Zeiger

Einführung in die Programmierung

Michael Felderer

Institut für Informatik, Universität Innsbruck

Zeiger (Pointer)

- Der Arbeitsspeicher eines Rechners ist in Speicherzellen eingeteilt.
  - Jede Speicherzelle trägt eine Nummer.
  - Die Nummer wird als Adresse bezeichnet.
  - Ist der Speicher eines Rechners byteweise ansprechbar, so sagt man, er sei byteweise adressierbar.
    - Pro Byte wird eine Adresse benötigt.
- Ein Zeiger (Pointer) ist eine Variable, welche die **Adresse** einer im Speicher befindlichen Variable oder Funktion aufnehmen kann.
  - Damit **verweist** eine Zeigervariable mit ihrem Wert **auf die jeweilige Adresse**.
- Zeiger und Speicherobjekte (auf die der Zeiger zeigt) sind vom Typ her gekoppelt.
  - Ist das Objekt vom Typ „Typname“, so benutzt man einen Zeiger vom Typ „Zeiger auf Typname“, um auf dieses Objekt zugreifen zu können.

Anwendungsgebiete von Zeigern

- Speicherbereiche können dynamisch reserviert, verwaltet und wieder gelöscht werden.
- Mit Zeigern können Datenobjekte direkt (call-by-reference) an Funktionen übergeben werden.
- Mit Zeigern lassen sich Funktionen als Argumente an andere Funktionen übergeben.
- Rekursive Datenstrukturen wie Listen oder Bäume lassen sich fast nur mit Zeigern erstellen.
- Es kann ein typenloser Zeiger definiert werden, womit Datenobjekte beliebigen Typs verarbeitet werden können.

Definition von Zeigervariablen

- Ein Zeiger wird wie eine Variable deklariert:

  `Typname *Zeigername;`

  - `Typname *` ist der Datentyp des Zeigers.
  - `Zeigername` ist der Name des Zeigers.
- Beispiele, traditionell notiert:

```
int *pointer1;
float *pointer2;
```

  - `int *pointer, counter;` (nur `pointer` ist eine Zeigervariable!)
- Beispiele, nach einer modernen Konvention notiert:

```
int* pointer1;
float* pointer2;
int* pointer;
int counter;
```

Wertzuweisung an einen Zeiger – Adressoperator

- Man kann Zeiger zuweisen
  - `pointer2 = pointer1;`
  - Beide Zeiger haben nach der Zuweisung den gleichen Inhalt, d.h. sie zeigen auf das **gleiche Objekt**.
- Adressoperator &
  - Ist x eine Variable von Typ Typname, dann liefert der Ausdruck &x einen Zeiger auf das Objekt x vom Typ Zeiger auf Typname.
  - Beispiel

```
int counter = 1;
int *pointer = &counter;
…
```

Initialisierung

- Wurde einem Zeiger noch keine Adresse zugewiesen, so hat dieser Zeiger **keinen definierten Wert**!
- Der Zeiger wird auf eine zufällige Speicherstelle zeigen!
- Die Verwendung wird zu Problemen führen!
- Beispiele für Probleme:
  - Z.B. einen Wert aus einer zufälligen Speicherstelle auslesen.
  - Auf eine Speicherstelle zugreifen, die gar nicht zum Programm gehört = Absturz des Programms.

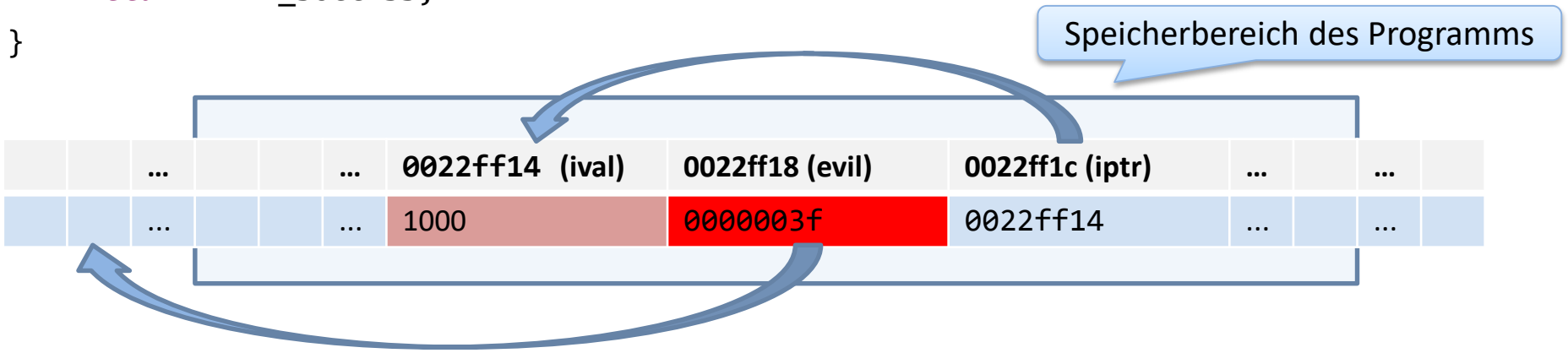
Beispiel

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *iptr, *evil;
    int ival = 1000;
    iptr = &ival;
    printf("Adresse iptr: %p\n", &iptr);
    printf("zeigt auf   : %p\n", iptr);
    printf("Adresse ival: %p\n", &ival);
    printf("Adresse evil: %p\n", &evil);
    printf("zeigt auf   : %p\n", evil); // Nicht initialisiert! Zufälliger Wert!
    return EXIT_SUCCESS;
}
```

Ausgabe (Windows 7):

```
Adresse iptr: 0022ff1c
zeigt auf   : 0022ff14
Adresse ival: 0022ff14
Adresse evil: 0022ff18
zeigt auf   : 0000003f
```



Wertebereiche

- Für den Wertebereich eines Zeigers gibt es zwei Möglichkeiten
  - Menge aller Zeiger, die auf Speicherobjekte vom Typ Typname zeigen können.
  - NULL-Pointer
    - Der Zeiger NULL ist vordefiniert (Konstante NULL in <stddef.h>).
    - `int *pointer = NULL;`
- Es wird kein Speicherplatz für ein Objekt vom Typ Typname reserviert!
- Es wird **Speicher für die Darstellung der Adresse** reserviert!

Dereferenzierung (1)

- Wurde einem Zeiger eine Adresse zugewiesen, dann will man auch auf das Objekt dahinter zugreifen können (den Inhalt manipulieren können).
- In C wird dafür der Inhaltsoperator (Dereferenzierungsoperator) * verwendet.
- Beispiel:

```
int counter = 1;
int counter2;
int *pointer;
pointer = &counter;
*pointer = 5;
counter2 = *pointer;
```

Beide Variablen counter und counter2 haben am Ende den Wert 5!

Dereferenzierung (2)

- Zeigt ein Zeiger `pointer` auf eine Variable `counter`, so kann statt `counter` immer der äquivalente Ausdruck `*pointer` verwendet werden.

```
*pointer = *pointer + 1;
```

entspricht

```
counter = counter + 1;
```

- Für eine Variable `alpha` ist `*&alpha` äquivalent zu `alpha`

Beispiel (Zeiger)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
      int *iptr;
      int ival = 255;
      int ival2 = 1024;
      iptr = &ival;
      printf("Adresse iptr: %p\n", &iptr);
      printf("zeigt auf:    %p\n", iptr);
      printf("Adresse ival: %p\n", &ival);
      printf("*iptr: %d\n", *iptr);
      printf(" ival: %d\n", ival);
      *iptr = 128;
      printf("*iptr : %d\n", *iptr);
      printf(" ival : %d\n", ival);
      printf(" ival2: %d\n", ival2);
      *iptr = ival2 * 2;
      ival2 = *iptr;
      printf("*iptr : %d\n", *iptr);
      printf(" ival : %d\n", ival);
      printf(" ival2: %d\n", ival2);
      return EXIT_SUCCESS;
}
```

Ausgabe (zid-gpl):

```
Adresse iptr: 0x7fff79485f10
zeigt auf:    0x7fff79485f0c
Adresse ival: 0x7fff79485f0c
*iptr: 255
 ival: 255
*iptr : 128
 ival : 128
 ival2: 1024
*iptr : 2048
 ival : 2048
 ival2: 2048
```

Interaktive Aufgabe

- Welcher Fehler wurde im nachfolgenden Code gemacht?

```
int *ptr;
int ival;
ptr = ival;
*ptr = 255;
```

NULL-Zeiger

- Nicht verwendete Zeiger sollten zunächst mit NULL initialisiert werden.
- Beispiel:

```
int *iptr = NULL;  // Zeiger mit NULL initialisiert
    …
// Überprüfung vor der Verwendung
if (iptr == NULL) {
    printf("Zeiger hat keine gültige Adresse\n");
    return EXIT_FAILURE;
}
// iptr hat eine gültige Adresse ...
```

- Bestimmte Zeiger (global, static) werden automatisch mit NULL initialisiert.
  - Wird noch ausführlich im Abschnitt Speicherklassen besprochen.

Interaktive Aufgabe

- Was gibt das nachfolgende Programm aus?

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {

  int *ptr = NULL;
  int ival;
  ptr = &ival;
  ival = 98765432;
  *ptr = 12345679;
  printf("%d\n", *ptr);
  printf("%d\n", ival);

  return EXIT_SUCCESS;
}
```

Speichergröße

- Die Größe eines Zeigers hängt nicht von dem Datentyp ab, auf den dieser Zeiger verweist.
  - Zeiger speichern Adressen.
- Beispiele
  - 32 Bit auf einem 32-Bit Rechner
  - 64 Bit auf 64-Bit-Architektur, 64-Bit-Betriebssystem und dem LP64 Datenmodell
- Typ bei einem Zeiger wird aber trotzdem benötigt!
  - Der Compiler kann Verletzungen der Typkompatibilität überprüfen.
    - Zuweisungen können auf Korrektheit überprüft werden.
    - Zum Beispiel kann man nicht direkt einen `int*` einem `double*` zuweisen!
  - Die Zeiger-Arithmetik wird dadurch realisiert.

C-Datenmodelle

- Auf 32-Bit Maschinen sind bei den meisten Entwicklungsumgebungen `int`, `long` und Zeiger 32-Bit groß.
- Auf 64-Bit Maschinen gibt es unterschiedliche Datenmodelle.
  - Beispiele
    - LP64: int = 32 Bits, long = 64 Bits, Zeiger = 64 Bits
    - ILP64: int = 64 Bits, long = 64 Bits, Zeiger = 64 Bits
- 64-Bit Datenmodelle

Datenmodell	`short`	`int`	`long`	`long long`	Zeiger/size_t	Beispiel für Betriebssystem
LLP64/IL32P64	16	32	32	64	64	Microsoft Windows (X64/IA-64)
LP64/I32LP64	16	32	64	64	64	Unix und Unix-ähnliche Systeme, z.B. Solaris, Linux, und Mac OS X
ILP64	16	64	64	64	64	Solaris auf SPARC64
SILP64	64	64	64	64	64	Unicos

- Die meisten Compiler orientieren sich an dem Modell des Betriebssystems.

Zeiger und Funktionsargumente

- C benutzt **call-by-value**
  - Argument wird dem formalen Parameter zugewiesen.
  - Es wird immer **eine Kopie** erstellt!
- Bei Zeigern
  - Man übergibt Zeiger und somit wird der Zeiger kopiert!
  - Übergabe von Zeigern (Adressen) als Argument wird **call-by-reference** genannt
  - Lokale Änderungen am dahinterliegenden Wert sind auch **außerhalb** der Funktion sichtbar.
    - Man braucht keinen Rückgabewert.
    - Man kann mehrere Zeigerparameter verwenden und damit mehrere Werte verändern.
  - Änderungen am Zeiger selbst sind aber nicht sichtbar!
  - Beispiel

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

Beispiel (Zeiger-Parameter)

```c
#include <stdio.h>
#include <stdlib.h>
void noswap(int, int);
void swap(int *, int *);

int main(void) {
	int a = 10;
	int b = 20;
	printf("a=%d b=%d\n", a, b);
	noswap(a, b);
	printf("a=%d b=%d\n", a, b);
	swap(&a, &b);
	printf("a=%d b=%d\n", a, b);
	return EXIT_SUCCESS;
}

void noswap(int px, int py) {
	int temp;
	temp = px;
	px = py;
	py = temp;
}

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

Ausgabe:

```
a=10 b=20
a=10 b=20
a=20 b=10
```

scanf (Wiederholung)

- Die Funktion `scanf` liest zeichenweise eine Folge von Eingabefeldern ein.
- Für jedes Eingabefeld muss eine Adresse vorhanden sein, wobei das Eingabefeld mit dem Datentyp des dahinterliegenden Wertes übereinstimmen muss.

```
scanf("%d", &i);
```

- Bei Erfolg liefert `scanf` die Anzahl der erfolgreich eingelesenen Felder zurück.
- Konnten keine Felder korrekt eingelesen werden, gibt `scanf` als Rückgabewert 0 zurück.
  - Für den Fall, dass bei der Eingabe schon ein Fehler auftrat, bevor die Daten überhaupt gelesen werden konnten, wird EOF zurückgegeben.

scanf – Probleme

- Wenn scanf Probleme bereitet:
  - `man scanf` ☺
- Das Verhalten von **`scanf`** hängt von der Umgebung (Betriebssystem) ab.
  - Ein häufiges Problem ist die Pufferung.
  - Diese ist je nach System und Anwendung zeilen- **oder** vollgepuffert.

scanf – Umgehen der Probleme

1. Benutzung von `fflush(stdin);`
   - Sollte nach jedem scanf-Aufruf benutzt werden.
   - Ist aber systemabhängig und produziert undefiniertes Verhalten!
   - **Nicht verwenden - schlecht!**
2. Benutzung einer Schleife, um verbleibende Zeichen aus dem Puffer herauszulesen
   - Auch in Kombination mit dem Rückgabewert von `scanf` möglich

```
while (scanf("%d %d %d", &a, &b, &c) != 3) {
    while (getchar() != '\n');
} …
```

3. Benutzung der Funktion `fgets`
   - Kompliziertere Variante mit `sscanf` zum Formatieren.

Arrays (Vektoren)

- Ein Array ist die Zusammenfassung von mehreren Variablen des gleichen Typs unter einem gemeinsamen Typ.
- Allgemeine Form
  - **`Typname Arrayname [GROESSE]`**
- Beispiele

```
int counter[5];
char letters[10];
```

- GROESSE ist eine positive ganze Zahl.
  - Konstante oder konstanter Integer-Ausdruck
- Array mit n Elementen
  - Indizierung beginnt bei 0 und endet bei n-1.
  - Beispiel: `counter[1] = 3; // 2. Element auf 3 setzen`

Zeiger und Arrays (1)

- Arrays und Zeiger sind **nicht** dasselbe.
- Ein Array belegt zum Programmstart automatisch einen Speicherbereich, der nicht mehr verschoben oder in der Größe verändert werden kann.
- Einem Zeiger muss man einen Wert zuweisen, damit er auf einen belegten Speicher zeigt.
- Ein Zeiger muss außerdem nicht nur auf den Anfang eines Speicherblocks zeigen.

Zeiger und Arrays (2)

- Der Name eines Arrays entspricht der Adresse des ersten Elements
  - Ist konstant während der Ausführung des Programms.
  - Kann anderen Zeigervariablen zugewiesen werden.
  - Kann für den Zugriff auf Elemente benutzt werden.

```
int alpha[5];
alpha[0] oder *alpha            // Zugriff auf das erste Element
alpha[i] oder *(alpha + i)      // Zugriff auf das i+1. Element
```

- Umgekehrt gilt das auch, d.h. die Zeigernotation ist gleichbedeutend zur Arraynotation.
- Der Compiler arbeitet intern immer zeigerbasiert.
- **ABER (siehe vorherige Folie)**
  - Einer Zeigervariable kann ein Wert zugewiesen werden.
  - Einem Arraynamen kann kein Wert zugewiesen werden (konstanter Zeiger!).

Gegenüberstellung Zeiger und Arrays

- Zugriff auf das erste Element

Zeiger-Variante	Array-Variante
`*ptr`	`ptr[0]`
`*array`	`array[0]`

- Zugriff auf das n-te Element

Zeiger-Variante	Array-Variante
`*(ptr+n)`	`ptr[n]`
`*(array+n)`	`array[n]`

- Zugriff auf die Anfangsadresse

Ohne Adressoperator	Mit Adressoperator
`ptr`	`&ptr[0]`
`array`	`&array[0]`

- Zugriff auf die Speicheradresse des n-ten Elements

Ohne Adressoperator	Mit Adressoperator
`ptr+n`	`&ptr[n]`
`array+n`	`&array[n]`

Zeigerarithmetik (1)

- Bei der Zeiger-Arithmetik geht es um die Verwendung von Operatoren für Zeiger und nicht um deren Werte.
- Addition und Subtraktion eines Zeigers mit einer Ganzzahl:
  - Operatoren: +, -, +=, -= , ++, --
  - `ptr = ptr + 2;`
    - Bedeutung: Zeiger `ptr` wurde jetzt um die **Länge von 2 Elementen des entsprechenden Datentyps** (Zeiger auf Datentyp) erhöht!
- Subtraktion von Zeiger:
  - Operatoren: -, -=
  - `ptr1 – ptr2` gibt die Anzahl der Elemente zwischen den Zeigern zurück.
    - In diesem Fall sollten `ptr1` und `ptr2` auf das gleiche Array zeigen.
      - Ansonsten kann es zu einem Absturz kommen.
      - Die Zeiger können auch auf die Adresse direkt hinter dem letzten Element eines Arrays zeigen.

Zeigerarithmetik (2)

- Vergleiche
  - Operatoren: ==, !=, <, <=, >, >=
  - Man vergleicht die Speicheradressen!
  - Zeiger können verglichen werden:
    - Wenn gleicher Typ.
    - Wenn ein Zeiger ein Nullzeiger ist.
- > (>=) bzw. < (<=) sind nur unter bestimmten Umständen (Verweis auf das gleiche Array) möglich!

Beispiel (Zeigerarithmetik)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int alpha[5] = { 3, 4, 5, 6, 7 };
    int *pointer, *endpointer;
    pointer = alpha;
    endpointer = alpha + 5;
    printf("%d\n", *(pointer + 1));
    printf("%d\n", *(endpointer - 4));
    printf("%d\n", endpointer - pointer);
    printf("%d\n", endpointer > pointer);
    return EXIT_SUCCESS;
}
```

Ausgabe:

```
4
4
5
1
```

Interaktive Aufgabe

- Was wird durch das folgende Programm ausgegeben?

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {

  int iarray[] = {12,34,56,78,90,23,45};
  int *ptr1, *ptr2;

  ptr1 = iarray;
  ptr2 = &iarray[4];
  ptr1 += 2;
  ptr2++;

  printf("%d\n", *ptr1);
  printf("%d\n", *ptr2);
  printf("%d\n", ptr2 - ptr1);
  printf("%d\n", (ptr1 < ptr2));
  printf("%d\n", ((*ptr1) < (*ptr2)));

  return EXIT_SUCCESS;
}
```

Array und Zeiger als Funktionsparameter

- Arrays werden nicht als Ganzes direkt an Funktionen übergeben.
  - Es wird immer nur die Adresse auf ein Array übergeben.
- Folgende zwei Funktionsköpfe sind daher **gleichbedeutend**:

```
void funktion(int arr[]) {…}
void funktion(int *arr) {…}
```

- Aufrufmöglichkeiten

```
int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int *iptr;
iptr = arr;
…
funktion(arr);
funktion(iptr);
funktion(&arr[2]);
```

Beispiel

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

void f1(int arr[], int n) {
	for (int i = 0; i < n; i++)
		printf("%d", arr[i]);
	printf("\n");
}

void f2(int *arr, int *end) {
	while (arr < end)
		printf("%d", *arr++);
	printf("\n");
}

int main(void) {
	int arr[MAX] = { 1, 2, 3, 4, 5 };
	int *iptr;
	iptr = arr;
	f1(arr, MAX);
	f1(iptr, MAX);
	f1(&arr[2], MAX-2);
	f2(arr, arr + MAX);
	f2(iptr, iptr + MAX);
	f2(&arr[2], arr + MAX);
	return EXIT_SUCCESS;
}
```

Ausgabe:

```
12345
12345
345
12345
12345
345
```

Zeichenketten (Strings) in C (Zeiger)

- In C werden Arrays **oder** auch Zeiger vom Datentyp `char` zum Speichern von Strings (Zeichenketten) verwendet.
  - Unterschiede ergeben sich aus dem Unterschied zwischen Arrays und Zeiger.
- Der C-Compiler markiert das Ende eines Strings mit dem Null-Zeichen `\0` (Byte, das nur 0-Bits enthält), damit das Ende der Zeichenkette erkannt werden kann.
- Der formale Parameter einer Funktion, der als eine Zeichenkette übergeben wird, kann vom Typ `char*` oder `char[]` sein.
  - In diesem Fall gibt es keinen Unterschied!

Beispiel (verschiedene Stringvarianten)

```c
#include <stdio.h>
#include <stdlib.h>

int string_length(const char what[]) {
	int i = 0;
	while (what[i]) i++;
	return i;
}

int main(void) {
	char buffer1[] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
	char buffer2[] = "Hello!";
	char *message = "Hello!";
	printf("Hello!\n");
	printf("%s\n", buffer1);
	printf("%s\n", buffer2);
	buffer2[1] = 'a';
	printf("%s\n", buffer2);
	printf("%s\n", message);
	message = "World!";
	printf("%s\n", message);
	//message[1] = 'u';     Undefiniertes Verhalten!
	//buffer2 = "World";    Nicht möglich!
	printf("String 1 length: %d\n", string_length(buffer1));
	printf("String 2 length: %d\n", string_length(buffer2));
	printf("String 3 length: %d\n", string_length(message));
	return EXIT_SUCCESS;
}
```

Zeiger auf konstante Zeichenkette

Ausgabe:

```
Hello!
Hello!
Hello!
Hallo!
Hello!
World!
String 1 length: 6
String 2 length: 6
String 3 length: 6
```

Interaktive Aufgabe

- Was ist der Unterschied zwischen den folgenden beiden Initialisierungen?

```
char str1[] = "Hallo";
char *str2 = "Hallo";
```

Zeigerarrays

- Arrays von Zeiger sind möglich

```
int *counters[10]; // Array von 10 Integer-Zeiger
```

- Solche Zeigertypen werden bevorzugt als Alternative für zweidimensionale Arrays und hier ganz besonders gerne bei zweidimensionalen Stringarrays eingesetzt.
- Folgende Arrays haben die gleiche Funktionalität

```
char numbers[10][50] = { "eins", "zwei", "drei", "vier",
"fünf", "sechs", "sieben", "acht", "neun", "zehn" };
char *numbers2[10] = { "eins", "zwei", "drei", "vier",
"fünf", "sechs", "sieben", "acht", "neun", "zehn" };
```

- Unterschied besteht hier nur in der Speicherbelegung.
  - Für `numbers` werden 500 Bytes reserviert, egal wie groß die Strings sind.
  - Bei `numbers2` wird für jeden String der dafür benötigte Speicherplatz verwendet.

Mehrdimensionale Arrays und Zeiger

- Sei folgendes Array gegeben
  - `int arr[4][2];`
- Dann gilt
  - `arr` ist die Adresse der ersten Zeile des Arrays.
  - `arr + 2` ist die Adresse der dritten Zeile des Arrays.
    - Hier werden 2 Elemente dazugezählt.
    - In diesem Fall sind die Elemente aber selbst Arrays!
  - `*(arr + 2)` ist die Adresse des ersten Elements in der dritten Zeile.
    - Entspricht `arr + 2`.
  - `*(arr + 2) + 1` ist die Adresse des zweiten Elements in der dritten Zeile.
  - `*(*(arr + 2) + 1)` ist der Wert des zweiten Elements in der dritten Zeile.

Beispiel

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char *numbers[] = { "eins", "zwei", "drei", "vier", "fuenf", "sechs", "sieben",
    "acht", "neun", "zehn", NULL };
    for (int i = 0; numbers[i] != NULL; i++) {
        printf("%2d : %s\n", i, numbers[i]);
    }
    // Zugriff auf 2. Buchstaben im 2.String
    printf("%c", numbers[1][1]); // = w
    // Zugriff auf 4.Buchstaben im 3.String
    printf("%c", *(numbers[2] + 3)); // = i
    // Zugriff auf 3.Buchstaben im 4.String
    printf("%c\n", *(*(numbers + 3) + 2)); // = e
    return EXIT_SUCCESS;
}
```

Ausgabe:

```
 0 : eins
 1 : zwei
 2 : drei
 3 : vier
 4 : fuenf
 5 : sechs
 6 : sieben
 7 : acht
 8 : neun
 9 : zehn
wie
```

Zeiger auf Arrays

- Beispiel für das Anlegen eines Zeigers auf ein Array:
  - `int (*ptr)[10];`
  - Ist ein Zeiger `ptr`, der auf ein Array mit 10 Integern zeigt.
  - Ohne die Klammern hätte man ein Array mit 10 Integer-Zeiger!
- Verwendung bei zweidimensionalen Arrays
  - Folgende Ausgabe liefert: 1 1 1 6 6

```
int arr[4][2] = { {1,2}, {3,4}, {5,6}, {7,8} };
int (*pz) [2];
pz = arr;
printf("%d %d %d %d %d", pz[0][0], *pz[0], **pz, pz[2][1], *(*(pz + 2) + 1));
```

- Hinweis zu Arrays
  - In C gibt es eigentlich nur eindimensionale Arrays.
  - Ein Element eines Arrays darf aber ein Speicherobjekt von einem beliebigen Typ (auch Array) sein.
  - Mehrdimensionale Arrays werden daher nur simuliert.

Zeiger auf Zeiger

- Zeiger auf Zeiger
  - Man hat einen Zeiger, der auf einen Zeiger zeigt, der auf eine Variable zeigt.
  - Es ist auch möglich Zeiger auf Zeiger auf Zeiger usw. zu verwenden.
- Beispiele

  `char **textpointer;`
  - Zeiger auf Zeiger – Zeigervariablenadresse
  - Zeiger – Variablenadresse
  - Variable – Wert

  `char **textpointer` und `char *textpointer[]` sind äquivalent als Parameter.
- Haupteinsatzgebiet ist die dynamische Erzeugung von mehrdimensionalen Arrays.

Zeigerkompatibilität

- Restriktionen bei Zuweisungen

```
int n = 5;
double x;
int *p1 = &n;
double *pd = &x;
x = n;
pd = p1;          // Fehler !
```

- Komplexeres Beispiel

```
int *pt;
int (*pa)[3];
int arr1[2][3];
int arr2[3][2];
int **p2;
…
pt = &arr1[0][0];
pt = arr1[0];
pt = arr1;        // Fehler !
pa = arr1;
pa = arr2;        // Fehler !
p2 = &pt;
*p2 = arr2[0];
p2 = arr2;        // Fehler !
```

Mehrdimensionale Arrays bei Funktionen

- Folgende Deklarationen sind äquivalent

```
void  f(int (*p)[4]){ }
void  f(int p[][4]){ }
```

  - Die leeren Klammern [] haben in diesem Fall die gleiche Bedeutung wie der Zeiger.
    - Es könnte auch eine Zahl angegeben werde – diese hat aber keine Auswirkung!
  - Die nachfolgenden Klammern dürfen nicht leer bleiben.
    - Größe der Arrayelemente muss bekannt sein!

- Beispiel

```
int sum2d(int ar[][COLS], int rows) {
        int tot = 0;
        for (int r = 0; r < rows; r++)
                for (int c = 0; c < COLS; c++)
                        tot += ar[r][c];
        return tot;
}
```

Zeiger auf void

- Wenn man den Typ der Variable, auf die der Zeiger verweisen soll, noch nicht kennt, dann vereinbart man einen Zeiger auf den Typ **`void`**.
- Solch ein Zeiger kann nicht dereferenziert werden.
- Später kann man den Zeiger in einen Zeiger auf einen bestimmten Typ umwandeln.
- Ein Zeiger auf **`void`** ist ein untypisierter (generischer) Zeiger, der zu allen anderen Zeigertypen kompatibel ist.
  - Zuweisung zwischen **`void`**-Zeiger und typisiertem Zeiger ist möglich.
  - Umgeht aber die Typprüfung!

Beispiel

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    float fval = 123.456f;
    int ival = 789;
    char *cval = "Hello World";
    void *vptr;
    vptr = &fval;
    printf("%.3f\n", *(float *) vptr);
    vptr = &ival;
    printf("%d\n", *(int *) vptr);
    vptr = cval;
    printf("%s\n", (char *) vptr);
    return EXIT_SUCCESS;
}
```

Ausgabe:

```
123.456
789
Hello World
```

Typ-Qualifizierer bei Zeiger (1)

- `const`
  - Mit diesem Qualifizierer wird angezeigt, dass etwas konstant ist und nach der Definition nicht mehr verändert werden kann.
    - Eine einzige Zuweisung kann erfolgen.
- Unterschiedliche Anwendungen
  - Konstanter Zeiger (const steht rechts vom Stern)
    - `int * const c_iptr1;`
    - Zeigt immer auf die gleiche Adresse.
    - Der referenzierte Wert darf aber verändert werden.
  - Zeiger auf konstante Daten (const steht links vom Stern, kann auch vor dem Datentyp stehen)
    - `int const *c_iptr2;`
    - Zeiger darf verändert werden, die Daten aber nicht!
    - Konstante Parameter für Funktionen
      - Damit kann die Funktion die Daten nicht überschreiben.
      - Beispiel in <stdio.h>: `printf (const char*, ...);`
  - Konstante Zeiger auf konstante Daten
    - Zweimal const verwenden (links und rechts vom Stern).

Beispiel (const bei Zeiger)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int ivar1 = 0;
    int ivar2 = 0;
    int * const c_iptr1 = &ivar1;
    *c_iptr1 = 1234;
    // c_iptr1++; // Nicht erlaubt!
    int const *c_iptr2 = &ivar2;
    c_iptr2 = c_iptr1;
    //*c_iptr2 = 2345; // Nicht erlaubt;
    int const * const ccp = &ivar1;
    printf("%d\n%d\n%d\n%d\n%d\n", ivar1, *c_iptr1, ivar2, *c_iptr2, *ccp);
    return EXIT_SUCCESS;
}
```

Ausgabe:

```
1234
1234
0
1234
1234
```

Beispiel (const bei Zeigerzuweisungen)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    double rates[5] = {10.1, 20.23, 15.12, 17.23, 33.22};
    const double locked[4] = {0.05, 0.55, 0.23, 0.05};
    const double *pc = rates;
    pc = locked;
    printf("%f\n", pc[1]);
    pc = &rates[3];
    printf("%f\n", pc[1]);
    double *pnc = rates;
    printf("%f\n", pnc[1]);
    //pnc = locked;        // Nicht erlaubt!
    pnc = &rates[3];
    printf("%f\n", pnc[1]);
    return EXIT_SUCCESS;
}
```

Ausgabe:

```
0.550000
33.220000
20.230000
33.220000
```

Typ-Qualifizierer bei Zeiger (2)

- `volatile`
  - Bei einer volatile-Variable wird vor jedem Zugriff der Wert neu aus dem Hauptspeicher eingelesen.
  - Compiler wird dann keine Optimierungen vornehmen (z.B. Caching).
- `restrict` (nur in C99)
  - Mit dem Qualifizierer wird angegeben, dass der Zugriff **nur über** diesen **`restrict`**-Zeiger erfolgt (es darf keinen anderen Zeiger auf das gleiche Objekt geben).
    - Verantwortung liegt beim Programmierer.
    - Compiler kann Optimierungen vornehmen.
    - Kann auch bei Funktionen verwendet werden, um anzuzeigen, dass sich zwei Zeiger in der Parameterliste nicht überlappen dürfen (siehe `string.h`).

Parameter für die main-Funktion (1)

- Der `main`-Funktion können Parameter übergeben werden.
- Dafür sind zwei formale Parameter vorgesehen.
- Typische Namen:
  - `argc` (argument counter) – Typ `int`
  - `argv` (argument vector) – Typ `char *[]` oder `char **`
  - Beispiel
    - `int main (int argc, char *argv[]) {…}`
- `argc`
  - Wenn größer als 1, dann enthalten `argv[1]` bis `argv[argc - 1]` die Programmparameter.
- `argv`
  - Array von Zeiger auf `char`.
  - Erster Zeiger zeigt meist auf den Programmnamen (wenn argc > 0).
    - Bei manchen Betriebssystemen wird der Name nicht gesetzt!

Parameter für die main-Funktion (2)

- Alle Parameter sind zunächst Zeichenketten.
- Umwandlung erfolgt durch entsprechende Funktionen
  - Veraltete Funktionen: `atoi, atol, atof`
  - Neuere Funktionen: `strtol, strtoul, strtod`
  - Beispiel strtol
    - `long int strtol(const char *nptr, char **endptr, int base);`
    - Konvertiert den Anfangsteil von `nptr` in einen Long-Wert zur Basis `base`.
    - Wenn `endptr != NULL`, dann enthält `endptr` die Adresse des erste Zeichens, das nicht konvertiert werden konnte.
      - Wenn keine Zeichen vorhanden waren, dann enthält `endptr` den Wert von `nptr` und `strtol` gibt 0 zurück.
    - man-Seiten lesen!

Beispiel (Parameter, Verhalten von strtol)

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    char *end;
    printf("Insgesamt %d Argumente\n", argc - 1);
    for (int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
        printf("%ld\n", strtol(argv[i], &end, 10));
    }
    return EXIT_SUCCESS;
}
```

Aufruf: ./test Hallo 1Welt 123
Ausgabe:

```
Insgesamt 3 Argumente
argv[0] = ./test
0
argv[1] = Hallo
0
argv[2] = 1Welt
1
argv[3] = 123
123
```

Zeiger auf Funktionen

- In C ist ein Funktionsname eine Adresskonstante
  - Der Funktionsname zeigt auf den ersten Maschinenbefehl der Funktion.
- Man kann einen Zeiger auf eine Funktion vereinbaren
  - Durch Dereferenzierung kann man die Funktion aufrufen (z.B. zur Laufzeit bestimmen, welche Funktion ausgeführt wird).
  - Über einen Zeiger können Funktionen auch als Parameter an andere Funktionen übergeben werden.
- Vereinbarung (Beispiel)

```
int (*ptr) (char);
```

  - Zeiger auf eine Funktion mit einem Rückgabewert vom Typ `int` und einem Übergabeparameter vom Typ `char`.
- Beispiel (Ausgabe: *Hallo Welt mit Funktionszeiger*)

```
int (*fptr)(const char*, ...);
fptr = printf;
(*fptr)("Hallo Welt mit Funktionszeiger\n");
```

Funktionszeiger und Arrays

- Zeiger auf Funktionen können auch in einem Array gespeichert werden.
- Die einzelnen Funktionen können dann über den Index aufgerufen werden.
- Siehe nachfolgendes Beispiel!

Beispiel (Funktionszeiger)

```c
#include <stdio.h>
#include <stdlib.h>

double addition(double x, double y) { return x + y; }
double subtraction(double x, double y) { return x - y; }
double multiplication(double x, double y) { return x * y; }
double division(double x, double y) { return x / y; }

int main(void) {
	double (*fptr[4])(double d1, double d2) = {addition, subtraction, multiplication, division};
	double v1, v2;
	int operator;
	printf("Number 1 and 2:-> ");
	scanf("%lf %lf", &v1, &v2);
	printf("Choose operator:\n");
	printf("0 = +\n1 = -\n2 = *\n3 = /\n");
	printf("Your choice: ");
	scanf("%d", &operator);
	if (!(operator >= 0 && operator <= 3)) {
		printf("Error: Wrong operator!\n");
		return EXIT_FAILURE;
	}
	printf("Result: %f\n", fptr[operator](v1, v2));
	return EXIT_SUCCESS;
}
```

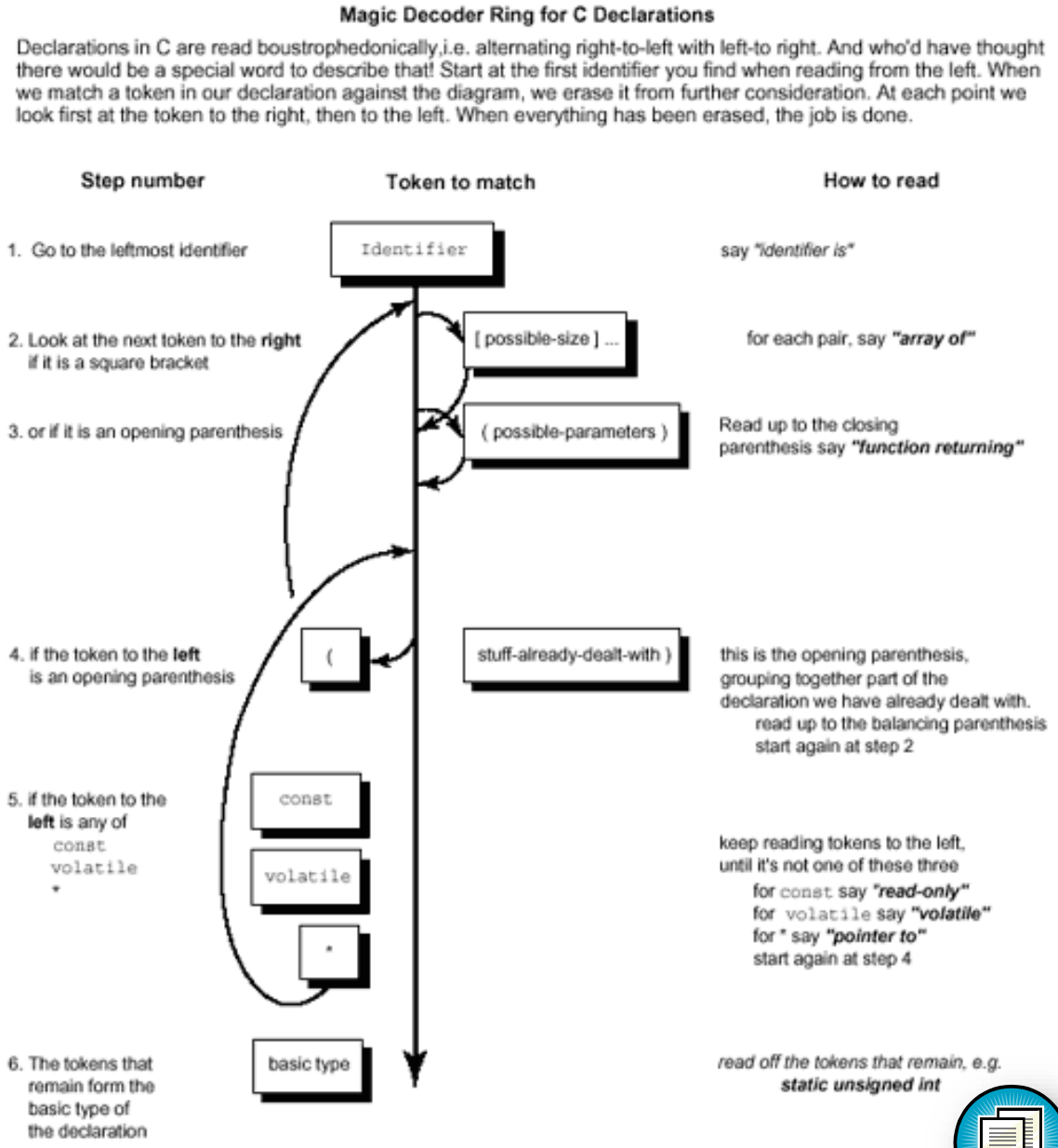
Beispiel für Funktionszeiger (qsort in stdlib.h)

- **`qsort`** in **`stdlib.h`**

```
void qsort(
   void *array,         // Anfangsadresse des Arrays
   size_t n,            // Anzahl der Elemente zum Sortieren
   size_t size,         // Größe des Datentyps, der sortiert wird
   int (*vergleich_func)(const void*, const void*) ); // Vergleichsfunktion
```

- Quicksort-Algorithmus
  - Sortiert ein Array.
  - Man muss eine Vergleichsfunktion zur Verfügung stellen.
  - Diese Vergleichsfunktion wird abhängig vom Typ die entsprechenden Vergleiche durchführen.
  - Weitere Informationen: `man 3 qsort`

Wie „dekodiert“ man C Deklarationen



[van der Linden 94]

Beispiel 1

- Was bedeutet `char * const *(*next)();` ?

Deklaration (verbleibend)	Nächster Schritt	Resultat
`char * const *(*next)();`	1	*next ist ein …*
`char * const *(*)();`	2, 3	Keine Übereinstimmung, nächster Schritt
`char * const *(*)();`	4	Keine Übereinstimmung, nächster Schritt
`char * const *(*)();`	5	*Zeiger auf …*, gehe zu Schritt 4
`char * const *()();`	4	(gehört zu), gehe zu Schritt 2
`char * const * () ;`	2	Keine Übereinstimmung, nächster Schritt
`char * const * () ;`	3	*Funktion, die zurückliefert …*
`char * const * ;`	4	Keine Übereinstimmung, nächster Schritt
`char * const * ;`	5	*Zeiger auf …*
`char * const ;`	5	*Read-only …*
`char * ;`	5	*Zeiger auf …*
`char ;`	6	*char*

- Daher: next ist ein Zeiger auf eine Funktion, die einen Zeiger auf einen konstanten Zeiger auf char zurückliefert.

Beispiel 2

- `char *(*c[10])(int **p);`

Deklaration (verbleibend)	Nächster Schritt	Resultat
`char *(*c[10])(int **p);`	1	*c ist ein …*
`char *(*[10])(int **p);`	2	*Array[0..9] von …*
`char *(*)(int **p);`	5	*Zeiger auf …* , gehe zu Schritt 4
`char *() (int **p);`	4	() entfernen, gehe zu Schritt 2 und dann 3
`char * (int **p) ;`	3	*Funktion, die zurückliefert …*
`char * ;`	5	*Zeiger auf …*
`char ;`	6	*char*

- c ist ein Array[0..9] von Zeiger auf jeweils eine Funktion, die einen Zeiger auf char zurückliefert.
- Hinweis: Die Funktion hat einen formalen Parameter vom Typ Zeiger auf Zeiger auf Integer!

Literatur

- [van der Linden 94] Peter van der Linden, **Expert C Programming – Deep C Secrets**, Prentice Hall, 1994