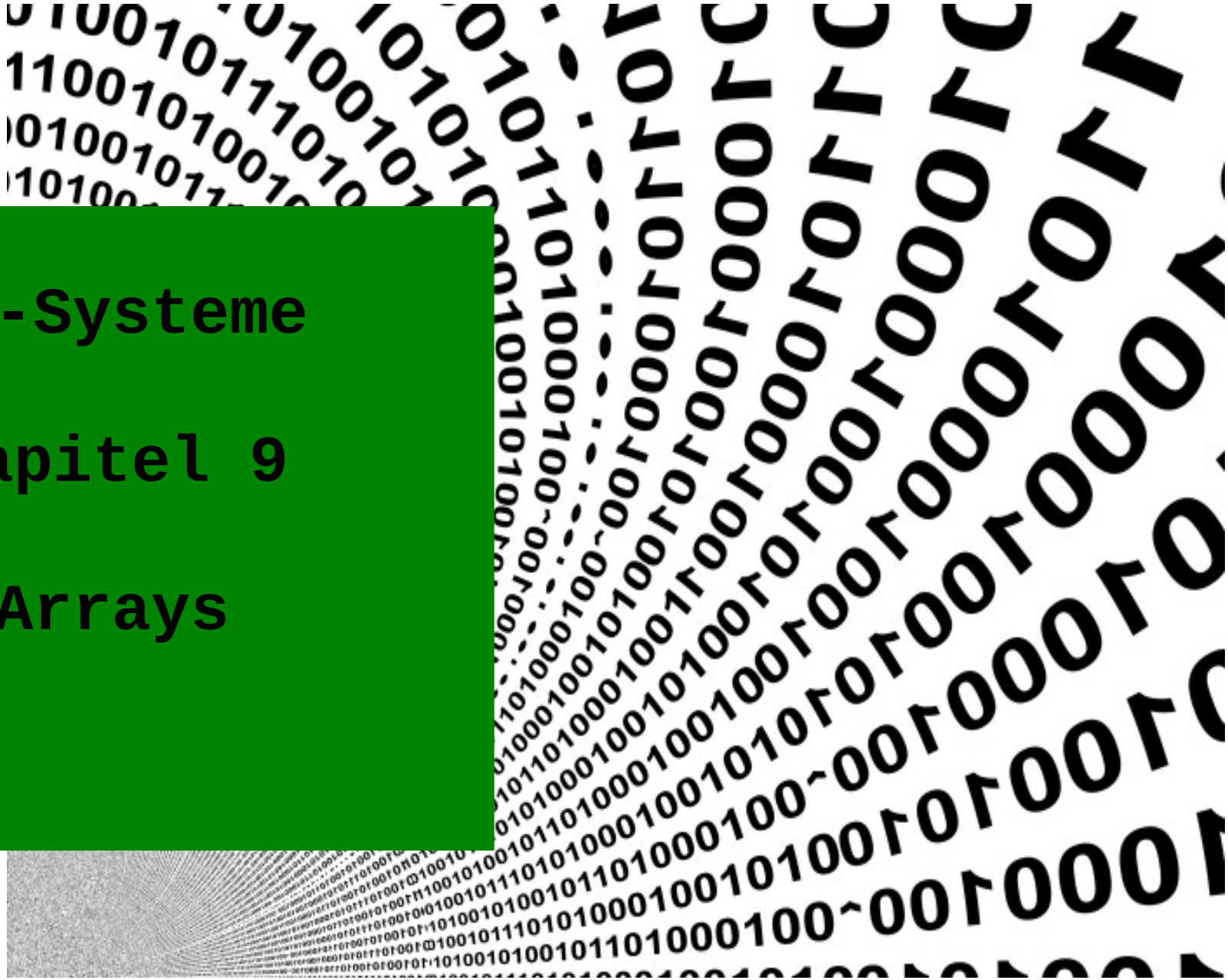


DS-Systeme

Kapitel 9

Arrays

Inhaltsverzeichnis

Arrays – Begriffsdefinition

- Ein Array ist eine Zusammenfassung / Abfolge von **gleichartigen Datenelementen** in einem zusammenhängenden Speicherbereich.
- Der Name eines Arrays (in C und in Assembler) repräsentiert die Adresse des ersten Arrayelements.

Beispiele für eindimensionale Arrays:

double Array

23.4	100.0	77.0	-20.89	19.9	2.3e10	5.0e-3	-98.3

Größe: **8 * sizeof(double) bytes**

int Array

23	100	77	-20	19	2	5	-98

Größe: **8 * sizeof(int) bytes**

char Array

'H'	'a'	'l'	'l'	'o'	' '	'd'	'u'

Größe: **8 * sizeof(char) bytes**

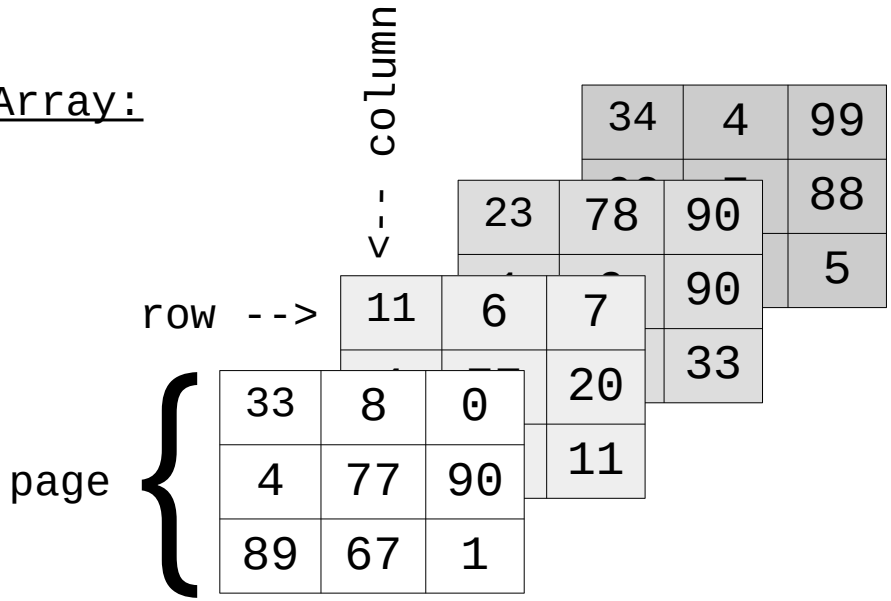
Arrays – Begriffsdefinition

Beispiel für ein mehrdimensionales Array:

```
unsigned int Array
```

Size:

```
   4 (Seiten)
 * 3 (Zeilen)
 * 3 (Spalten)
 * sizeof(unsigned int) Bytes
= 36 * sizeof(unsigned int) Bytes
```



Darstellung der ersten beiden Seiten im Speicher:

33	8	0	4	77	90	89	67	1	11	6	7	4	77	20	19	88	11

siehe auch **ArraysInMemory.pdf**

Arrays – Deklaration bzw. Definition

Globale Arrays werden entweder in der **.section .data** oder der **.section .bss** deklariert und definiert. Lokal definierte Arrays werden auf dem Stack abgelegt.

.section .data

Der Unterschied zwischen einem **initialisierten Array** und einer **einzelnen initialisierten Variablen** ist, dass bei der Initialisierung mehrere Werte durch Komma getrennt angegeben werden.

Beispiel: **Initialisiertes Array**

C/C++ - Code:

```
int intarr[11] = {10, 15, 20, 25,
                  30, 35, 40, 45,
                  50, 55, 60};
```

Assembler - Code:

```
.section .data
intarr:
  .int 10, 15, 20, 25,
       30, 35, 40, 45,
       50, 55, 60
```

Arrays – Deklaration bzw. Definition

.section .bss

Der Unterschied zwischen einem mit 0en initialisierten Array (engl.: **zero-initialized array**) und einer mit 0 initialisierten Variablen (engl.: **zero-initialized variable**) besteht darin, dass die Größe als ein Vielfaches der Größe des einzelnen Array-Elements angegeben wird.

Beispiel: Nicht initialisiertes (zero-initialized) Array

C/C++ - Code:

```
int intarrzero[11];
```

Assembler - Code:

```
.section .bss
# 11 * 4 bytes = 44 bytes
.lcomm intarrzero, 44
```

oder

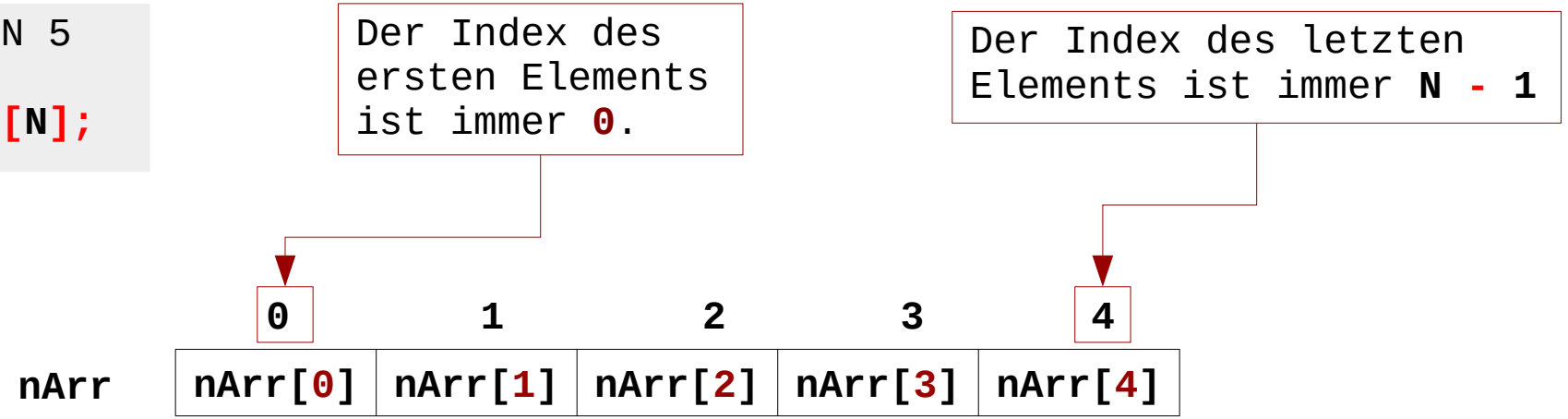
```
.section .bss
# 11 * 4 bytes = 44 bytes
.comm intarrzero, 44
```

Arrays – Zugriff - allgemein

Ein Arrayelement **A[i]** wird an der Adresse X_A **+ i * L** gespeichert. Dabei bedeuten

- X_A: Adresse des Arrays bzw. des ersten Elementes des Arrays
- **i**: Der Index zwischen **0** und **N-1**
- **N**: Anzahl der Arrayelemente
- **L**: Größe des Datentyps der Arrayelemente in Bytes

```
#define N 5
...
int nArr[N];
```



Arrays – Zugriff - allgemein

Beispiele:

Name L N * L

C/C++ - Code:

```
char A[12];
char* B[8];
int C[6];
double* D[5];
```

Array	Element size	Total size	Start address	Element *i*
A	1	12	X_A	$X_A + i$
B	8	64	X_B	$X_B + 8i$
C	4	24	X_C	$X_C + 4i$
D	8	40	X_D	$X_D + 8i$

Zeigerarithmetik

Annahme:
- Startadresse des **int**-Arrays **E** in Register **rdx** als X_E.
- Ergebnis des Ausdrucks (Expression) entweder
  - als **int**-Wert in Register **eax** oder
  - als Zeiger auf **int**-Wert in Register **rax**
- Index **i** in Register **rcx** (8 Byte)

Spalten **Expression** und **Type** beziehen sich auf **C** unter **Linux** (x86-64).

Expression	Type	Value	Assembly code
`E`	`int *`	x_E	`movq %rdx,%rax`
`E[0]`	`int`	$M[x_E]$	`movl (%rdx),%eax`
`E[i]`	`int`	$M[x_E + 4i]$	`movl (%rdx,%rcx,4),%eax`
`&E[2]`	`int *`	$x_E + 8$	`leaq 8(%rdx),%rax`
`E+i-1`	`int *`	$x_E + 4i - 4$	`leaq -4(%rdx,%rcx,4),%rax`
`*(E+i-3)`	`int`	$M[x_E + 4i - 12]$	`movl -12(%rdx,%rcx,4),%eax`
`&E[i]-E`	`long`	i	`movq %rcx,%rax`

Zeigerarithmetik

Annahme:
- Startadresse des **short**-Arrays **S** in Register **rdx** als X_s.

- Ergebnis des Ausdrucks (Expression) entweder
  - als **short**-Wert in Register **ax** oder
  - als Zeiger auf **short**-Wert in Register **rax**

- Index **i** in Register **rcx** (8 Byte)

Expression	Type	Value	ASM-Code
S+1			
S[3]			
&S[i]			
S[4*i+1]			
S+i-5			

Aufgabe: Füllen Sie die Felder entsprechend der Vorgaben und der vorherigen Seite aus.

Arrays – Zugriff Version 1

```
.section .data
intarr:
  .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60 # 11 elements

.section .text
.globl main
main:
  pushq %rbp
  movq %rsp, %rbp

  movq $0, %rcx # index
  jmp loopTest

loop:
  movl intarr(, %rcx, 4), %eax
  #... do something
  inc %rcx

loopTest:
  cmpq $11, %rcx
  jne loop

  # exit main
  movq $0, %rax
  popq %rbp
  ret
```

Verwendung des folgenden Zugriffsschemas:

- **array access math:**
  &A[i] = X_A + i * L

- **general asm access math:**
  effective_addr
  = offset_addr + base_addr + index * sf

- **general asm code:**
  offset_addr(base_addr_reg, index_reg, sf)
  Imm(rb, ri, s);

Hier verwendetes Schema:

&A[i] = X_A_asImm(, i_indexReg, L_as_Sf)

- **alternative array access scheme:**
  &A[i] = (X_A_inreg, i_inreg, L_as_sf)

Arrays – Zugriff Version 2

```
.section .data
intarr:
  .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60 # 11 elements

.section .text
.globl main
.type main, @function
main:
  pushq %rbp
  movq %rsp, %rbp
  leaq intarr, %rdx  # startaddress of array
  movq $0, %rcx # index
  jmp loopTest

loop:
  movl (%rdx, %rcx, 4), %eax
  #... do something
  inc %rcx

loopTest:
  cmpq $11, %rcx
  jne loop

  # exit main
  movq $0, %rax
  popq %rbp
  ret
```

Aufgabe:

Übersetzen Sie dieses und das GNU-Assembler-Programm von der vorherigen Seite in die Sprache C.

Hier verwendetes Schema:

- **alternative array access scheme:**
  &A[i] = (X_A_inreg, i_inreg, L_as_sf)

Mehrdimensionale Arrays – Zugriff

Die Adresse eines Elements **&A[i][j]** eines zweidimensionalen Arrays **A** wird an der Adresse $\mathbf{X_A + L * (C * i + j)}$ gespeichert. Dabei bedeuten

- $\mathbf{X_A}$: Adresse des Arrays bzw. des ersten Elementes des Arrays
- **i**: Zeilenindex
- **j**: Spaltenindex
- **C**: Dimension von **j** (Breite einer Zeile)
- **L**: Größe des Datentyps der Arrayelemente in Bytes

12	45	-3	16
77	36	-2	98
48	66	5	8

Beispiel 1:

Das Element **A[2][1] = 66** des **int**-Arrays **A** (angenommene Adresse: 1000) befindet sich an der Adresse:

$$\mathbf{X_A + L * (C * i + j)}$$
$$\mathbf{= 1000 + 4 * (4 * 2 + 1)}$$
$$\mathbf{= 1036}$$

Mehrdimensionale Arrays – Zugriff

Beispiel 2:

Row	Element	Address
A[0]	A[0][0]	x_A
	A[0][1]	$x_A + 4$
	A[0][2]	$x_A + 8$
A[1]	A[1][0]	$x_A + 12$
	A[1][1]	$x_A + 16$
	A[1][2]	$x_A + 20$
A[2]	A[2][0]	$x_A + 24$
	A[2][1]	$x_A + 28$
	A[2][2]	$x_A + 32$
A[3]	A[3][0]	$x_A + 36$
	A[3][1]	$x_A + 40$
	A[4][2]	$x_A + 44$

ASM-Code für Array-Zugriff bzw. Kopieren eines Elements aus dem gegebenen Array der Größe **4 * 3** aus Integer-Elementen (**int A[4][3]**) (genereller Aufbau s. Abbildung links) ins Register **eax**:

&A[i][j] = X_A + L * (C * i + j)
==> &A[i][j] = X_A + 4 * (3 * i + j)
==> &A[i][j] = X_A + 12 * i + 4 * j

Anmerkung:

X_A in **rdi**, **i** in **rsi** und **j** in **rdx**.

```
# Compute (3 * i)
leaq  (%rsi, %rsi, 2), %rax  # 2i + i = 3i in rax

# Compute (X_A + 4 * (3 * i)) =  X_A + 12 * i
leaq  (%rdi, %rax, 4), %rax  # X_A + L*3*i  (L = 4)
                             # = X_A + 12i in rax
# Read from (X_A + 12 * i + 4 * j)
movl  (%rax, %rdx, 4), %eax  # X_A + 12i + 4j
```

Mehrdimensionale Arrays – Aufgabe

C/C++ - Code:

```
short aSh[2][4] = {
    { 3, 17, -5, 6},
    { 2,  1,  9, 8}
};
```

Addieren Sie die **17** und **9** aus dem Array und speichern das Ergebnis im Register **ax**.

Hinweis:

- **aSh** in **rdi**
- **i** in **rsi**
- **j** in **rdx**

Zwischenergebnisse in **rax** und **rcx**.

Starten Sie mit:

```
leaq aSh, %rdi
movq $..., %rsi
movq $..., %rdx
```

Vorgehensweise wie auf vorheriger Seite.