Systemnahe Software
WS 2006/2007

Andreas F. Borchert
Universität Ulm

27. November 2006

Zeigerarithmetik

- Es ist zulässig, ganze Zahlen zu einem Zeiger zu addieren oder davon zu subtrahieren.
- Dabei wird jedoch der zu addierende oder zu subtrahierende Wert implizit mit der Größe des Typs multipliziert, auf den der Zeiger zeigt.
- Weiter ist es zulässig, Zeiger des gleichen Typs voneinander zu subtrahieren. Das Resultat wird dann implizit durch die Größe des referenzierten Typs geteilt.

Zeigerarithmetik

zeiger1.c

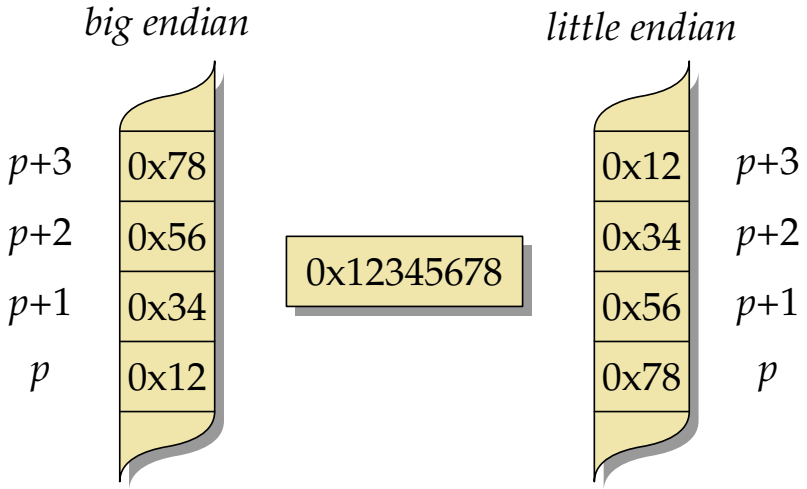
```
#include <stdio.h>

int main() {
   unsigned int value = 0x12345678;
   unsigned char* p = (unsigned char*) &value;

   for (int i = 0; i < sizeof(unsigned int); ++i) {
      printf("p+%d --> 0x%02hhx\n", i, *(p+i));
   }
}
```

- Hier wird der Speicher byteweise „durchleuchtet“.
- Hierbei fällt auf, dass die interne Speicherung einer ganzen Zahl bei unterschiedlichen Architekturen (SPARC vs. Intel x86) verschieden ist: *big endian* vs. *little endian*.

big vs. little endian



- Bei *little endian* wird das niedrigstwertige Byte an der niedrigsten Adresse abgelegt, während bei
- *big endian* das niedrigstwertige Byte sich bei der höchsten Adresse befindet.

Typkonvertierungen

- Typ-Konvertierungen können in C sowohl implizit als auch explizit erfolgen.
- Implizite Konvertierungen werden angewendet bei Zuweisungs-Operatoren, Parameterübergaben und Operatoren. Letzteres schliesst auch die monadischen Operatoren mit ein.
- Explizite Konvertierungen erfolgen durch den sogenannten Cast-Operator.

Konvertierungen bei numerischen Datentypen

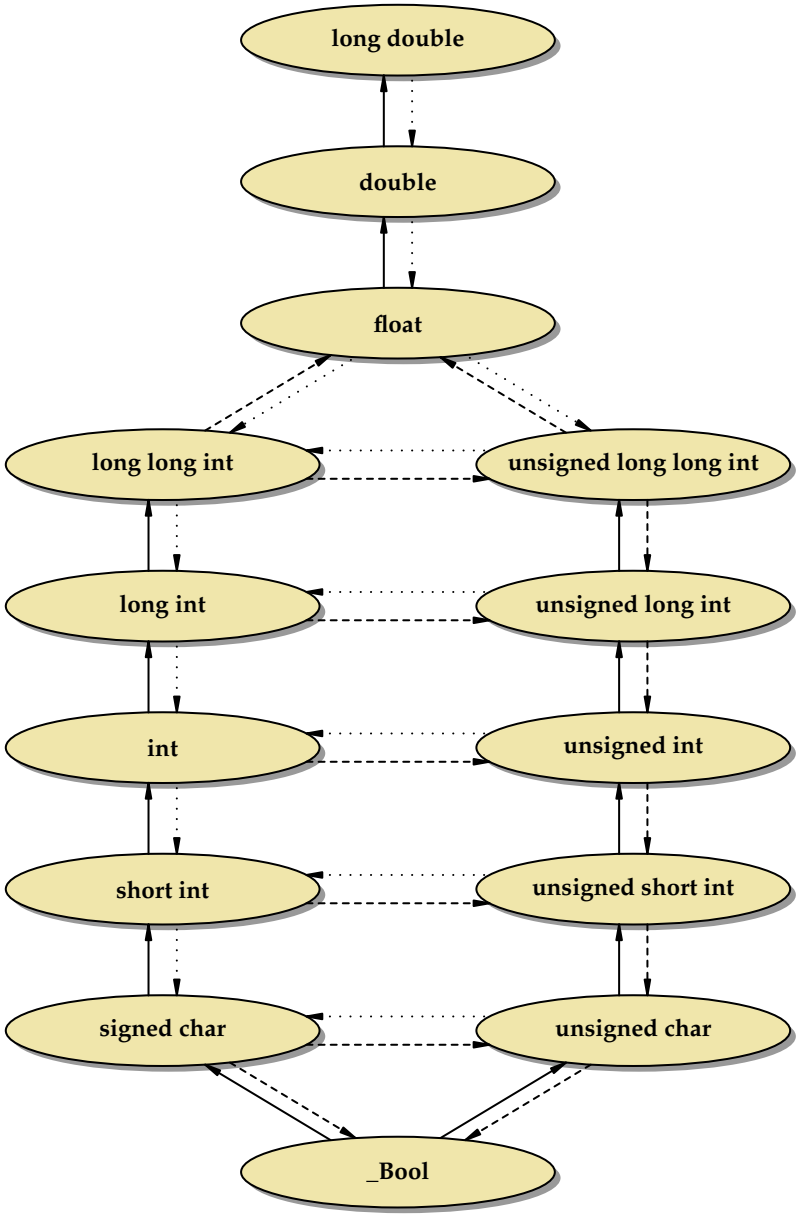
Bei einer Konvertierung zwischen numerischen Typen gilt der Grundsatz, dass – wenn irgendwie möglich – der Wert zu erhalten ist. Falls das jedoch nicht möglich ist, gelten folgende Regeln:

- Bei einer Konvertierung eines vorzeichenbehafteten ganzzahligen Datentyps zum Datentyp ohne Vorzeichen *gleichen Ranges* (also etwa von **int** zu **unsigned int**) wird eine ganze Zahl $a < 0$ zu b konvertiert, wobei gilt, dass $a \bmod 2^n = b \bmod 2^n$ mit n der Anzahl der verwendeten Bits, wobei hier der mod-Operator entsprechend der F-Definition bzw. Euklid gemeint ist. Dies entspricht der Repräsentierung des Zweier-Komplements.
- Der umgekehrte Weg, d.h. vom ganzzahligen Datentyp ohne Vorzeichen zum vorzeichenbehafteten Datentyp gleichen Ranges (also etwa von **unsigned int** zu **int**) hinterlässt ein *undefiniertes* Resultat, falls der Wert nicht darstellbar ist.

Konvertierungen bei numerischen Datentypen

- Bei einer Konvertierung von größeren ganzzahligeren Datentypen zu entsprechenden kleineren Datentypen werden die nicht mehr darstellbaren höherwertigen Bits weggeblendet, d.h. es gilt wiederum $a \bmod 2^n = b \bmod 2^n$, wobei n die Anzahl der Bits im kleineren Datentyp ist. (Das Resultat ist aber nur bei ganzzahligen Datentypen ohne Vorzeichen wohldefiniert.)
- Bei Konvertierungen zu *_Bool* ist das Resultat 0 (*false*), falls der Ausgangswert 0 ist, ansonsten immer 1 (*true*).
- Bei Konvertierungen von Gleitkommazahlen zu ganzzahligen Datentypen wird der ganzzahlige Anteil verwendet. Ist dieser im Zieltyp nicht darstellbar, so ist das Resultat undefiniert.
- Umgekehrt (beispielsweise auf dem Wege von **long long int** zu **float**) ist einer der beiden unmittelbar benachbarten darstellbaren Werte zu nehmen, d.h. es gilt entweder $a = b$ oder $a < b \wedge \nexists x : a < x < b$ oder $a > b \wedge \nexists x : a > x > b$ mit x aus der Menge des Zieltyps.

Konvertierungen bei numerischen Datentypen



Konvertierungen anderer skalarer Datentypen

- Jeder Aufzählungsdatentyp ist einem der ganzzahligen Datentypen implizit und implementierungsabhängig zugeordnet. Eine Konvertierung hängt von dieser (normalerweise nicht bekannten) Zuordnung ab.
- Zeiger lassen sich in C grundsätzlich als ganzzahlige Werte betrachten. Allerdings garantiert C nicht, dass es einen ganzzahligen Datentyp gibt, der den Wert eines Zeigers ohne Verlust aufnehmen kann.
- C99 hat hier die Datentypen *intptr_t* und *uintptr_t* in <stdint.h> eingeführt, die für die Repräsentierung von Zeigern als ganze Zahlen den geeignetsten Typ liefern.
- Selbst wenn diese groß genug sind, um Zeiger ohne Verlust aufnehmen zu können, so lässt der Standard dennoch offen, wie sich die beiden Typen *intptr_t* und *uintptr_t* innerhalb der Hierarchie der ganzzahligen Datentypen einordnen. Aber die weiteren Konvertierungsschritte und die damit verbundenen Konsequenzen ergeben sich aus dieser Einordnung.
- Die Zahl 0 wird bei einer Konvertierung in einen Zeigertyp immer in den Null-Zeiger konvertiert.

Implizite Konvertierungen

- Bei Zuweisungen wird der Rechts-Wert in den Datentyp des Links-Wertes konvertiert.
- Dies geschieht analog bei Funktionsaufrufen, wenn eine vollständige Deklaration der Funktion mit allen Parametern vorliegt.
- Wenn diese fehlt oder (wie beispielsweise bei *printf*) nicht vollständig ist, dann wird **float** implizit zu **double** konvertiert.

Implizite Konvertierungen

Die monadischen Operatoren !, $-$, $+$, ~ und $*$ konvertieren implizit ihren Operanden:

- Ein vorzeichenbehafteter ganzzahliger Datentyp mit einem Rang niedriger als **int** wird zu **int** konvertiert,
- Ganzzahlige Datentypen ohne Vorzeichen werden ebenfalls zu **int** konvertiert, falls sie einen Rang niedriger als **int** haben und ihre Werte in jedem Falle von **int** darstellbar sind. Ist nur letzteres nicht der Fall, so erfolgt eine implizite Konvertierung zu **unsigned int**.
- Ranghöhere ganzzahlige Datentypen werden nicht konvertiert.

Die gleichen Regeln werden auch getrennt für die beiden Operanden der Schiebe-Operatoren $<<$ und $>>$ angewendet.

Implizite Konvertierung

Bei dyadischen Operatoren mit numerischen Operanden werden folgende implizite Konvertierungen angewendet:

- Sind die Typen beider Operanden vorzeichenbehaftet oder beide ohne Vorzeichen, so findet eine implizite Konvertierung zu dem Datentyp mit dem höheren Rang statt. So wird beispielsweise bei einer Addition eines Werts des Typs **short int** zu einem Wert des Typs **long int** der erstere in den Datentyp des zweiten Operanden konvertiert, bevor die Addition durchgeführt wird.
- Ist bei einem gemischten Fall (**signed** vs. **unsigned**) in jedem Falle eine Repräsentierung eines Werts des vorzeichenlosen Typs in dem vorzeichenbehafteten Typ möglich (wie etwa typischerweise bei **unsigned short** und **long int**), so wird der Operand des vorzeichenlosen Typs in den vorzeichenbehafteten Typ des anderen Operanden konvertiert.
- Bei den anderen gemischten Fällen werden beide Operanden in die vorzeichenlose Variante des höherrangigen Operandentyps konvertiert. So wird beispielsweise eine Addition bei **unsigned int** und **int** in **unsigned int** durchgeführt.

Datentypen für unveränderliche Werte

C sieht einige spezielle Attribute bei Typ-Deklarationen vor. Darunter ist auch **const**:

$$
\begin{aligned}
\langle\text{declaration-specifiers}\rangle &\longrightarrow \langle\text{storage-class-specifier}\rangle\ [\ \langle\text{declaration-specifiers}\rangle\ ] \\
&\longrightarrow \langle\text{type-specifier}\rangle\ [\ \langle\text{declaration-specifiers}\rangle\ ] \\
&\longrightarrow \langle\text{type-qualifier}\rangle\ [\ \langle\text{declaration-specifiers}\rangle\ ] \\
&\longrightarrow \langle\text{function-specifier}\rangle\ [\ \langle\text{declaration-specifiers}\rangle\ ] \\
\langle\text{type-qualifier}\rangle &\longrightarrow \textbf{const} \\
&\longrightarrow \textbf{volatile} \\
&\longrightarrow \textbf{restrict}
\end{aligned}
$$

Datentypen für unveränderliche Werte

Die Verwendung des **const**-Attributs hat zwei Vorteile:

- Der Programmierer wird davor bewahrt, eine Konstante versehentlich zu verändern. (Dies funktioniert aber nur beschränkt.)
- Besondere Optimierungen sind für den Übersetzer möglich, wenn bekannt ist, dass sich bestimmte Variablen nicht verändern dürfen.

Datentypen für unveränderliche Werte

const.c

```
#include <stdio.h>

int main() {
    const int i = 1;

    i++; /* das geht doch nicht, oder?! */
    printf("i=%d\n", i);
}
```

- Der gcc beschränkt sich selbst dann nur auf Warnungen, wenn Konstanten offensichtlich verändert werden.

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

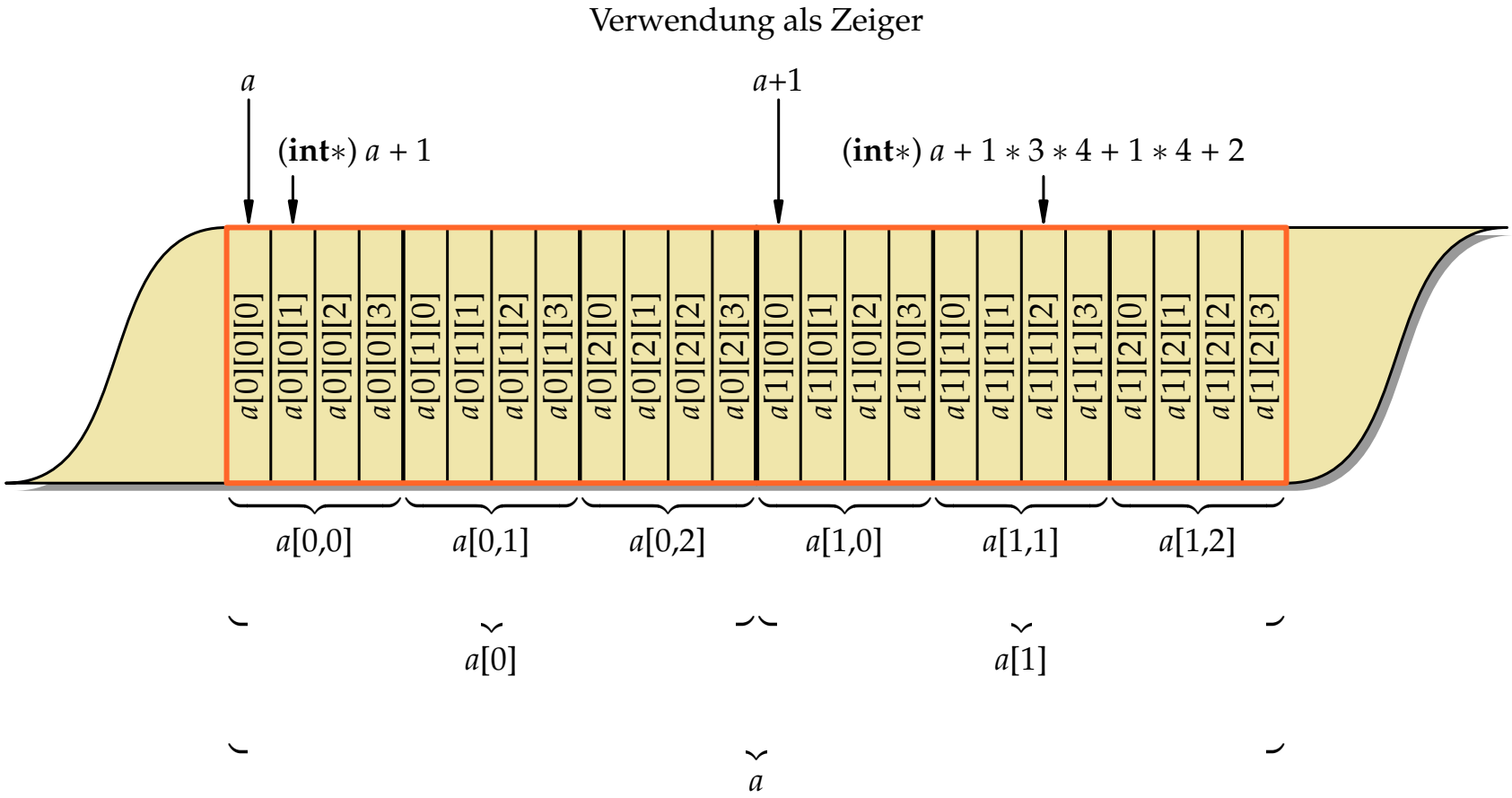
Angenommen die Anfangsadresse des Vektors liege bei 0x1000 und eine ganze Zahl vom Typ **int** würde vier Bytes belegen, dann würde die Repräsentierung des Vektors *matrix* im Speicher folgendermaßen aussehen:

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.