⟨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=%zu, sizeof(a)=%zu, sizeof(p)=%zu\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 <stddef.h>
#include <stdio.h>

const size_t 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.
- Dies wird jedoch von C++ nicht unterstützt und sollte daher vermieden werden.

- 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}};
```

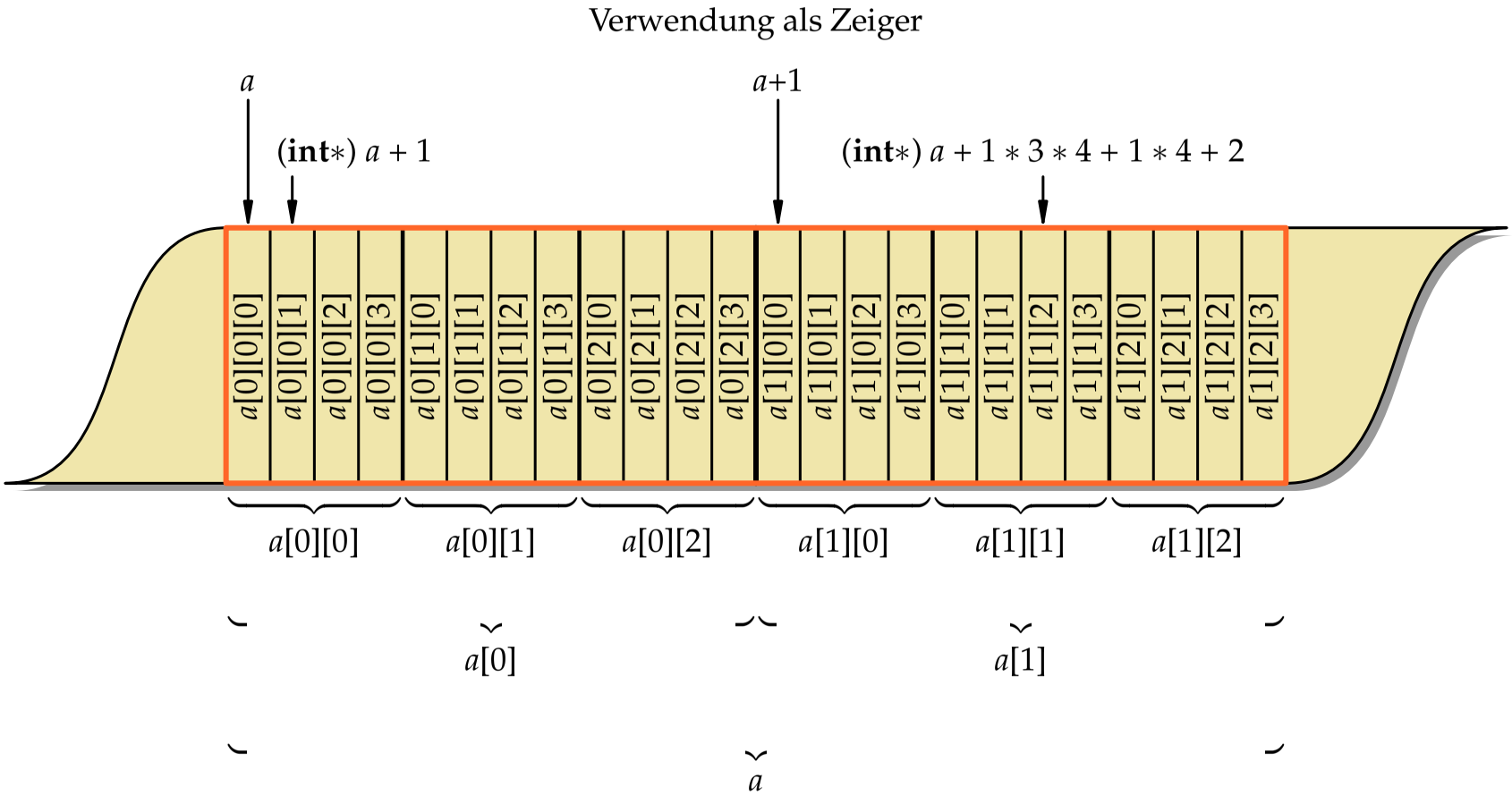
Angenommen, die Anfangsadresse des Arrays liege bei 0*x1000* und eine ganze Zahl vom Typ **int** würde vier Bytes belegen, dann würde die Repräsentierung des Arrays *matrix* im Speicher folgendermaßen aussehen:

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 entsprechend ist davon abzuraten.

dynarray.c

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

void print_matrix(size_t rows, size_t cols,
      const double A[rows][cols]) {
   for (size_t i = 0; i < rows; ++i) {
      for (size_t j = 0; j < cols; ++j) printf(" %10lg", A[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.

Matrizen (oder generell mehrdimensionale Arrays) können auch über Zeigerlisten realisiert werden:

- Jede Zeile ist ein eigenes Array.
- Die Zeigerliste ist ein Array von Zeigern auf die einzelnen Zeilen.
- Das gesamte Array wird dann durch einen einzigen Zeiger repräsentiert zuzüglich der Information über die Dimensionierungen.

Der Vorteil liegt darin, dass notationell der Zugriff auf die gleiche Weise erfolgt. Nachteil ist, dass solche Konstruktionen cache-unfreundlich sind, wenn die Zeilenarrays nicht kontinuierlich hintereinander im Speicher liegen. Java realisiert mehrdimensionale Arrays mit solchen Zeigerlisten.

ptrlist.c

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

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

int main() {
   const double A_rows[] = {
      1.0, 2.3, 4.7, 2.0,
      2.3, 4.4, 9.9, 3.4,
      2.1, 7.2, 3.1, 4.2};
   const double* A[] = {A_rows, A_rows + 4, A_rows + 8};
   print_matrix(3, 4, A);
}
```

Es bietet sich auch an, Matrizen durch ein eindimensionales Array zu repräsentieren. Wir benötigen dann zusätzliche Parameter:

- *incRow* spezifiziert, wieviel Elemente wir überspringen müssen, um zur nächsten Zeile zu gelangen.
- *incCol* spezifiziert, wieviel Elemente wir überspringen müssen, um zur nächsten Spalte zu gelangen.
- Bei einer *row-major*-Anordnung wird *incRow* auf die Zahl der Spalten gesetzt und *incCol* auf 1.
- Bei einer *col-major*-Anordnung wird *incCol* auf die Zahl der Zeilen gesetzt und *incRow* auf 1.

Diese Vorgehensweise ist flexibler und lässt sich auch bei C++ übernehmen.

dynarray2.c

```
void print_matrix(size_t rows, size_t cols,
      ptrdiff_t incRow, ptrdiff_t incCol,
      double* A) {
   for (size_t i = 0; i < rows; ++i) {
      for (size_t j = 0; j < cols; ++j) {
         printf(" %10lg", A[i * incRow + j * incCol]);
      }
      printf("\n");
   }
}
```

- Die Notaton $A[i][j]$ ist nun nicht mehr länger möglich. Stattdessen ist die Speicheradresse des gewünschten Elements explizit zu berechnen: $A[i * incRow + j * incCol]$.

dynarray2.c

```
void print_matrix(size_t rows, size_t cols,
      ptrdiff_t incRow, ptrdiff_t incCol,
      double* A) {
   for (size_t i = 0; i < rows; ++i) {
      for (size_t j = 0; j < cols; ++j) {
         printf(" %10lg", A[i * incRow + j * incCol]);
      }
      printf("\n");
   }
}

int main() {
   double A[4*3];
   init_matrix(4, 3, 3, 1, A); /* 4x3 matrix in row-major */
   printf("4x3 matrix A:\n");
   print_matrix(4, 3, 3, 1, A);
   printf("A transposed:\n");
   print_matrix(3, 4, 1, 3, A);
}
```

```
theon$ gcc -Wall -o dynarray2 dynarray2.c
theon$ dynarray2
4x3 matrix A:
          1          2          3
          4          5          6
          7          8          9
         10         11         12
A transposed:
          1          4          7         10
          2          5          8         11
          3          6          9         12
theon$
```

- Zum Transponieren einer Matrix sind nur *rows* und *cols* sowie *incRow* und *incCol* jeweils miteinander zu vertauschen.

mkl.h

```
/* from https://software.intel.com/en-us/node/520775 */
void cblas_dgemm (const CBLAS_LAYOUT Layout,
   const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb,
   const MKL_INT m, const MKL_INT n, const MKL_INT k,
   const double alpha,
   const double *a, const MKL_INT lda,
   const double *b, const MKL_INT ldb,
   const double beta,
   double *c, const MKL_INT ldc);
```

- Diese Funktionsdeklaration für eine Matrix-Matrix-Multiplikation entstammt der BLAS-Implementierung von Intel (Intel Math Kernel Library).
- BLAS steht für *Basic Linear Algebra Subprograms*, einer Standardschnittstelle, die sich seit 1979 entwickelt hat (ursprünglich nur für Fortran).

mkl.h

```
/* from https://software.intel.com/en-us/node/520775 */
void cblas_dgemm (const CBLAS_LAYOUT Layout,
   const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb,
   const MKL_INT m, const MKL_INT n, const MKL_INT k,
   const double alpha,
   const double *a, const MKL_INT lda,
   const double *b, const MKL_INT ldb,
   const double beta,
   double *c, const MKL_INT ldc);
```

- Die Funktion berechnet $C \leftarrow \alpha op_A(A) \times op_B(B) + \beta C$, wobei op_A und op_B entweder die identische Abbildung sind oder die Matrix jeweils transponieren (Parameter *transa* und *transb*).
- Der *layout*-Parameter legt fest, ob die Matrizen normalerweise in *col-major* (typisch für Fortran) oder in *row-major* angeordnet werden.

mkl.h

```
/* from https://software.intel.com/en-us/node/520775 */
void cblas_dgemm (const CBLAS_LAYOUT Layout,
   const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb,
   const MKL_INT m, const MKL_INT n, const MKL_INT k,
   const double alpha,
   const double *a, const MKL_INT lda,
   const double *b, const MKL_INT ldb,
   const double beta,
   double *c, const MKL_INT ldc);
```

- *lda*, *ldb* und *ldc* stehen für die sogenannten *leading dimensions*, d.h. statt *incRow* und *incCol* wird nur einer der beiden Parameter angegeben, der andere wird auf 1 gesetzt. Bei *col-major* wäre *incCol* bei der *leading dimension* anzugeben.
- Transponierungen führen dann zu entsprechenden Vertauschungen der Dimensionierungen und der Offsets *incRow* und *incCol*.

- 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): %zu\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.

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.

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 $s > t$.

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.