse auf eine Variable bestimmten Typs;
  der Adresswert selbst ist (zumeist) als numerischer Wert realisiert
- **ein Zeiger ist kein Integer-Wert**

Grund dafür, dass ein Zeiger einen Typ besitzt:

- Definition, worauf es zeigt und wie viele Bytes mit dem Wert verbunden sind
- wird ein Zeiger erhöht,
  so wird er um eine gewisse Anzahl n von Bytes erhöht, die sich nach der Größe des entsprechenden Typs richtet

Adressarithmetik (2)

Beispiel

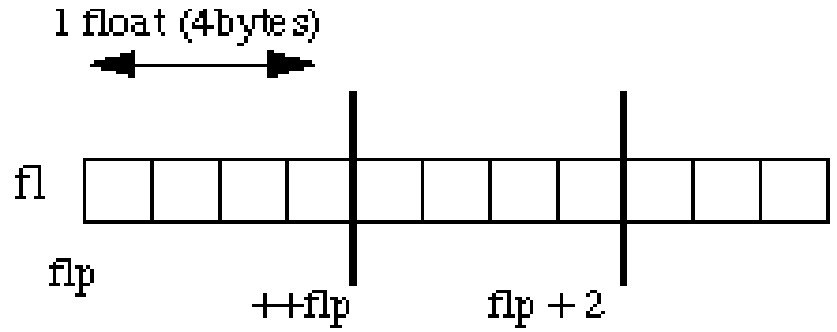
- Char-Zeiger
  ++ch_ptr; */* erhöht Adresse um sizeof(char), immer 1 */*

- für einen integer- oder float- Zeiger:
  ++i_ptr;
  ++f_ptr; */* erhöht Adresse um sizeof(float), z.B. 4 */*

Adressarithmetik (3)

Beispiel:

- gegeben:
  float Variable fl u. ein Zeiger auf eine float Variable flp



- (++flp) „verschiebt“ den Zeiger um 4 Bytes

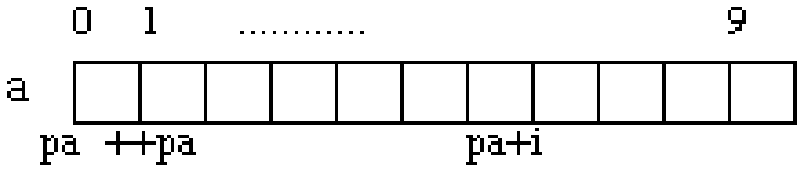
- addiert man dagegen 2 zum Zeiger,
  so bewegt sich der Zeiger um 2 float-Positionen,
  d.h. um 8 Bytes (flp wird um den Wert 8 erhöht)

Zeiger und Felder

- Zeiger und Felder sind verwandte Konzepte in C

  Beispiel:

```
int a[10], x;
int *pa;
pa = &a[0]; /* pa zeigt auf Adresse von a[0] */
x = *pa;    /* x = Inhalt von pa ( also a[0] in diesem Fall) */
```



- für Wertezugriffe im Feld werden Zeiger benutzt

  pa + i == & a[i]

- Achtung: Keine Grenzwertüberprüfung bei dem Zugriff auf das Feld

  es ist leicht, Werte anderer Speicherzellen zu überschreiben

Zeiger und Felder (2)

- legaler Ausdruck:

  Zuweisung:

  pa = a; **statt** pa = &a[0]

- äquivalente Ausdrücke:

  &a[i] ≡ a + i */* Achtung: Offset ist: i*sizeof(int) und nicht i */*

  a[i] ≡ *(a + i)

- Unterschied zwischen Zeiger und Feld:
  - ein Zeiger ist eine Variable:

    pa = a und pa++

  - ein Feld ist **keine** Variable:

    a = pa und a++ sind illegale Anweisungen

Zeiger und Felder (3)

int f[n], *pint, k; /* ein Feld, ein Zeiger, ein Integer */

f und pint sind vom selben Typ: Adressen von Integer-Werten
(Zeiger auf Integer)

jeweils dasselbe sind:

- pint= f; pint= &f[0];
- f[0]= 1; *pint= 1; /* 0. Element */
- f[1]= 1; *(pint+1)= 1; /* 1. Element */
- f[k]= 1; *(pint+k)= 1; /* k-tes Element */

Adressarithmetik statt Indexarithmetik

Bewertung

- Effektivitätsgewinn (zur Entstehungszeit von C bedeutender als heute)
- Unsicheres Sprachkonzept: keine Indexkontrolle möglich (Laufzeitfehler bei Grenzverletzung werden nicht angezeigt)
- Optimierende Compiler: ersetzen selbständig Multiplikation durch Addition

Zeiger und Felder (4)

int f[n], *pint; */* *Feld und Zeiger sind fast äquivalent* */*

Unterschied:

- pint ist Variable (Wert von pint kann geändert werden)
  z.B.: pint= f;

- f ist keine Variable, d.h. Wert (Adresse) kann nicht geändert werden (Anfangsadresse und Größe des Feldes sind unveränderlich)
  z.B.: f= pint; /* ist falsch */

Zeiger und Funktionen

- Funktionsargumente in C: “call by value”
- Ergebnisrückgabe durch Variablen: Zeiger
- Beispiel: Funktion um Wert zweier Variablen zu vertauschen:
- Der Funktionsaufruf wird nicht den gewünschten Effekt haben:

```
swap(a, b);   /* wird nicht funktionieren */
```

- Zeiger sind die Lösung:

  Übergabe der Adresse der Variablen und Zugriff auf die Werte durch Zeiger innerhalb der Funktion

Swap-Beispiel

- Funktionsaufruf von swap:

  swap(&a, &b)

- Swap Code:

```
void swap(int *px, int *py)
    {
        int temp = *px;   /* Inhalt von Zeigerwert */
        *px = *py; *py = temp;
     }
```

Zeiger und Funktionen (2)

- Achtung: Zombies

  Rückgabe von Zeigern aus Funktionen ist u.U. undefiniert, wenn Zeiger auf lokale Variablen im Spiel sind:

```
int* foo () {
        int i = 12;
  return &i;
}
/* gcc: warning: function returns address of local variable */
```

  nachdem die Funktion zurückkehrt, gibt es i nicht mehr !

Zeiger und Funktionen (3)

- Parameterübergabe:

  Bei Übergabe eines Felder an eine Funktion wird der Verweis auf das erste Element des Feldes im Speicher übergeben

  Beispiel:

  strlen(s) oder strlen(&s[0])

- Funktionsdeklaration:

  ```
  int strlen(char s[]);
  ```

  ist äquivalent zu:

  ```
  int strlen(char *s);
  ```

  da

  char s[] ≡ char *s

Felder und Funktionen (1)

strlen() ist eine Standardbibliotheksfunktion (#include <string.h>)

- gibt Länge eines Strings zurück
- Funktionsdefinition:

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

Felder und Funktionen (2)

dbis
Institut für Informatik
Humboldt-Universität zu Berlin

Weiteres Beispiel: *strcpy*:

t → s (Vor. s ist hinreichend groß, t ist ‚\0' terminiert)

```
void strcpy(char *s, char *t)
  {  while ( *s++ = *t++ ); }
```

Ausdruck liefert Wert (zugewiesener Wert), der logisch ausgewertet wird

Bemerkungen:

- *While*-Schleife hat leeren "Rumpf"
- Zuweisung erfolgt an den Inhalt der Zelle (Indirektion!!)
- Erhöhung der Variablen nach der Zuweisung
- Abbruchbedingung '\0' == 0

Felder und Funktionen (3)

dbis
Institut für Informatik
Humboldt-Universität zu Berlin

lesbarere strcpy-Variante:

```
while ( *t) {
        *s= *t;
        s++;
        t++;
}
*s= '\0';
```

Ausdruck liefert Wert (zugewiesener Wert), der logisch ausgewertet wird

Bemerkung:

- Sollte Maschine jedoch Spezialbefehl für Kopieren und Zeigererhöhung liefern, kann aus voriger Variante effizienterer Code produziert werden

Fragen