⟨declaration⟩ ⟶ ⟨declaration-specifiers⟩ [⟨init-declarator-list⟩]
⟨declaration-specifiers⟩ ⟶ ⟨storage-class-specifier⟩ [⟨declaration-specifiers⟩]
⟶ ⟨type-specifier⟩ [⟨declaration-specifiers⟩]
⟶ ⟨type-qualifier⟩ [⟨declaration-specifiers⟩]
⟶ ⟨function-specifier⟩ [⟨declaration-specifiers⟩]
⟨init-declarator-list⟩ ⟶ ⟨init-declarator⟩
⟶ ⟨init-declarator-list⟩ „,“ ⟨init-declarator⟩
⟨init-declarator⟩ ⟶ ⟨declarator⟩
⟶ ⟨declarator⟩ „=“ ⟨initializer⟩
⟨declarator⟩ ⟶ [⟨pointer⟩] ⟨direct-declarator⟩
⟨pointer⟩ ⟶ „*“ [⟨type-qualifier-list⟩]
⟶ „*“ [⟨type-qualifier-list⟩] ⟨pointer⟩

zeiger.c

```
#include <stdio.h>

int main() {
   int i = 13;
   int* p = &i; /* Zeiger p zeigt auf i; &i = Adresse von i */

   printf("i=%d, p=%p (Adresse), *p=%d (Wert)\n", i, p, *p);

   ++i;
   printf("i=%d, *p=%d\n", i, *p);

   ++*p; /* *p ist ein Links-Wert */
   printf("i=%d, *p=%d\n", i, *p);
}
```

- 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.

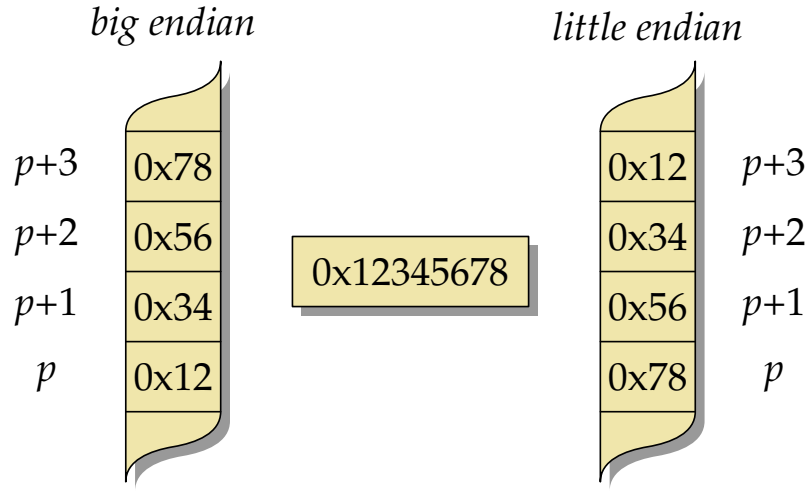
zeiger1.c

```
#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*.



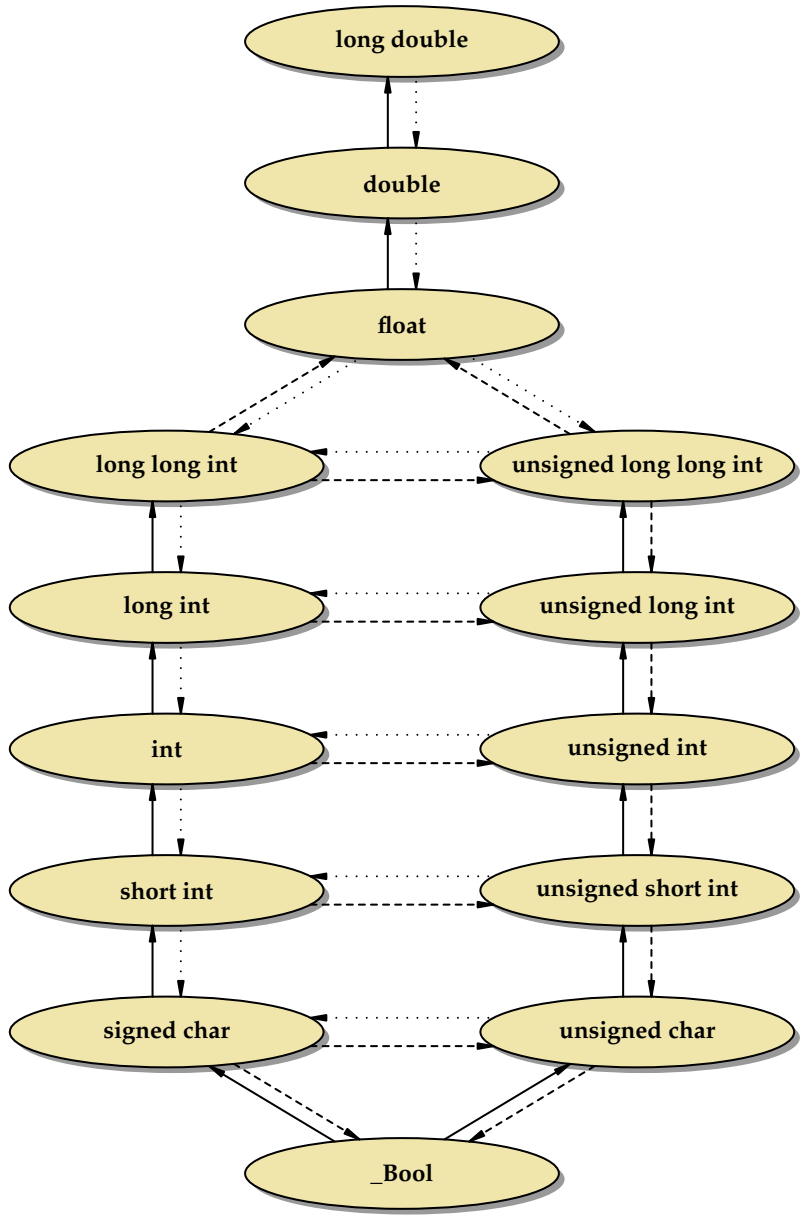
- Bei *little endian* wird das niedrigstwertige Byte an der niedrigsten Adresse abgelegt, während bei
- *big endian* das niedrigstwertige Byte sich bei der höchsten Adresse befindet.

- 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.

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.

- 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.


long double
double
float
long long int
unsigned long long int
long int
unsigned long int
int
unsigned int
short int
unsigned short int
signed char
unsigned char
_Bool

- 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.

- 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.

Die monadischen Operatoren !, $-$, $+$, $\sim$ 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.

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.

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

⟨declaration-specifiers⟩ ⟶ ⟨storage-class-specifier⟩ [⟨declaration-specifiers⟩]
⟶ ⟨type-specifier⟩ [⟨declaration-specifiers⟩]
⟶ ⟨type-qualifier⟩ [⟨declaration-specifiers⟩]
⟶ ⟨function-specifier⟩ [⟨declaration-specifiers⟩]

⟨type-qualifier⟩ ⟶ **const**
⟶ **volatile**
⟶ **restrict**
⟶ **_Atomic**

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

- Der Programmierer wird davor bewahrt, ein mit **const** deklarierte Variable 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.
- Teilweise schränkt **const** auch den Zugriff nur auf das Lesen ein, während nicht ausgeschlossen ist, dass andere ihn verändern. Beispiel: **const double**∗ *ptr*.

const.c

```
#include <stdio.h>
int main() {
    const int i = 1;
    i++;         /* das geht doch nicht, oder?! */
    printf("i=%d\n", i);
}
```

- Ältere Versionen des gcc beschränken sich selbst dann nur auf Warnungen, wenn Konstanten offensichtlich verändert werden. Ab gcc 4 wurde das geändert.

```
xylopia$ gcc --version | head -1
gcc (GCC) 3.2
xylopia$ gcc -o const const.c
const.c: In function `main':
const.c:6: warning: increment of read-only variable `i'
xylopia$ ./const
i=2
xylopia$
```

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

- Wie bei den Zeigertypen erfolgt die Typspezifikation eines Arrays nicht im Rahmen eines ⟨type-specifier⟩.
- Stattdessen gehört eine Array-Deklaration zu dem ⟨init-declarator⟩. Das bedeutet, dass die Präzisierung des Typs zur genannten Variablen unmittelbar gehört.
- Entsprechend deklariert

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

  eine Array-Variable *a* und eine ganzzahlige Variable *i*.

- Arrays und Zeiger sind eng miteinander verwandt.
- Der Variablenname eines Arrays ist ein konstanter Zeiger auf den zugehörigen Element-Typ, der auf das erste Element verweist.
- Allerdings liefert **sizeof** mit dem Namen des Arrays als Operand die Größe des gesamten Arrays und nicht etwa nur die des Zeigers.
- Entsprechend liefert **sizeof**(a)/ **sizeof**(a[0]) die Anzahl der Elemente eines Arrays a. (Grundsätzlich gilt, dass **sizeof**(a[0]) ein Teiler von **sizeof**(a) ist, d.h. Alignment-Anforderungen eines Element-Typs erzwingen bei Bedarf eine Aufrundung des Speicherbedarfs des Element-Typs und nicht erst des Arrays.)

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 (size_t i = 0; i < SIZE; ++i) {
      *(a + i) = i+1; /* gleichbedeutend mit a[i] = i+1 */
   }
   /* Elemente von a aufsummieren */
   int sum = 0;
   for (size_t i = 0; i < SIZE; i++) {
      sum += p[i];   /* gleichbedeutend mit ... = a[i]; */
   }
   printf("Summe: %d\n", sum);
}
```

- Die Indizierung beginnt immer bei 0.
- Ein Array 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 Java gibt es aber keine automatisierte Überprüfung. Diese wäre auch wegen der Verwandtschaft von Arrays und Zeigern nicht mit einem vertretbaren Aufwand in C umzusetzen.

In C ist das Belegen bzw. Deklarieren von Speicher für Arrays getrennt von dem Zugriff:

- Mit **int** $a[10]$; lässt sich ein Array mit 10 Elementen auf globaler oder lokaler Ebene deklarieren.
- Bei globalen Arrays erfolgt die Speicherbelegung beim Programmstart. Bei lokalen Arrays geschieht dies bei jedem Aufruf des umgebenden Blocks.
- Der Zugriff erfolgt über Zeiger und Zeigerarithmetik. Bei Zeigern ist jedoch nicht mehr der ursprüngliche Umfang des Arrays bekannt.
- Im folgenden Beispiel erfolgt der Zugriff über den Zeiger b:
  int$*$ $b = a[2]$; $b[1] = 42$;
  Nun hat $a[3]$ den Wert 42.
- Es ist darauf zu achten, dass der Speicher belegt bleibt, solange mit Zeigern noch darauf zugegriffen wird.

- Da der Name eines Arrays 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 Arrays, sondern hat dank dem Zeiger den direkten Zugriff auf das Array des Aufrufers.
- Die Dimensionierung eines Arrays muss explizit mit Hilfe weiterer Parameter übergeben werden, wenn diese variabel sein soll.

array2.c

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

const int SIZE = 10;

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

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

array2.c

```
int summe2(int* a, size_t length) {
   int sum = 0;
   for (size_t 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));
}
```

array3.c

```
int summe3(size_t length, int a[length]) {
   int sum = 0;
   for (size_t i = 0; i < length; i++) {
      sum += a[i];
   }
   return sum;
}

int main() {
   int array[17];

   init(sizeof(array)/sizeof(array[0]), array);
   printf("Summe: %d\n",
      summe3(sizeof(array)/sizeof(array[0]), array));
}
```

- Seit C99 sind auch variabel lange Arrays möglich bei lokalen Deklarationen und Parameterübergaben.

- So könnte ein zweidimensionales Array 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}};
```

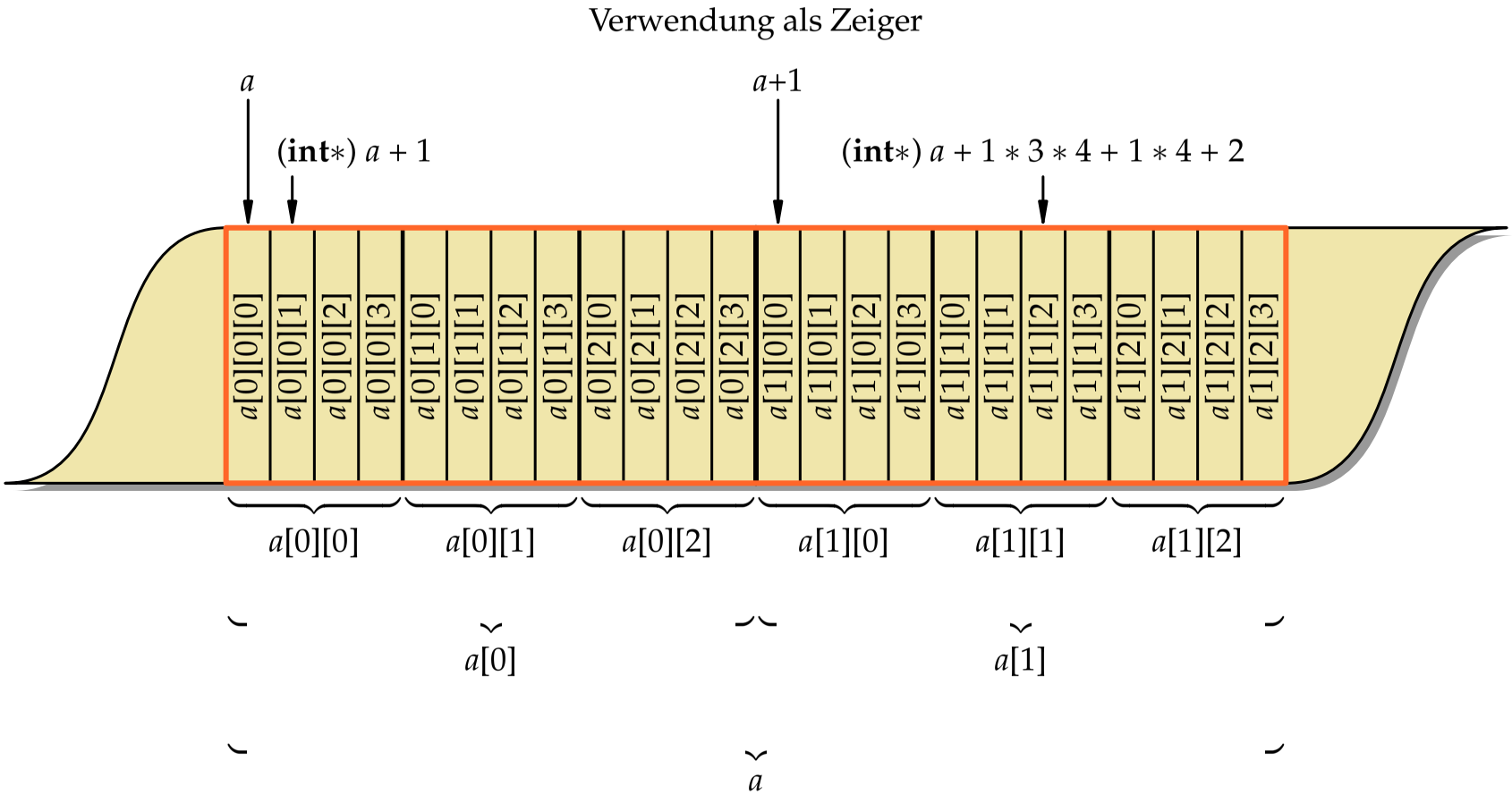
Angenommen, die Anfangsadresse des Arrays liege bei 0*x1000* und eine ganze Zahl vom Typ **int** würde vier Bytes belegen, dann würde die Repräsentierung des Arrays *matrix* im Speicher folgendermaßen aussehen:

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

Diese zeilenweise Anordnung nennt sich *row-major* und hat sich weitgehend durchgesetzt mit der wesentlichen Ausnahme von Fortran, das Matrizen spaltenweise anordnet (*column-major*).

- Gegeben sei:

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



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.
- Das gesamte Array wird zu einem eindimensionalen Array verflacht. Eine mehrdimensionale Indizierung erfolgt dann „per Hand“.
- Beginnend mit C99 wird auch mehrdimensionale dynamische Parameterübergaben für Arrays unterstützt. Dies wird von C++ nicht unterstützt und wird daher bislang kaum eingesetzt.

dynarray.c

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

void print_matrix(size_t rows, size_t cols,
      double m[rows][cols]) {
   for (size_t i = 0; i < rows; ++i) {
      for (size_t j = 0; j < cols; ++j) printf(" %10g", m[i][j]);
      printf("\n");
   }
}

int main() {
   double a[][3] = {{1.0, 2.3, 4.7}, {2.3, 4.4, 9.9}};
   print_matrix(/* rows = */ sizeof(a)/sizeof(a[0]),
      /* cols = */ sizeof(a[0])/sizeof(a[0][0]), a);
}
```

- Die Dimensionierungsparameter müssen dem entsprechenden Array in der Parameterdeklaration vorangehen.

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

- 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";
```

- Wenn die Zeichenketten-Konstante nicht eigens mit einer Initialisierung in ein deklariertes Array kopiert wird und somit der Zugriff nur über einen Zeiger erfolgt, sind nur Lesezugriffe zulässig.

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));
}
```

strings1.c

```
/* Laenge einer Zeichenkette bestimmen */
size_t my_strlen1(char s[]) {
   size_t len = 0;
   while (s[len]) { /* mit Null-byte vergleichen */
      ++len;
   }
   return len;
}
```

- 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.

strings1.c

```
/* Laenge einer Zeichenkette bestimmen */
size_t 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.

strings1.c

```
/* Kopieren einer Zeichenkette von s nach t
   Vorauss.: genuegend Platz in t */
void my_strcpy1(char t[], char s[]) {
   for (size_t 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 Arrays: Da die tatsächlich zur Verfügung stehende Länge des Arrays *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.

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.

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 sein könnte.
- Diese Warnung lässt sich (per Konvention) durch die doppelte Klammerung unterdrücken. Damit wird klar lesbar zum Ausdruck gegeben, dass es sich dabei nicht um ein Versehen handelt.

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[]) {
   size_t 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.