Grundlagen der Betriebssysteme

Praktische Einführung mit Virtualisierung

Stefan Bosse

Universität Koblenz - FB Informatik

C Programmierung (Teil 2)

1. Datenstrukturen
2. Aggregationen
3. Zeiger und Speichereferenzen

Datenstrukturen

C ist eine prozedurale und speicherorientierte Programmiersprache, d.h. Daten und Funktionen werden getrennt formuliert.

- Wie einzelne Variablen definiert und benutzt werden wurde bereits eingeführt
- Mit Rekords werden einzelne Variablen zu benannten Datenstrukturen zusammengefasst
- Eine Definition einer Datenstruktur ist eine Typedefinition, und es können Variablen von diesem Typ erstellt werden.

Datenstrukturen

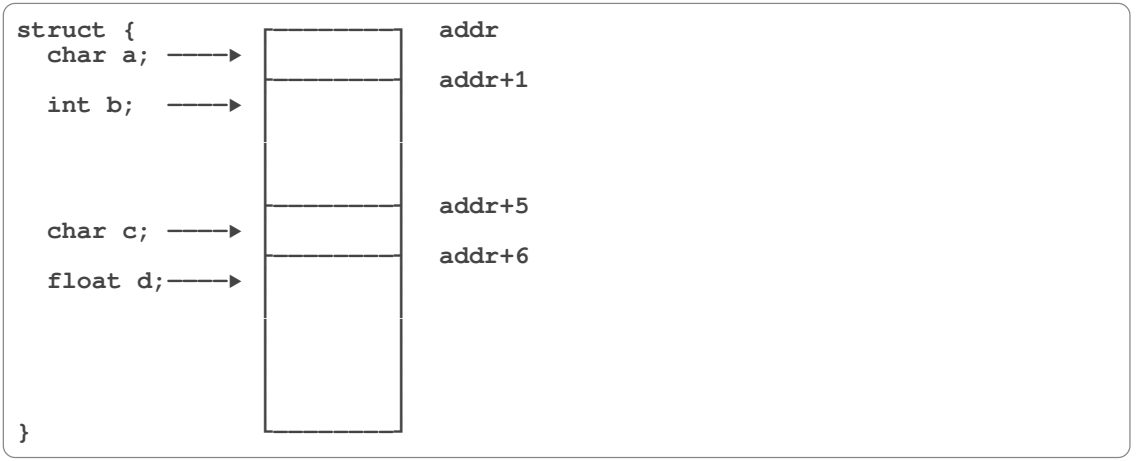
```
struct <sbezeichner> {
  <datentyp> <ebezeichner>;
  <datentyp> <ebezeichner>;
  ..
};
typedef struct <sbezeichner> <tbezeichner>;
struct <sbezeichner> ds1;
<tbezeichner> ds2;
```

Def. 1. Datenstrukturen in C. Die Deklaration eines Strukturtyps fängt immer mit dem Schlüsselwort struct an und enthält eine Liste von Elementen in geschweiften Klammern.

- *sbzeichner*: Name der Datenstrukur (Datentyp ist noch `struct <sbezeichner>`
- *ebezeichner*: Name eines Datenstrukturfelds (Element)
- *tbezeichner*: Name eines Datentyps (hier als Ersatz für `struct <sbezeichner>`)

Datenstrukturen

Die Größe des Datenebereichs im Speciher für eine Datenstruktur ist die Summe alle Datenbereiche der einzelnen Strukturelemente



Def. 2. Lineares Speichermodell einer Datenstruktur

Datenstrukturen

Folgende Operationen sind mit Strukturvariablen erlaubt:

1. Zuweisen einer Struktur an eine andere Struktur mit demselben Typ (aber nur als Speicherobjekt und Speicherkopie) mit Ausnahme von Zeigern und Funktionsargumenten
2. Übergabe von Strukturen an eine Funktion und Verwenden von Strukturen als Rückgabeparameter (Strukturen werden immer als Referenz, d.h. die Startadresse, übergeben)
3. Ermitteln der Adresse einer Struktur oder eines Elements mit dem Adressoperator `&`
4. Ermitteln der Größe einer Struktur mit dem sizeof-Operator und Speicher mit dem Alignof-Operator anordnen (seit C11)
5. Selektieren einzelner Komponenten über den Punktoperator `s.e`,sowohl LHS als auch RHS.

Datenstrukturen

Datenstrukturen

Datenstrukturen als ganzes sind immer als Pointer zu betrachten.

- Werden Datenstrukturen als Argument and Funktionen übergeben so geschieht dass immer als Speicherreferenz uns es wird keine Wertkopie erstellt!
- D.h. aber nicht das Variablen von Datenstrukturen immer als Pointer angesehen werden, wie nachfolgendes Beispiel zeigt.

```
struct s {
  int x;
  int y;
};
struct s ds1,ds2;
// falsch: memcpy(ds2,ds1,sizeof(ds1));
// richtig mit Umwandlung in einen Pointer, als
memcpy((void*)&ds2,(void *)&ds1,sizeof(ds1));
// richtig!
int foo(struct s ds) {
  return ds.x;
}
.. foo(ds1) ..
```

Bsp. 1. Verwendung von Datenstrukturen: 1. mit expliziter, 2. impliziter Pointer Wandlung.

Zeiger

Eines der wichtigsten Themen in C: Zeiger(variablen).

Zeiger werden auch Pointer genannt und stellen einen Verweis auf eine bestimmte Speicheradresse dar, in der z.B. ein Wert steht.

Ist die Zeigerarithmetik erst einmal verstanden, sind auch fortgeschrittene Themen keine so große Hürde mehr.

Zeiger

Es folgt ein kleiner Überblick über die wichtigsten Dinge, die man mit Zeigern realisieren kann:

- Speicherbereiche können dynamisch zur Laufzeit reserviert, verwaltet und wieder gelöscht werden.
- Mit Zeigern können Sie Datenobjekte per Referenz an Funktionen übergeben.
- Mit Zeigern lassen sich Funktionen als Argumente an andere Funktionen übergeben.
- Komplexe verkettete Datenstrukturen wie Listen und Bäume lassen sich ohne Zeiger gar nicht realisieren.
- Es lässt sich ein typenloser Zeiger (`void *`) definieren, mit dem Datenobjekte beliebigen Typs verarbeitet werden können, auch Funktionen!
- **Achtung: Zeiger sind bei der Definition uninitialisiert! Gefahr fehlerhafter Speicherzugriffe!**

Zeiger

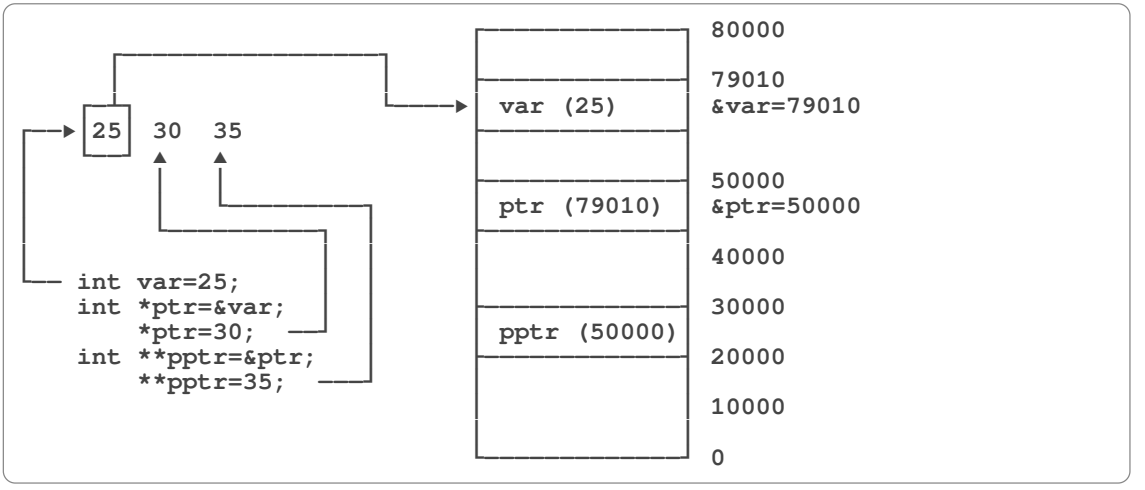


Abb. 1. Der Zusammenhang zwischen Wert- und Zeigervariablen. Das Widersprüchliche ist: Zeiger sind auch Wertvariablen, und obwohl sie mit einen ausgwiesenen Datentyp erzeugt werden sind sie doch immer vom gleichen Datentyp (z.B. long oder int)

Zeiger

Zeiger stellen lediglich die Adresse und den Typ eines Speicherobjekts dar. Die Definition eines solchen Zeigers sieht wie folgt aus:

```
<Datentyp> *<pbezeichner>;
*<pbezeichner> = ε(*<pbezeichner>; // Ausdrücke mit Pointern
<pbezeichner> = &<vbezeichner>;
```

Def. 3. Definition von Zeigervariablen von einem bestimmten Zieldatentyp, Indirektionsoperationen, und Adressierung.

Am Stern * zwischen Datentyp und Bezeichner kann man den Zeiger erkennen. Der Name ist der Bezeichner und wird als Zeiger auf einem Typ Datentyp deklariert.

Zeiger enthalten Adressen auf Speicherobjekte. Man braucht einen Dereferenzierungs- oder Indirektionsoperator * in Ausdrücken um an den Wert des referenzietren Speicherobjekts zu gelangen bzw. den Wertcontainer. Dei Adresses einer Variable (auch von Zeigern selbst!) erhält man durch den Adressoperator &.

Zeiger

Zeiger als Funktionsparameter

Mit Zeigern kann man Referenzsemantik in Funktionsaufrufen simulieren:

```
void foo(int x, int *y) {
  *y=x*2;
}
int main() {
  int x,y;
  x=2;
  foo(x,&y);
  printf("%d %d\n",x,y);
}
```

Bsp. 2. Referenzsemantik für Funktionsargumente mit Pointern

- Sowohl Wertvariablen als auch Zeigervariablen können über Zeiger (d.h. Adressenwerte) an Funktionen übergeben werden und dort verändert werden können

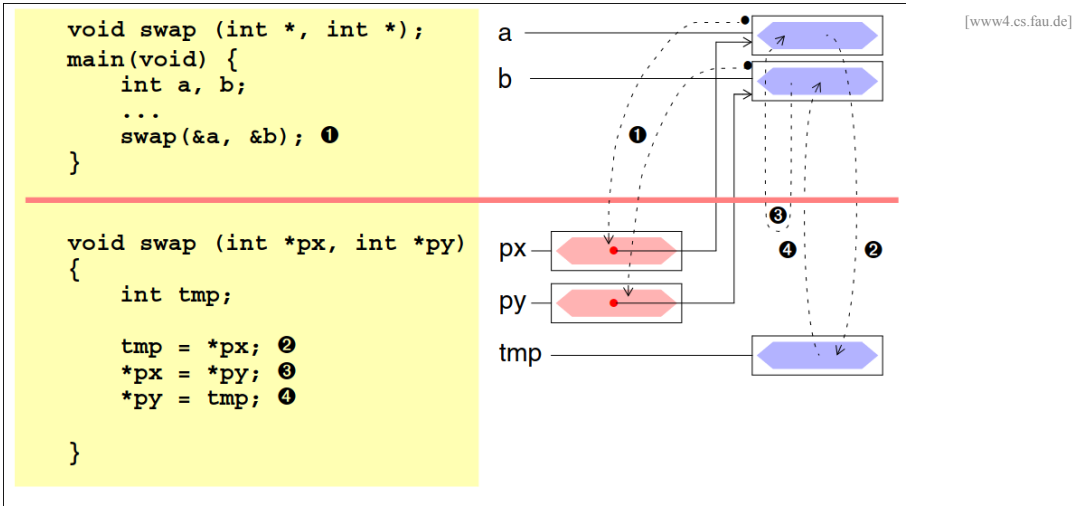


Abb. 2. Beispiel von Zeigern als Funktionsargumente

Zeiger als Funktionsparameter

Zeiger als Rückgabewert

Man kann auch Speicheradressen aus Funktionen zurückgeben (also Werte von Zeigern).

Hier muss auf das Speichersegment geachtet werden aud dem die Speicheradresse stammt. Nur Heap Adressen dürfen aus einer Funktion "hochgereicht" werden, **niemals** Stack Adressen.

Dynamische Speicherallokation

- Will man zur Laufzeit neue Speicherobjekte (vor allem Strukturen und Arrays) erzeugen, muss man dazu einen Speicherbereich auf dem Heap reserviren.
- Dazu wird die `<addr>=malloc(<numbytes>)` Funktion verwendet.

Der Speicherbereich enthält noch keine gültigen Daten, sondern kann alle mögliche (Byte)Werte enthalten. Einige *malloc* Implementierungen (Betriebssystem!) füllen den Speicherbereich mit Nullen. Kostest Laufzeit, aber verhindert Datenlecks und "Spionage", wenn der Speicherbereich zuvor von einem anderen Prozess benutzt wurde.

- Anders als Speicherobjekte die auf dem Stack reserviert wurden benötigt der Heap explizite Freigabe von nicht mehr benötigten Speicherbereichen mittels der `free(<addr>)` Funktion.
- Allokation und Freigabe müssen immer paarweise verwendet werden (wobei bei Beendigung eines Programms der gesamte belegte Speicher aber freigegeben wird).

Der NULL-Zeiger

- Den Indirektionsoperator darf man natürlich nur verwenden, wenn der Zeiger eine gültige Adresse enthält.
  - Wurde dem Zeiger keine gültige Adresse zugewiesen und wird der Indirektionsoperator trotzdem verwendet, wird das Programm vermutlich aufgrund einer Speicherzugriffsverletzung (segmentation fault) abstürzen oder im schlimmsten Fall sogar weiterlaufen und unvorhergesehene Dinge auslösen.

In der Praxis kann man viele Fehler vermeiden, wenn ein (vorerst) nicht verwendeten Zeiger zunächst mit NULL (0) belegt, also gleich bei der Definition initialisieren und einen Zeiger vor jeder Verwendung überprüfen.

- NULL ist ein Zeiger, der keine gültige Adresse verwendet, sondern meist einfach auf die Adresse 0 zeigt.
  - Globale Zeiger oder Zeiger, die mit `static` gekennzeichnet wurden, werden automatisch mit dem NULL-Zeiger initialisiert.
  - Lokale Zeiger in einem Anweisungsblock enthalten dagegen ein beliebiges und somit undefiniertes Bitmuster.

Der NULL-Zeiger

- Das Problem mit nicht oder falsch initialisierten Zeigern ist ein ungültiger Speicherzugriff.
  - In vielen Betriebssystemen wird im virtuellen Speicher die Adressse 0 (oder der erste Block, z.B. 0-511) nicht belegt. Ein Speicherzugriff auf die 0-Adresse wird als Fehler erkannt.
  - Aber eine andere ungültige Adresse (ungleich 0) wird nicht als fehlerhaft erkannt und kann zu "Spätfolgen" bei der Benutzung führen (Überschreibung von anderen Variablen usw.).

Der Funktionszeiger

Obwohl Funktionen in C keine Werte erster Ordnung sind (also nicht an Funktionen als Argumente übergeben werden können), kann man sich wieder mit Zeigern auf Funktionen behelfen.

- Die Definition und Verwendung von Funktionszeigern ist in ihrer Schreibweise etwas verwirrend.

```
int foo(int x) { return x*2 };
int (*fooPtr)(int);
int main() {
  fooPtr=&foo;
  printf("%d\n",foo(2));
  printf("%d\n",(*fooPtr)(2));
}
```

Bsp. 3. Funktionszeiger

Zeiger können alsi auf Funktionen verweisen. Sie verweisen dann auf die Anfangsadresse des Codes in der Funktion. Es sind hierbei zusätzliche Klammern nötig. Ein solcher Zeiger kann wie folgt erstellt werden:

```
<Rückgabetyp> (*funktionsZeiger)(<formale Parameter>);
```

- Ohne die Klammerungen von (*funktionsZeiger) hätte man lediglich den Prototyp einer Funktion und keine Definition eines Zeigers erstellt.
- Der Name der Funktion wird dann implizit in einen Zeiger auf diese Funktion umgewandelt.
- Aufrufen kann man die Funktion über Zeiger mit:

```
( *funktionsZeiger ) (parameter);
```

womit der Funktionszeiger explizit dereferenziert wird.

Im C2JS Transpiler werden Funktionen aber als JS Funktionen implementiert. "Speicherzeiger" auf Funktionen sind daher nicht möglich. Stattdessen werden negative Adressen als ein Funktionsindex in einer (globalen) Funktionstabelle verwendet um Funktionen referenzierbar zu machen!

```
CS={"-1":"free","-2":"malloc","-3":"memcpy","-4":"printf",
    "-5":"foo","-6":"main"}
CSI={"free":-1,"malloc":-2,"memcpy":-3,"printf":-4,
     "foo":-5,"main":-6}
DS.writeInt32(CSI["foo"],fooPtr);
printf("%d\n",eval(CS[DS.readInt32(fooPtr)])(2));
```

Bsp. 4. JS Code vom fooPtr C Code unter Verwendung von Funktionstabellen

Arrays

Arrays dienen dazu, mehrere Variablen gleichen Typs zu speichern. Bei Arrays werden hierzu die einzelnen Elemente als eine Folge von Werten eines bestimmten Typs im Speicher linear abgelegt.

Wir müssen unterscheiden:

1. Statische Definition und Allokation von Arrays ⇒ Aggregatvariablen
2. Dynamische Allokation ⇒ Zeigervariablens + malloc

Statische Definition

```
<Datentyp> <Arrayname> [ <Anzahl_der_Elemente N> ];
<Datentyp> <Arrayname> [ <N> ][ <M> ] ..;
```

Def. 4. Definition einer Aggregatvariable mit Speicherallokation. Der gesamte belegte Speicherbeich ist *N sizeof*(*elementtype*), sowie mehrdimensionale Arrays.

Speicherlayout

- Auch mehrdimensionale Arrays werden im Speicher linear abgelegt. Es gibt eine Indexreihenfolge, typischerweise:
  - Führender Index sind die Spalten (also hier der zweite Index), nachfolgend der Zeilenindex (hier der erste Index).
  - RGB Bildmatrizen weichen ab, hier ist der Farbindex führender Index, gefolgt von Spalten und dann Zeilen.

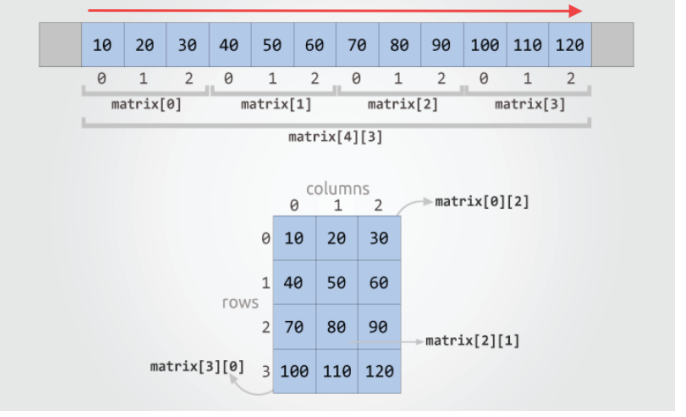


Abb. 3. Speicherlayout bei zweidimensionalen Matrizen

```
<Datentyp> <Arrayname> [ <Anzahl_der_Matrizen Z> ]
                       [ <Anzahl_der_Zeilen Y> ]
                       [ <Anzahl_der_Spalten X> ];
```

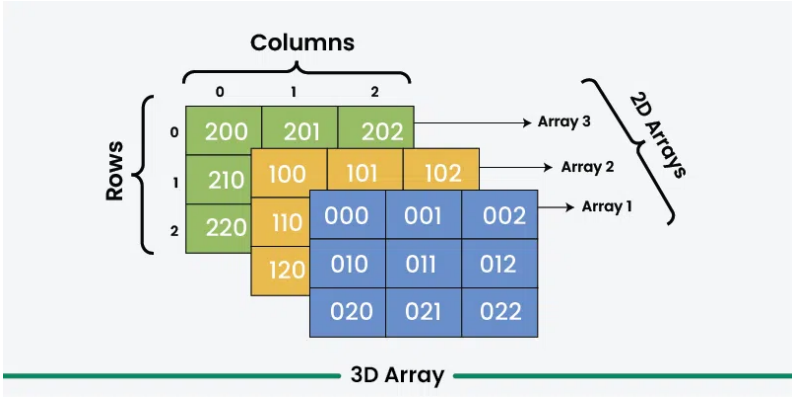


Abb. 4. Bei dreidimensionalen Arrays ist der Tiefenindex führend, dann folgen Zeilen- und Spaltenindex.

Arrayzugriff

- Arrayelemente (von Aggregatvariablen und Zigervariablen) werden inidiziert.
  - Um einzelnen Array-Elementen einen Wert zu übergeben oder Werte daraus zu lesen, wird der Indizierungsoperator [] (auch Subskript-Operator genannt) verwendet, bei mehreren Dimensionen je ein Indizierungsoperator pro Dimension.

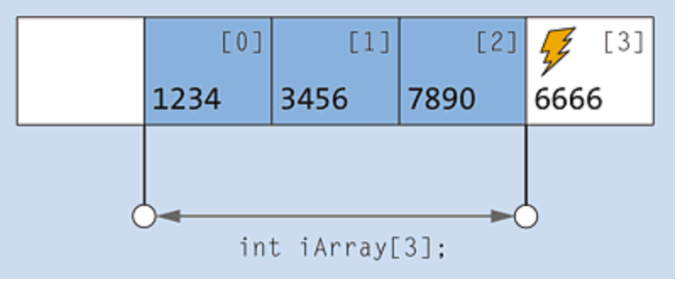


Abb. 5. Arrayzugriffe über den Index werden nicht geprüft. Ein zu großer Index führt beim Lesen zu fehlerhaften Werten, beim Schreiben zu einer Kollision mit einer anderen Variable, und ggfs. zu einem Speicherschutzfehler (z.B. Lesen/Schreiben über Segmentgrenzen hinaus).

Initialisierung mit einer Initialisierungsliste

- Ein Array (vor allem kleine Arrays) kann bereits bei der Definition mit einer Initialisierungsliste initialisiert werden. Hierbei wird bei der Definition eine Liste von Werten in geschweiften Klammern, getrennt durch Kommata, angegeben.

```
<Datentyp> <Arrayname> [ <Anzahl_der_Elemente N> ] =
  { <Wert El1>, <Wert El2> , .. , <Wert ElN> };
```

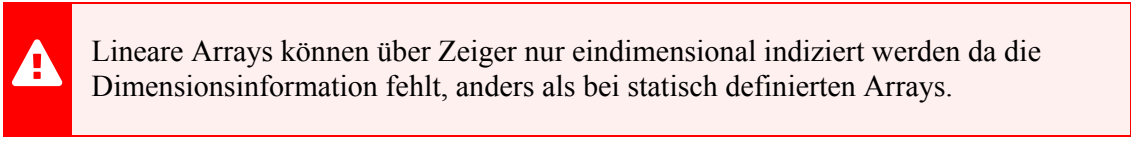
Def. 5. Initialisierung von Arrays. Die Angabe der Anzahl der Elemente kann auch weggelassen werden, ergibt sich aus der Werteliste. Die Anzahl der Elemente in der Werteliste darf aber kleiner als ein explizit angegebenes *N* sein.

Dynamische Arrayvariablen

- Anstelle einer Aggregatvariable `type name[N]` kann auch eine Zeigervariable `type *name` definiert werden.
  - Hier muss dann explizit Speicher reserviert werden.

```
<Datentyp> *<Arrayname>;
..
  <Arrayname>=(<Datentyp> *)malloc(N*sizeof(<Datentyp>));
..
```

Def. 6. Definition einer Zeigervariable für Arrays mit Speicherallokation. Ein Array wird daraus erst durch die Speicherbelegung.



Adressrechnung

1. Eindimensionale Arrays `[i]` mit Definition `<eltype> x[I]`:

$$A = i \cdot \text{sizeof}(\text{eltype})$$

2. Zweidimensionale Arrays `[i][j]` mit Definition `<eltype> x[I][J]`:

$$A = (iJ + j) \cdot \text{sizeof}(\text{eltype})$$

3. Multidimensionale Arrays `[i1][i2]..` mit Definition `<eltype> x[I1][I2]..` und Dimension *d*:

$$A = \sum_{l=1}^{d} \left(i_l \cdot \prod_{k=l}^{d} I_k \right) \cdot \text{sizeof}(\text{eltype})$$

$$\prod_{u}^{u} = 1$$

Zeigerarrays

- Wir haben festgestellt dass mehrdimensionale Arrays nicht dynamisch linear erzeugt werden können bzw. man selber explizit die Indexberechnung gemäß vorheriger Gleichungen (ohne Datentypmultiplikation) durchführen.
- Eine Möglichkeit multidimensionale Arrays zu erstellen und zu nutzen sind Zeigerarrays, d.h. die Partitionierung einer lineare Tabelle in einzelnen verkettete Tabellen.
- Bei Zeigerarrays kann man dann wieder multidimensionale Indizierung durchführen!
- Es werden als Zeigertabellen verwendet um weitere Zeigertabellen oder die Wertarrays zu referenzieren.

Zeigerarrays

```
int **xp;

int main() {
  xp=(int **)malloc(10*sizeof(int *));
  for(int i=0;i<10;i++) {
   xp[i]=(int *)malloc(10*sizeof(int));
  }
  for(int i=0;i<10;i++)  {
    for(int j=0;j<10;j++) {
      xp[i][j]=i+j;
    }
  }
}
```

Bsp. 5. Zeigerarrays von Arrays, so z.B. auch `main(char **argv)`

Zeigerarithmetik

Jetzt kann es verwirrend werden. Zeigerarihmetik ist einfach eine arithmetische Operation wie Addition die auf die Speicheradressen angewendet wird. Aber durchaus anders als erwartet.

- Numerische Addition von Werten (1:=1):

```
int x=1;
x=x+1;
x++;
print(x) // x==3;
```

- Numerische Addition von Zeigern immer in Datentypgrößeneinheiten (Basistyp des Zeigers, d.h. 1:=sizeof(typ)):

```
int *x=1
x=x+1;
x++;
print(x) // x==9 !!!!;
```

Zeigerarithmetik

```
int *xp;
..
  int *xpp;
  xp=(int *)malloc(100*sizeof(int));
  xpp=xp;
  printf("%d==%d\n",xpp,xp);
  for(int i=0;i<100;i++) {
   *xpp=i;
   xpp++;
  }
  printf("%d!=%d\n",xpp,xp);
  xpp=xp;
..
```

Bsp. 6. Arrayzugriff über dynamische Zeiger

Zeichenketten

- Zeichenketten (Strings) sind in C utf8 oder ASCII kodierte *char* Arrays.
  - Da keine Länge gespeichert wird muss eine Zeichenkette immer mit einem NULL Zeichen (Ganzzahlwert 0) abgeschlossen werden.
  - Ansonsten übliche Arrays

```
char message[]="Hello World";
char *m;
int main() {
  m=(char *)malloc(100);
  memcpy(m,message,strlen(message)+1)
  pritnf("%s %s\n",message,m);
}
```

Bsp. 7. Zeichenketten in C

Zusammenfassung

1. Aggregate sind:
   - Mehrsortige Datenstrukturen (Records, Structures)
   - Einsortige Datenstrukturen (Arrays)
2. Structures bestehen aus Feldelementen die über Namen referenziert werden (Namenentliche Indizierung)
3. Arrays bestehen aus Feldelementen die über einen numerischen Index (Beginn imm 0) refrenziert werden (Numerische Indizierung)
   - Es gibt ein- und mehrdimensionale Arrays, die auch bei der Definition mit Werten initialisiert werden könne
   - Man unterscheidet statisch und dynamisch definierte Arrays
4. Es gibt ein 1:1 Speichermodell
5. Zeiger sind Wertvariablen die aber Speicheradressen enthalten
   - Zeiger können auf alle anderen Variablen sowie Funktionen zeigen
   - Zeigerarithmetik findet in Vielfachen der Datengröße des Basistyps statt