Vektoren

⟨direct-declarator⟩	$\longrightarrow$	⟨simple-declarator⟩
	$\longrightarrow$	„(“ ⟨simple-declarator⟩ „)“
	$\longrightarrow$	⟨function-declarator⟩
	$\longrightarrow$	⟨array-declarator⟩
⟨array-declarator⟩	$\longrightarrow$	⟨direct-declarator⟩ „[“ [⟨array-qualifier-list⟩] [⟨array-size-expression⟩] „]“
⟨array-qualifier-list⟩	$\longrightarrow$	⟨array-qualifier⟩
	$\longrightarrow$	⟨array-qualifier-list⟩ ⟨array-qualifier⟩
⟨array-qualifier⟩	$\longrightarrow$	**static**
	$\longrightarrow$	**restrict**
	$\longrightarrow$	**const**
	$\longrightarrow$	**volatile**
⟨array-size-expression⟩	$\longrightarrow$	⟨assignment-expression⟩
	$\longrightarrow$	„*“
⟨simple-declarator⟩	$\longrightarrow$	⟨identifier⟩

Deklaration von Vektoren

- Wie bei den Zeigertypen erfolgen die Typspezifikationen eines Vektors nicht im Rahmen eines ⟨type-specifier⟩.
- Stattdessen gehört eine Vektordeklaration zu dem ⟨init-declarator⟩. Das bedeutet, dass die Präzisierung des Typs zur genannten Variablen unmittelbar gehört.
- Entsprechend deklariert

```
int a[10], i;
```

  eine Vektorvariable *a* und eine ganzzahlige Variable *i*.

Vektoren und Zeiger

- Vektoren und Zeiger sind eng miteinander verwandt.
- Der Variablenname eines Vektors ist ein konstanter Zeiger auf den zugehörigen Element-Typ, der auf das erste Element verweist.
- Allerdings liefert **sizeof** mit dem Vektornamen als Operand die Größe des gesamten Vektors und nicht etwa nur die des Zeigers.

Vektoren und Zeiger

array.c

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

int main() {
   int a[] = {1, 2, 3, 4, 5};
   /* Groesse des Arrays bestimmen */
   const size_t SIZE = sizeof(a) / sizeof(a[0]);
   int* p = a;  /* kann statt a verwendet werden */
   /* aber: a weiss noch die Gesamtgroesse, p nicht */
   printf("SIZE=%zd, sizeof(a)=%zd, sizeof(p)=%zd\n",
      SIZE, sizeof(a), sizeof(p));
   for (int i = 0; i < SIZE; ++i) {
      *(a + i) = i+1; /* gleichbedeutend mit a[i] = i+1 */
   }
   /* Elemente von a aufsummieren */
   int sum = 0;
   for (int i = 0; i < SIZE; i++) {
      sum += p[i];   /* gleichbedeutend mit ... = a[i]; */
   }
   printf("Summe: %d\n", sum);
}
```

Indizierung

- Grundsätzlich beginnt die Indizierung bei 0.
- Ein Vektor mit 5 Elementen hat entsprechend zulässige Indizes im Bereich von 0 bis 4.
- Wird mit einem Index außerhalb des zulässigen Bereiches zugegriffen, so ist der Effekt undefiniert.
- Es ist dann damit zu rechnen, dass irgendeine andersweitig belegte Speicherfläche adressiert wird oder es zu einer harten Unterbrechung kommt, weil eine unzulässige Adresse dereferenziert wurde. Was tatsächlich passiert, hängt von der jeweiligen Adressraumbelegung ab.
- Viele bekannte Sicherheitslücken beruhen darauf, dass in C-Programmen die zulässigen Indexbereiche verlassen werden und auf diese Weise eingeschleuster Programmtext zur Ausführung gebracht werden kann.
- Anders als in Modula-2, Oberon oder Java gibt es aber keine automatisierte Überprüfung. Diese wäre auch wegen der Verwandtschaft von Vektoren und Zeigern nicht mit einem vertretbaren Aufwand in C umzusetzen.

Parameterübergabe bei Vektoren

- Da der Name eines Vektors nur ein Zeiger auf das erste Element ist, werden bei der Parameterübergabe entsprechend nur Zeigerwerte übergeben.
- Entsprechend arbeitet die aufgerufene Funktion nicht mit einer Kopie des Vektors, sondern hat dank dem Zeiger den direkten Zugriff auf den Vektor des Aufrufers.
- Die Dimensionierung des Vektors muss explizit mit Hilfe weiterer Parameter übergeben werden, wenn diese variabel sein soll.

Parameterübergabe bei Vektoren

array2.c

```
#include <stdio.h>

const int SIZE = 10;

/* Array wird veraendert, naemlich mit
   0, 1, 2, 3, ... initialisiert! */
void init(int a[], int length) {
   for (int i = 0; i < length; i++) {
      a[i] = i;
   }
}

int summe1(int a[], int length) {
   int sum = 0;
   for (int i = 0; i < length; i++) {
      sum += a[i];
   }
   return sum;
}
```

Parameterübergabe bei Vektoren

array2.c

```
int summe2(int* a, int length) {
   int sum = 0;
   for (int i = 0; i < length; i++) {
      sum += *(a+i); /* aequivalent zu ... += a[i]; */
   }
   return sum;
}

int main() {
   int array[SIZE];

   init(array, SIZE);

   printf("Summe: %d\n", summe1(array, SIZE));
   printf("Summe: %d\n", summe2(array, SIZE));
}
```

Mehrdimensionale Vektoren

- So könnte ein zweidimensionaler Vektor angelegt werden:

```
int matrix[2][3];
```

- Eine Initialisierung ist sofort möglich. Die geschweiften Klammern werden dann entsprechend verschachtelt:

```
int matrix[2][3] = {{0, 1, 2}, {3, 4, 5}};
```

Repräsentierung eines Vektors im Speicher

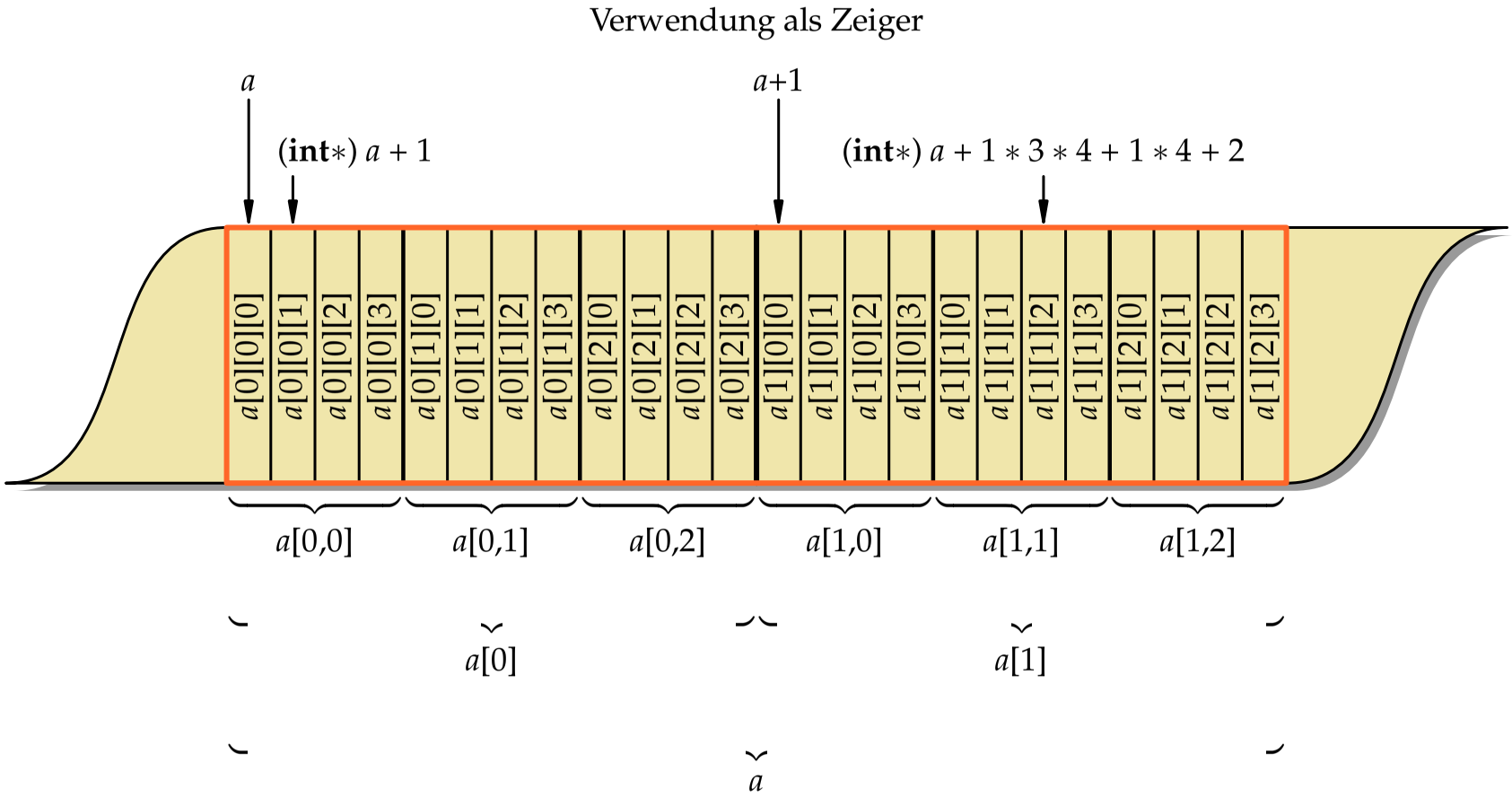
Angenommen die Anfangsadresse des Vektors liege bei 0*x1000* und eine ganze Zahl vom Typ **int** würde vier Bytes belegen, dann würde die Repräsentierung des Vektors *matrix* im Speicher folgendermaßen aussehen:

Element	**Adresse**	**Inhalt**
matrix[0][0]	0x1000	0
matrix[0][1]	0x1004	1
matrix[0][2]	0x1008	2
matrix[1][0]	0x100C	3
matrix[1][1]	0x1010	4
matrix[1][2]	0x1014	5

Repräsentierung eines Vektors im Speicher

- Gegeben sei:

```
int a[2][3][4];
```



Parameterübergabe mehrdimensionaler Vektoren

Folgende Möglichkeiten stehen zur Verfügung:

- Alle Dimensionen mit Ausnahme der ersten werden explizit bei der Parameterdeklaration festgelegt. Nur die erste Dimension ist dann noch variabel.
- Der gesamte Vektor wird zu einem eindimensionalen Vektor verflacht. Eine mehrdimensionale Indizierung erfolgt dann „per Hand“.
- Beginnend mit C99 gibt es auch mehrdimensionale dynamische Parameterübergaben von Vektoren. Dies ist analog zu den offenen mehrdimensionalen Feldparametern in Oberon. Im Unterschied zu Oberon müssen die Dimensionierungsparameter jedoch explizit benannt werden.

Zeichenketten

- Zeichenketten werden in C als Vektoren von Zeichen repräsentiert: **char**[]
- Das Ende der Zeichenkette wird durch ein sogenanntes Null-Byte ('\0') gekennzeichnet.
- Da es sich bei Zeichenketten um Vektoren handelt, werden bei der Parameterübergabe nur die Zeiger als Werteparameter übergeben.
- Die Zeichenkette (also der Inhalt des Vektors) kann entsprechend von der aufgerufenen Funktion verändert werden.

Zeichenketten-Konstanten

- Zeichenketten-Konstanten können durch von Doppelapostrophen eingeschlossene Zeichenfolgen spezifiziert werden. Hier im Rahmen einer Initialisierung:

```
char greeting[] = "Hallo";
```

- Dies ist eine Kurzform für

```
char greeting[] = {'H', 'a', 'l', 'l', 'o', '\0'};
```

- Eine Zeichenketten-Konstante steht für einen Zeiger auf den Anfang der Zeichenkette:

```
char* greeting = "Hallo";
```

- Zeichenketten-Konstanten dürfen nicht verändert werden. Sie werden, falls die zugrundeliegende Architektur dies ermöglicht, in einem Speicherbereich abgelegt, der nur Lesezugriffe zulässt.

Zeichenketten: Was ist zulässig?

strings.c

```
#include <stdio.h>

int main() {
   char array[10];
   char string[] = "Hallo!"; /* Groesse wird vom Compiler bestimmt */
   char* s1 = "Welt";
   char* s2;

   /* array = "not OK"; */   /* nicht zulaessig */
   array[0] = 'A';           /* zulaessig */
   array[1] = '\0';
   printf("array: %s\n", array);
   /* s1[5] = 'B'; */        /* nicht zulaessig */
   s1 = "ok";                /* zulaessig */
   printf("s1: %s\n", s1);
   s2 = s1;                  /* zulaessig */
   printf("s2: %s\n", s2);
   string[0] = 'X';          /* zulaessig */
   printf("string: %s\n", string);
   printf("sizeof(string): %zd\n", sizeof(string));
}
```

Länge einer Zeichenkette

strings1.c

```
/* Laenge einer Zeichenkette bestimmen */
int my_strlen1(char s[]) {
   int i;
   /* bis zum abschliessenden Null-Byte laufen */
   for (i = 0; s[i] != '\0'; i++); /* leere Anweisung! */
   return i;
}
```

- Die Bibliotheksfunktion *strlen()* liefert die Länge einer Zeichenkette zurück.
- Als Länge einer Zeichenkette wird die Zahl der Zeichen *vor* dem Null-Byte betrachtet.
- *my_strlen1()* bildet hier die Funktion nach unter Verwendung der vektoriellen Notation.

Länge einer Zeichenkette

strings1.c

```
/* Laenge einer Zeichenkette bestimmen */
int my_strlen2(char* s) {
   char* t = s;
   while (*t++);
   return t-s-1;
}
```

- Alternativ wäre es auch möglich, mit der Zeigernotation zu arbeiten.
- Zu beachten ist hier, dass der Post-Inkrement-Operator $++$ einen höheren Rang hat als der Dereferenzierungs-Operator $*$.
- Entsprechend bezieht sich das Inkrement auf t. Das Inkrement wird aber erst *nach* der Dereferenzierung als verspäteter Seiteneffekt ausgeführt.

Kopieren einer Zeichenkette

strings1.c

```
/* Kopieren einer Zeichenkette von s nach t
   Vorauss.: genuegend Platz in t */
void my_strcpy1(char t[], char s[]) {
   for (int i = 0; (t[i] = s[i]) != '\0'; i++);
}
```

- Das ist ein Nachbau der Bibliotheksfunktion *strcpy()* die (analog zur Anordnung bei einer Zuweisung) den linken Parameter als Ziel und den rechten Parameter als Quelle der Kopier-Aktion betrachtet.
- Hier zeigt sich auch eines der grossen Probleme von C im Umgang mit Vektoren: Da die tatsächlich zur Verfügung stehende Länge des Vektors *t* unbekannt bleibt, können weder *my_strcpy1()* noch die Laufzeitumgebung dies überprüfen.

Kopieren einer Zeichenkette

strings1.c

```
/* Kopieren einer Zeichenkette von s nach t
   Vorauss.: genuegend Platz in t */
void my_strcpy2(char* t, char* s) {
   for (; (*t = *s) != '\0'; t++, s++);
}
```

- In der Zeigernotation wird es einfacher.

Kopieren einer Zeichenkette

strings1.c

```
/* Kopieren einer Zeichenkette von s nach t
   Vorauss.: genuegend Platz in t */
void my_strcpy3(char* t, char* s) {
   while ((*t++ = *s++) != '\0');
}
```

- Die Inkrementierung lässt sich natürlich (wie schon bei der Längenbestimmung) mit integrieren.

Kopieren einer Zeichenkette

strings1.c

```
/* Kopieren einer Zeichenkette von s nach t
   Vorauss.: genuegend Platz in t */
void my_strcpy4(char* t, char* s) {
   while ((*t++ = *s++));
}
```

- Der Vergleichstest mit dem Nullbyte lässt sich natürlich streichen.
- Allerdings gibt es dann eine Warnung des gcc, dass möglicherweise der Vergleichs-Operator == mit dem Zuweisungs-Operator = verwechselt worden ist.
- Diese Warnung lässt sich (per Konvention) durch die doppelte Klammerung unterdrücken. Damit wird klar lesbar zum Ausdruck gegeben, dass das kein Versehen ist.

Vergleich zweier Zeichenketten

strings1.c

```
/* Vergleich zweier Zeichenketten
   Ergebnis: 0 fuer s = t, > 0 fuer s > t und < 0 fuer s < t */
int my_strcmp1(char s[], char t[]) {
   int i;
   for (i = 0; s[i] == t[i] && s[i] != '\0'; i++);
   return s[i] - t[i];
}
```

- Um alle sechs Vergleichsrelationen mit einer Funktion unterstützen zu können, arbeitet die Bibliotheksfunktion *strcmp()* mit einem ganzzahligen Rückgabewert, der < 0 ist, falls $s < t$, $= 0$ ist, falls s mit t übereinstimmt und > 0, falls $s > t$.

Vergleich zweier Zeichenketten

strings1.c

```
/* Vergleich zweier Zeichenketten
   Ergebnis: 0 fuer s = t, > 0 fuer s > t und < 0 fuer s < t */
int my_strcmp2(char* s, char* t) {
   for (; *s == *t && *s != '\0'; s++, t++);
   return *s - *t;
}
```

- Auch dies lässt sich in die Zeigernotation umsetzen.
- Auf ein integriertes Post-Inkrement wurde hier verzichtet, da dann die beiden Zeiger eins zu weit stehen, wenn es darum geht, die Differenz der unterschiedlichen Zeichen zu berechnen.