

Programmiertechnik 1

Unit 9: Programmiersprache C –
Zeiger und Felder

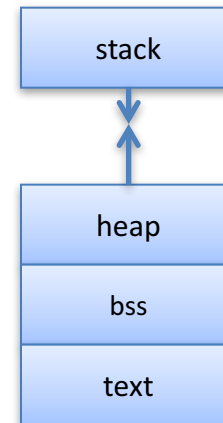
Ablauf

- Heap und Stack
- Zeiger und Adressen
- Zeiger und Funktionen (-argumente)
- Zeiger und Arrays
- Adreßarithmetik
- Beispiel: malloc und Algorithmen zur Speicherallokation
- Mehrdimensionale Felder
- Initialisierung von Feldern
- Kommandzeilenbearbeitung
- Funktionszeiger
- Komplizierte Deklarationen

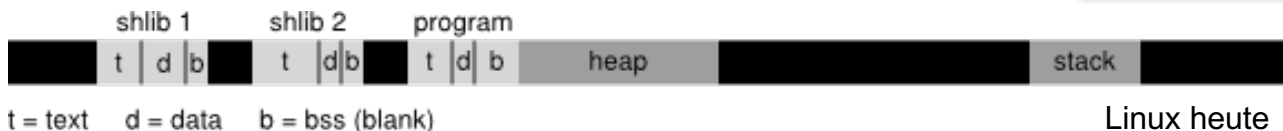
Speicher eines UNIX -Prozesses

- Virtueller Speicher
 - 2^{32} Adressen oder 2^{64} Adressen – pro Prozess
 - Aufteilung user space/kernel space
 - Benutzerprozess in 32-bit OS kann 2^{31} Byte = 2GB adressieren
 - Zumindest bei 64-bit OS gilt user space \gg physischer Hauptspeicher
- C-Programme benutzen symbolische Namen für Speicheradressen
 - Statische Variablen
 - Block static storage, bss
 - Lokale (automatische) Variablen
 - Auf dem Stack angelegt
 - Dynamisch allozierte Speicherbereiche
 - Auf dem Heap angelegt – malloc(), brk(), sbrk()
 - Gemeinsam benutzte Speicherbereiche
 - mmap()
 - Shared libraries

Alle Speicheradressen in einem C-Programm sind virtuelle Adressen



UNIX klassisch



Linux heute

Dynamik

- Größe der initialisierten statischen Variablen zur Compile-Zeit bekannt
 - Compiler kann Code zur Initialisierung von bss generieren
 - Laufzeitsystem (crt0.o) und Loader führen diesen Code aus
- Größe des Stacks ändert sich dynamisch
 - Jeder Funktionsaufruf speichert Variablen auf dem Stack
 - Rücksprungadresse, Rückgabewert
 - Parameter, lokale Variablen
 - Lokalität, Sichtbarkeit
 - Wie groß sollte Stack idealerweise sein?
- Daten auf dem Heap werden zur Programmlaufzeit angelegt
 - malloc()-, free()-Systemaufrufe
 - Unabhängig von Funktionsverschachtelung
 - Programmierer ist für Allokation/ Freigeben von Speicher verantwortlich
 - Referenzierung von Speicher auf dem Heap über ***Pointer***

Pointer (Zeiger)

- Ein Pointer ist eine Variable, die die Adresse einer Variablen enthält
 - In C exzessiv benutzt
 - Felder werden in C durch Verweis auf den Feldanfang dargestellt



- Bsp: 4-byte Zeiger p verweist auf 1-byte char c
 - Adressen sind architekturabhängig (16, 32, 64 bit)
 - Speicher ist byte-adressierbar
- Einstelliger Operator & ergibt die Adresse eines Objekts
 - `char c;`
 - `char * p = & c; /* Initialisierung */`
- Einstelliger Operator * bewirkt Dereferenzierung (*indirection operator*)
 - `char d = *p; /* Kopieren des Wertes von c */`

Zeiger – Deklaration und Benutzung

- Ein Beispiel:

```
int x = 1, y = 2, z[10];
```

```
int *ip;           /* ip ist ein Zeiger auf ein Objekt vom Typ int */
```

```
ip = &x;          /* ip verweist nun auf x    */
```

```
y = *ip;         /* y wird zu 1          */
```

```
*ip = 0;         /* x wird zu 0         */
```

```
ip = &z[0];       /* ip verweist nun auf z[0] */
```

- Mnemonic: - int *ip;

- Bedeutung: *ip ist vom Typ int

- Zeiger sind eingeschränkt: können nur auf Objekte eines vereinbarten Typs verweisen
- Ausnahme: „pointer to void“ (void *) – ein beliebiger Zeiger; keine Dereferenzierung

- Compiler überprüft passende Verwendung von Zeigern

- *ip ist ein int und kann so verwendet werden: *ip = *ip + 10;

Zeiger (contd.)

- Operatoren *, &
 - binden stärker als arithmetische Operatoren (Vorrang)

Bsp:

```
y = *ip + 1;    /* y enthält den um 1 erhöhten Wert auf den ip verweist */
*ip += 1;       /* Variable, auf die ip verweist wird inkrementiert */
++*ip;         /* ebenso */
iq = ip;       /* Zuweisung; iq verweist nun auf gleiche Adresse wie ip */
(*ip)++;
```

- Klammern sind wegen Assoziativität der einstelligen Operatoren nötig

```
*ip++;
```

- Hier würde erst ip inkrementiert und danach dereferenziert



Zeiger und Funktionsargumente

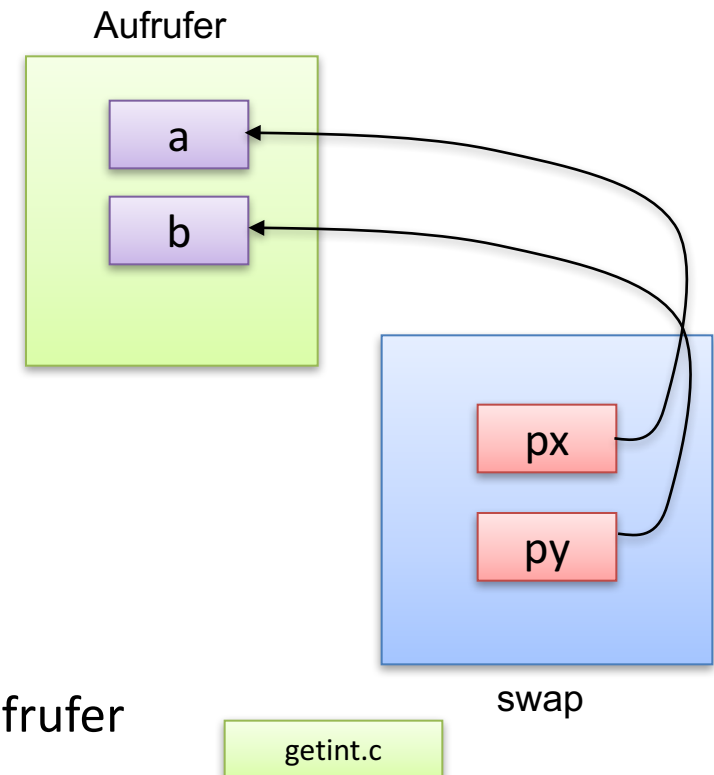
- C kennt nur Werteparameter (pass-by-value)
 - Keine direkte Möglichkeit für eine Funktion, Werte im Aufrufer zu ändern
 - Übergabe der Adresse einer Variablen ermöglicht gerufener Funktion, diese Variable zu ändern

– Bsp:

- Vertauschen der Werte zweier Variablen
- `swap(&a, &b);`

```
void swap( int *px, int *py) {  
    int temp = *px;  
    *px = *py;  
    *py = temp;  
}
```

- Zeiger ermöglichen Zugriff auf Objekte im Aufrufer



Zeiger und Felder

- Enge Beziehung zwischen Zeigern und Feldern:
 - Jede Operation, die durch Feldindizierung ausgedrückt werden kann, kann auch als Zeiger-Operation dargestellt werden.
 - Zeiger-Operationen sind häufig schneller

- Beispiel:

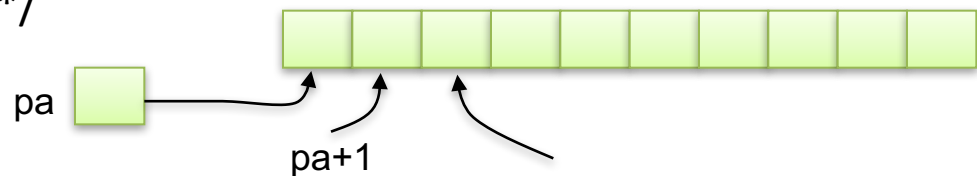
- `int a[10];`



- `int *pa = &a[0];`



- `x = *pa; /* kopiert a[0] */`



- `y = *(pa+1); /* kopiert a[1] */`

Zeiger und Felder (contd.)

- $*(pa+3)$ – verweist auf viertes Feldelement
 - Unabhängig vom Typ der im Feld gespeicherten Elemente
- „Zeiger um 1 erhöhen“
 - Bedeutung ist abhängig vom Typ, auf den der Zeiger verweist
 - Besser: Zeiger auf das nächste Objekt „vorrücken“
 - Bsp: 4-byte int $*pa$; $\rightarrow (pa+1)$ bewirkt einer Erhöhung um 4 Byte
- Index-Operationen und Zeigeroperationen sind sehr ähnlich:
 - Wert einer Array-Variablen ist per Definition die Adresse des ersten Elements
 - $pa = \&a[0]$;
 - Bewirkt, dass pa und a identische Werte haben
 - Äquivalent zu $pa = a$;
 - Referenz $a[i]$
 - Kann geschrieben werden als $*(a+i)$; - beide Formen sind äquivalent

Zeiger und Felder (contd.)

- Array != Pointer

- Zeigervariable kann in Ausdrücken links stehen (lvalue)

- Felder sind keine Variablen

```
char a[5];  
char * pa = a; pa++;    /* legal    */  
a = pa;                /* illegal  */
```

- Sind Felder Funktionsargumente, so wird Adresse des ersten Elements übergeben.

- Bsp: strlen() operiert auf eigener lokaler Kopie des Zeigers

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

- Gültige Verwendungen:

```
strlen("hello world");    /* Zeichenkettenkonstante */  
strlen(array);           /* char array[100]; */  
strlen(ptr);             /* char *ptr; */
```

strlen.c

Formale Argumente

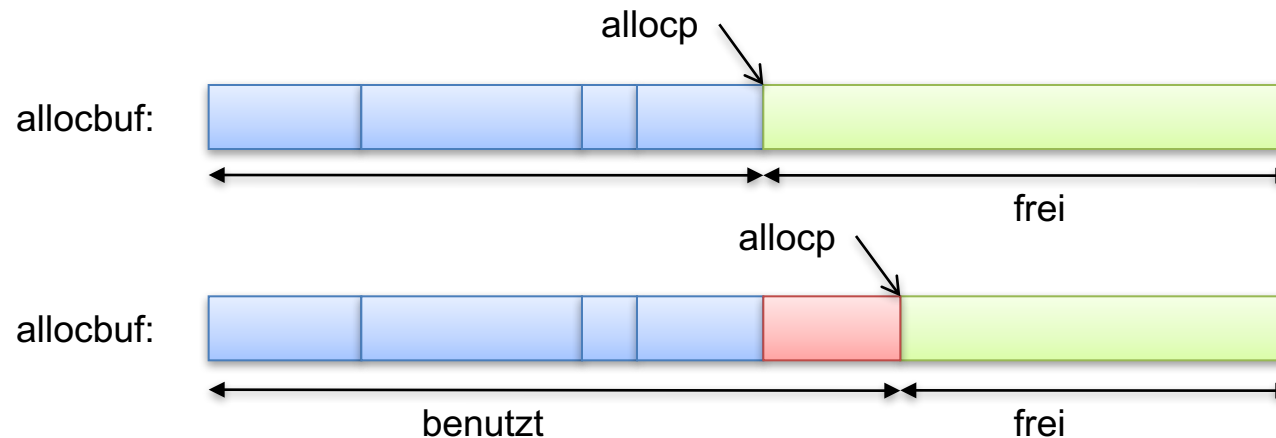
- Zeiger als Funktionsargumente
 - Folgende Deklarationen sind syntaktisch äquivalent
 - `func(char s[]);`
 - `func(char * s);`
 - Variante 1 ist besser lesbar; bevorzugt!
- Beliebige Teile eines Feldes können übergeben werden
 - Nicht unbedingt der Anfang des Feldes
 - `func(&a[2]);` oder `func(a+2);`
 - Für aufgerufene Funktion ist nicht sichtbar, dass a Teil eines größeren Feldes ist
- Keine Überprüfung des Indexwertebereiches durch den **Compiler**
 - Anders als Pascal, Modula-2, Java, C#...
 - Auch negative Indizes sind erlaubt
 - Bspw. In `func(): char c = s[-2];`
 - Dennoch: Zugriff auf Objekt außerhalb Feldindex ist **illegal**.

Adressarithmetik

- Sei p eine Zeiger auf ein Feldelement
 - $p++$; inkrementiert p auf das nächste Feldelement
 - $p+=i$; verschiebt Zeiger um i Feldelemente
- Seien p und q Zeiger auf dasselbe Feld
 - Addition und Subtraktion sind für Zeiger zulässig: $\text{next10} = p + 10$;
 - Vergleich: $==$, $!=$, $<$, $>=$, etc.
 - $\text{if } (p > q) \text{ num_elem} = q - p + 1$; /* Zahl der zwischenliegenden Elemente */
- Zeiger sind keine Integer
 - Einzige Ausnahme: Konstante 0
 - Zeiger können mit 0 verglichen werden
 - 0 ist niemals eine gültige Adresse
 - Symbolische Konstante `NULL` /* definiert in `stdio.h` */
- Resultate undefiniert wenn p und q nicht auf dasselbe Feld verweisen

Speicherverwaltung

- Ein Beispiel für eine einfache Heap-Speicherverwaltung
 - Verwaltung in einem statischen Feld `allocbuf`
 - `alloc()` und `afree()` müssen „passend“ aufgerufen werden (Strukturierung als Stack)



- `allocp` markiert den Punkt, bis zu dem Speicher benutzt ist
 - `alloc()` inkrementiert `allocp` und gibt alten Wert zurück
 - `Afree()` dekrementiert `allocp`

`alloc.c`

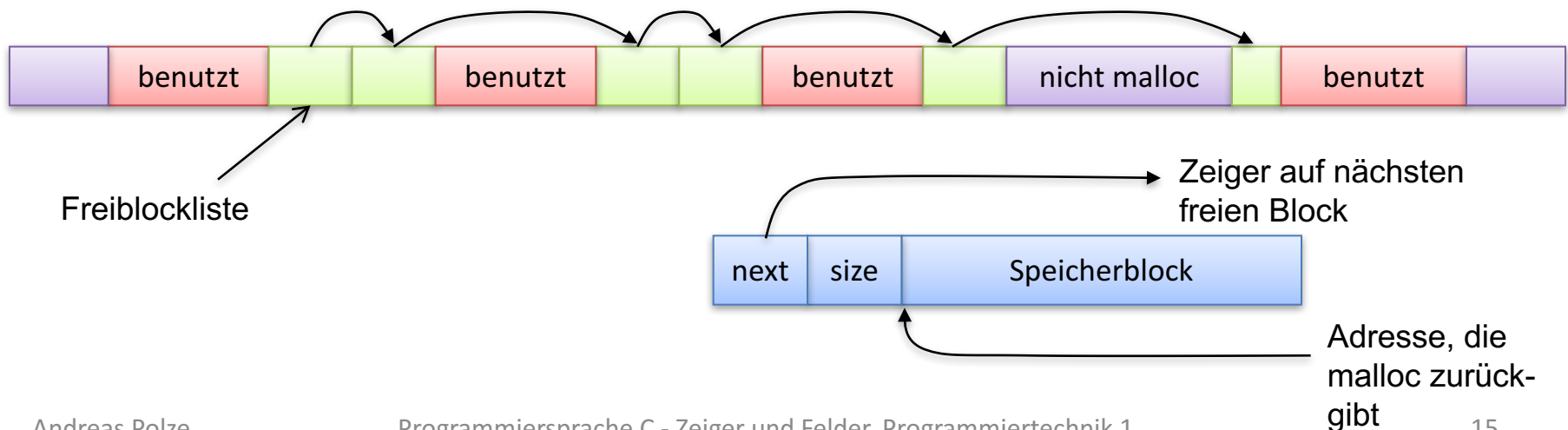
`textstore.c`

Speicherverwaltung (richtig)

- C Laufzeitsystem bietet mehrere Routinen zur Verwaltung des Heaps

```
# include <stdlib.h>  
void free(void *ptr);  
void * malloc(size_t size);
```

- malloc fordert Heap-Speicher vom Betriebssystem an
 - Verwaltet free-Liste
 - Vergibt Blöcke nach „first fit“-Strategie



Verwendung von Heapspeicher

- malloc und free können in beliebiger Reihenfolge gerufen werden
 - Vergessene free-Operationen führen zu „Speicherlecks“
 - Betriebssystem gibt bei Terminierung eines Prozesses allen Speicher frei
- Granularität der Speicherallokation
 - Häufige Aufrufe von malloc → großer Overhead
also: große Blöcke verwenden
 - Das Finden eines freien Blocks ist viel einfacher bei kleinen Blöcken
also: kleine Blöcke verwenden
 - malloc() verwaltet Speicher im user-mode
→ nur selten Betriebssystemaufrufe zum Ausfassen von Speicher nötig
 - malloc() und free() sind nicht thread-sicher

textstore2.c

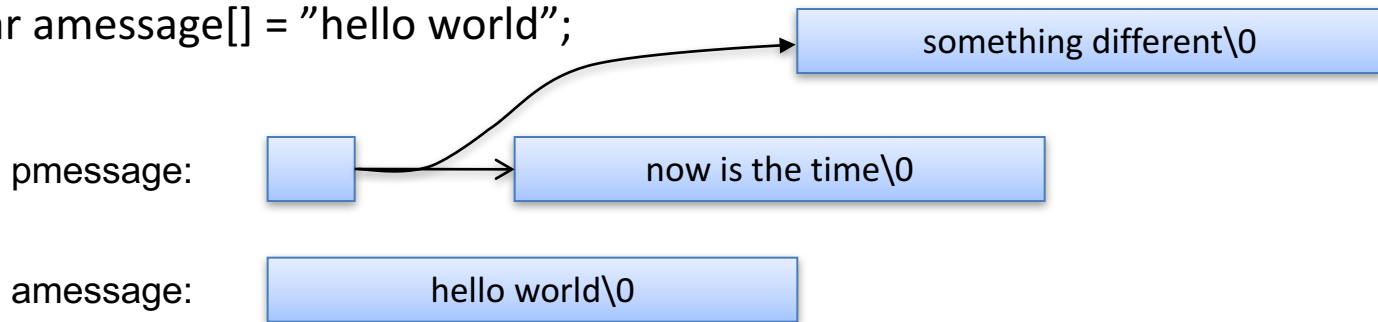
Zeigerarithmetik

- Zeiger und Integer können addiert/subtrahiert werden
 - $p + n \rightarrow$ das n -te Objekt hinter dem, auf das p gegenwärtig zeigt
 - Unabhängig vom Typ des Objektes p
 - n wird entsprechend der Größe des Objekttyps skaliert
 - Objekttyp wird bei Definition von p festgelegt
 - Wäre p vom Type char^* , so würde byteweise inkrementiert
 - Wäre p vom Type float^* , so würde $p++$ auf den nächsten float-Wert weisen
 - Inkrement um 4 Byte
- Gültige Operationen für Zeiger
 - Zuweisung, Addition/Subtraktion von Zeiger und Integer
 - Vergleich zweier Zeiger die auf dasselbe Feld zeigen
 - Zuweisung von NULL
 - Vergleich mit NULL
- Alle Zeiger können nach void^* umgewandelt werden und umgekehrt

Zeichenkettenkonstanten vs. Felder

- Zeichenkettenkonstanten

- `char * pmessage = "now is the time";`
- `char amessage[] = "hello world";`



- `pmessage`: Zeiger kann geändert werden, Zeichenkette nicht
- `amessage`: Zeichenkette kann beliebig modifiziert werden

```
amessage[0] = 'H'; amessage[6] = 'W';
```

Dereferenzierung

- Feldzugriffs-Operator ([]) und Zeiger-Dereferenzierung (*) lassen sich in Funktionen synonym verwenden

```
/* strcpy: copy t to s; array subscript version */
```

```
void strcpy1( char * s, char * t ) {  
    int i = 0;  
    while ((s[i] = t[i]) != '\0')  
        i++;  
}
```

```
/* strcpy: copy t to s; pointer version 1 */
```

```
void strcpy2( char * s, char * t ) {  
    while ((*s = *t) != '\0') {  
        s++; t++;  
    }  
}
```

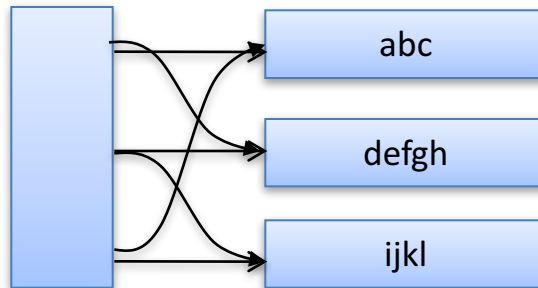
Vorteil strcpy2:
Funktioniert auch für sehr
lange Zeichenketten

(wenn Länge nicht mehr
durch int ausgedrückt
werden kann)

strcpy.c

Felder von Zeigern

- Zeiger sind Variablen
- Können in Feldern gespeichert werden
 - Bsp: Der Länge nach sortieren



- Zeiger sind in einem Feld gespeichert
- Vertauschen der Zeilen durch Ändern der Zeiger
- Speicherverwaltung bleibt einfach

lensort.c

Mehrdimensionale Felder

- C bietet rechteckige, mehrdimensionale Felder - Matrizen
 - `int A[3][4]; int B[3][4][5];`
 - Indexbereich: jeweils 0..n

1	2	3	4
5	6	7	8
9	10	11	12

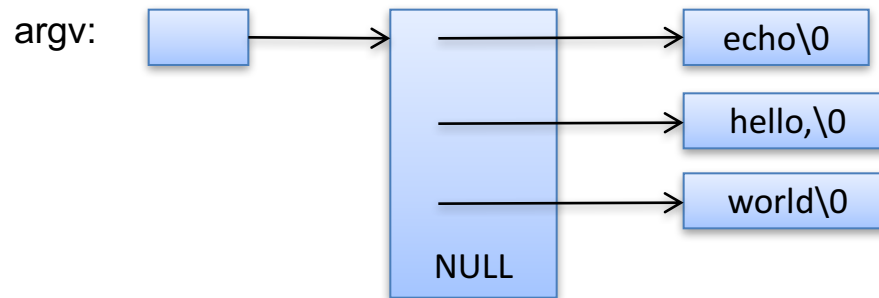
- Zugriff auf Feldelemente:
`int i = A[1][2];`
- Alle Feldelemente haben denselben Typ
 - Zeilenweise im Speicher abgelegt
 - Initialisierung durch Werte in geschweiften Klammern
`A = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 } }`
- Bei Deklaration von Funktionsparametern muss Zahl der Spalten angegeben werden
 - `func(int a[][4]);` besser `func(int a[3][4]);`
 - Erste Dimension kann weggelassen werden (...Übergeben wird noch immer ein Zeiger auf erstes Element)

Zeiger vs. Mehrdimensionale Felder

- Mehrdimensionale Felder belegen stets (n x m x p...) viele Speicherzellen
 - `int A[10][20]` belegt 200 Zellen á 4byte = 800 byte
 - Alternative: `int *B[10];`
Feld von Zeigern auf Felder mit 20 Integern
 - Belegt initial nur 10 Speicherzellen á 4byte (32bit) == 40 byte
 - `A[3][4] = B[5][6]` → syntaktisch äquivalenter Zugriff
 - Passende Initialisierung von B vorausgesetzt
 - Zeilen in einem Zeiger-Feld können unterschiedliche Länge haben
 - Effiziente Datenstrukturen
 - „Löcher“
 - Für Zeichenketten bieten sich generell Felder von Zeigern an

Kommandozeilen

- C-Laufzeitumgebung enthält Standardmechanismus der Parameterübergabe
 - `int argc` → Zahl der Argumente
 - `char * argv[]` → NULL-terminiertes Feld von Zeigern auf die Argumente
 - Bsp:
 `echo hello, world`



- Erstes Argument: `argv[0]` → Name des Programms
- Optionale Argumente: `argv[1]...argv[argc-1]`
- C-Standard verlangt: `argv[argc] == NULL`
- Feld von Zeigern auf Zeichenketten
 - Kann gelesen und geschrieben werden

lensort.c

Zeiger auf Funktionen

- Funktionen repräsentieren Sprünge im Code
 - Funktion wird repräsentiert durch Anfangsadresse eines Codesegments
 - Zeiger auf Funktionen speichern diese Adresse
- Funktionszeiger können
 - In Variablen, Feldern etc. gespeichert werden
 - Als Funktionsargumente übergeben werden
 - Von Funktionen zurückgegeben werden
- Deklaration
 - `<type> (* <name>) ([<type>+]);`
 - `int (* compare) (char *, char *) = strcmp; /* strcmp wird zugewiesen */`
- Aufruf
 - `(* compare) ("hello", "world"); /* strcmp wird aufgerufen */`

strsort.c

Komplizierte Deklarationen

- C Syntax ist mitunter verwirrend

- `int * f();` /* f: eine Funktion, die einen Zeiger auf int zurückgibt */
- `int (*pf)();` /* pf: Zeiger auf eine Funktion, die int zurückgibt */

– Problem:

- * ist Präfix-Operator, hat niedrigeren Vorrang als ()
- Klammersetzung ist nötig um richtige Assoziation zu erzwingen

- Beispiele:

<code>char ** argv</code>	→ argv: Zeiger auf Zeiger auf char
<code>int (*daytab)[13]</code>	→ daytab: Zeiger auf Feld[13] von int
<code>int * daytab[13]</code>	→ daytab: Feld[13] von Zeigern auf int
<code>void * comp()</code>	→ comp: Funktion, die Zeiger auf void zurückgibt
<code>void (*comp)()</code>	→ comp: Zeiger auf Funktion, die void zurückgibt
<code>char ((*x()) [])()</code>	→ x: Funktion, die einen Zeiger auf ein Feld[] von Zeigern auf Funktionen die char zurückgeben, zurückgibt
<code>char (* (*x[3])()) [5]</code>	→ x: Feld[3] von Zeigern auf Funktionen, die einen Zeiger auf ein Feld[5] von char zurückgeben

- Lösung: typedef – schrittweise, verständliche Typdefinitionen (Unit 10)

Zusammenfassung

- Heap und Stack
 - Zwei getrennte Speicherbereiche, Lebensdauer von Objekten
- Zeiger und Adressen
 - Zeiger und Funktionen (-argumente) → call-by-value emulieren
 - Zeiger und Arrays → jedes Feld wird durch Zeiger auf Anfang dargestellt
- Adreßarithmetik
 - Erlaubt für Zeiger, die auf das gleich Feld verweisen
 - Beispiel: malloc und Algorithmen zur Speicherallokation
- Mehrdimensionale Felder
 - Variabel nur in erster Dimension
 - Initialisierung von Feldern → legt erste Dimension fest
 - Bei unvollständiger Liste von Initialisierern bleiben Einträge undefiniert
- Kommandozeilenbearbeitung
 - Feld mit Verweisen auf Argumente und Argumentzahl vom Laufzeitsystem bereitgestellt
- Funktionszeiger
 - Eleganter, flexibler Ansatz um generische Programme zu schreiben
 - Komplizierte Deklarationen