⟨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.

- 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. Hier würde **sizeof**(*a*) in *summe3* die Gesamtgröße des Arrays liefern.

- 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}};
```

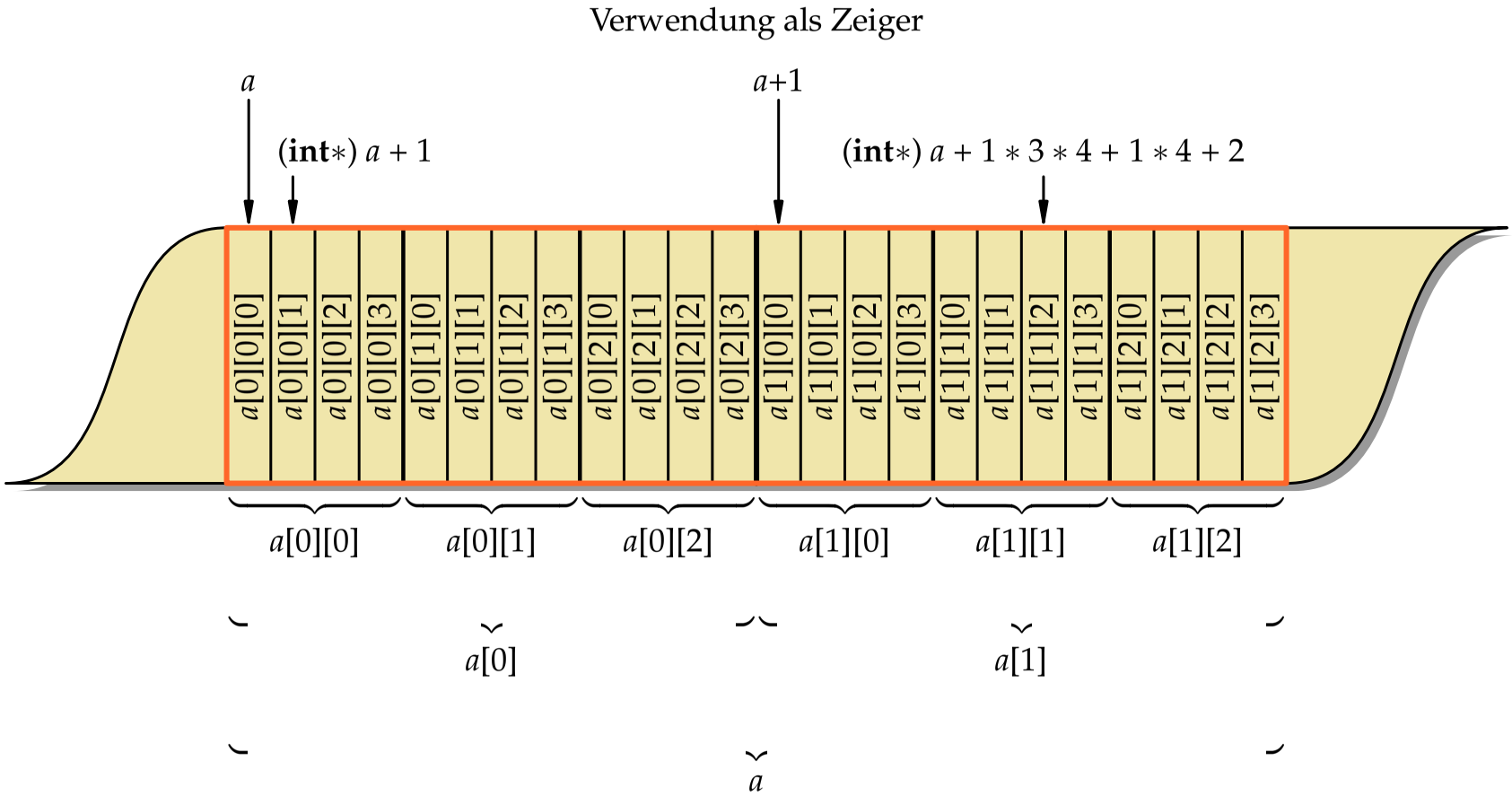
Angenommen, die Anfangsadresse des Arrays liege bei 0*x1000* und eine ganze Zahl vom Typ **int** würde vier Bytes belegen, dann würde die Repräsentierung des Arrays *matrix* im Speicher folgendermaßen aussehen:

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.

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.

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.

⟨structure-type-specifier⟩ ⟶ **struct** [⟨identifier⟩] „{“
⟨struct-declaration-list⟩ „}“
⟶ **struct** ⟨identifier⟩

⟨struct-declaration-list⟩ ⟶ ⟨struct-declaration⟩
⟶ ⟨struct-declaration-list⟩ ⟨struct-declaration⟩

⟨struct-declaration⟩ ⟶ ⟨specifier-qualifier-list⟩
⟨struct-declarator-list⟩ „;“

⟨specifier-qualifier-list⟩ ⟶ ⟨type-specifier⟩ [⟨specifier-qualifier-list⟩]
⟶ ⟨type-qualifier⟩ [⟨specifier-qualifier-list⟩]

⟨struct-declarator-list⟩ ⟶ ⟨struct-declarator⟩
⟶ ⟨struct-declarator-list⟩ „,“ ⟨struct-declarator⟩

⟨struct-declarator⟩ ⟶ ⟨declarator⟩
⟶ [⟨declarator⟩] „:“ ⟨constant-expression⟩

- Ein *Verbundtyp* (in C auch Struktur genannt) fasst mehrere *Elemente* zu einem Datentyp zusammen. Im Gegensatz zu Arrays können die Elemente *unterschiedlichen* Typs sein.
- Mit dem Schlüsselwort **struct** kann ein Verbundtyp wie folgt deklariert werden:

```
struct datum {
   unsigned short tag, monat, jahr;
};
```

- Hier ist *datum* der *Name des Verbundtyps*, der allerdings nur in Verbindung mit dem Schlüsselwort **struct** erkannt wird. Der hier deklarierte Verbundtyp repräsentiert – wie der Name schon andeutet – ein Datum. Jede Variable dieses Verbundtyps besteht aus drei ganzzahligen Komponenten, dem Tag, dem Monat und dem Jahr.

- Eine Variable *geburtsdatum* des Verbundtyps **struct** *datum* kann danach wie folgt angelegt werden:

```
struct datum geburtsdatum;
```

- Analog zu Aufzählungen lassen sich auch Variablen für namenlose Verbundtypen anlegen:

```
struct {
   unsigned short tag, monat, jahr;
} my_geburtsdatum;
```

- Ohne den Namen fehlt jedoch die Möglichkeit, weitere Variablen dieses Typs zu deklarieren oder den Typnamen in einer Typkonvertierung oder einem Aggregat zu spezifizieren.

- Variablen eines Verbund-Typs können bereits bei ihrer Definition initialisiert werden:

```
struct datum geburtsdatum = {3, 5, 1978};
```

- Es ist auch zulässig, die Elementnamen für die Initialisierung zu verwenden:

```
struct datum geburtsdatum = {.tag = 3, .monat = 5, .jahr = 1978};
```

- Alternativ kann auch der Wert eines Verbundtyps innerhalb eines Ausdrucks mit Hilfe eines Aggregats konstruiert werden:

```
struct datum geburtsdatum;
geburtsdatum = (struct datum) {3, 5, 1978};
/* oder mit Namen: */
geburtsdatum = {.tag = 3, .monat = 5, .jahr = 1978};
```

- Wenn durch eine Initialisierung nicht alle Variablen eines Verbundtyps nicht explizit initialisiert werden, dann werden die verbleibenden Teile auf 0 gesetzt.

- Auf die *Komponenten* eines Verbundtyps kann wie folgt zugegriffen werden:

```
struct datum gebdat = ...;

printf("%hu.%hu.%hu", gebdat.tag, gebdat.monat, gebdat.jahr);

struct datum *p = ...;

/* Zeiger zuerst dereferenzieren ... */
printf("%hu.%hu.%hu", (*p).tag, (*p).monat, (*p).jahr);
/* ... oder einfacher (und aequivalent) mit -> ... */
printf("%hu.%hu.%hu", p->tag, p->monat, p->jahr);
```

- Aufgrund der *Vorrang-Regeln* bei Operatoren ist $*p.tag$ äquivalent zu $*(p.tag)$ und nicht zu $(*p).tag$.
- Das Ausgabeformat *%hu* passt genau zu dem verwendeten Datentyp **unsigned short**.

- Die Elemente eines Verbundtyps können (beinahe) beliebigen Typs sein. Insbesondere ist es auch möglich, Verbundtypen ineinander zu verschachteln:

```
struct person {
   char* name;
   char* vorname;
   struct datum geburtsdatum;
};
```

- Wenn dann eine Variable *p* als **struct** *person p* vereinbart ist, dann kann wie folgt auf die Elemente zugegriffen werden:

```
p.name = ...;
p.vorname = ...;
p.geburtsdatum.tag = ...;
p.geburtsdatum.monat = ...;
p.geburtsdatum.jahr = ....;
```

struct.c

```
struct s {
   /* ... */
   struct s* p; /* Zeiger auf die eigene Struktur ist ok */
   /* struct s elem; */ /* nicht erlaubt! */
};

struct s1 {
   /* ... */
   struct s2* p;         /* Zeiger als Vorwaertsverweis ist ok */
   /* struct s2 elem;*/ /* nicht erlaubt! */
};

struct s2 {
   /* ... */
   struct s1* p;         /* Zeiger als Rueckwaertsverweis ok */
   struct s1 elem;       /* ok */
};
```

- Zeiger auf Verbundtypen können bereits verwendet werden, auch wenn die zugehörigen Strukturen noch nicht (bzw. nicht vollständig) deklariert sind.

Zuweisung von Verbundtypen

struct1.c

```
#include <stdio.h>

struct datum {
   unsigned short tag, monat, jahr;
};

int main() {
   struct datum vorl_beginn = {15, 10, 2015};
   struct datum ueb_beginn = {16, 10, 2015};

   printf("vorher: %hu.%hu.%hu\n",
      vorl_beginn.tag, vorl_beginn.monat, vorl_beginn.jahr);

   vorl_beginn = ueb_beginn;

   printf("nachher: %hu.%hu.%hu\n",
      vorl_beginn.tag, vorl_beginn.monat, vorl_beginn.jahr);
}
```

- Variablen des gleichen Verbundtyps können einander auch zugewiesen werden.
- Dabei werden die einzelnen *Elemente* der Struktur jeweils *kopiert*.

struct2.c

```
/* Werteparameter-Semantik */
void ausgabe1(struct datum d) {
   printf("%hu.%hu.%hu\n", d.tag, d.monat, d.jahr);
}

/* Referenzparameter-Semantik (wirkt sich hier nicht aus) */
void ausgabe2(struct datum* d) {
   printf("%hu.%hu.%hu\n", d->tag, d->monat, d->jahr);
}
```

- Verbunde können als Werteparameter übergeben werden oder – durch die Verwendung von Zeigern – auch als Referenz-Parameter verwendet werden.

struct2.c

```
/* Werteparameter-Semantik: Verbund des Aufrufers aendert sich nicht */
void setJahr1(struct datum d, int jahr) {
   d.jahr = jahr;
}

/* Referenzparameter-Semantik erlaubt die Aenderung */
void setJahr2(struct datum* d, int jahr) {
   d->jahr = jahr;
}

int main() {
   struct datum start = {15, 10, 2015};

   ausgabe1(start);
   setJahr1(start, 2016);      /* keine Aenderung! */
   ausgabe2(&start);           /* aequivalent zu ausgabe1(...) */
   setJahr2(&start, 2016);     /* setzt das Jahr auf 2016 */
   ausgabe1(start);
}
```

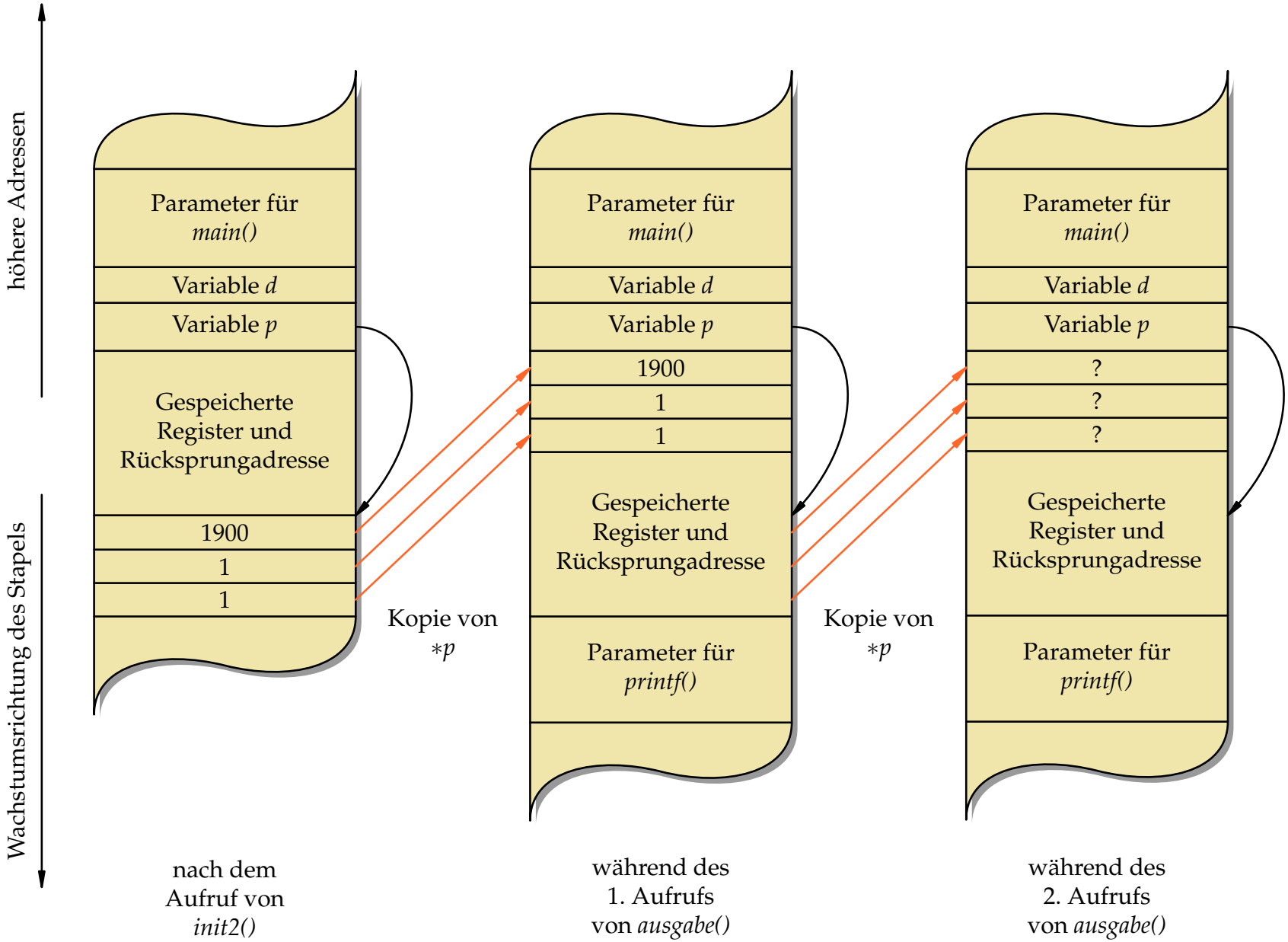
- Funktionen können als Ergebnistyp auch einen Verbundtyp verwenden.
- Hingegen ist Vorsicht angebracht, wenn Zeiger auf Verbunde zurückgegeben werden:

struct3.c

```
struct datum init1() {
   struct datum d = {1, 1, 1900};
   return d;   /* ok, denn es wird eine Kopie erzeugt */
}

struct datum* init2() {
   struct datum d = {1, 1, 1900};
   return &d;  /* nicht zulaessig, da Zeiger auf lokale Variable! */
}
```

struct3.c

```
#include <stdio.h>

struct datum {
   unsigned short tag, monat, jahr;
};

void ausgabe(struct datum d) {
   printf("%hu.%hu.%hu\n", d.tag, d.monat, d.jahr);
}

/* init1() & init2() */

int main() {
   struct datum d;
   struct datum* p;

   d = init1();
   ausgabe(d);

   p = init2(); /* Zeiger auf Variable, die nicht mehr existiert! */
   ausgabe(*p); /* wenn's klappt ... dann ist das Glueck! */
   ausgabe(*p); /* sollte eigentliche dasselbe ausgeben :-( */
}
```


höhere Adressen
Wachstumsrichtung des Stapels
Parameter für main()
Variable d
Variable p
Gespeicherte Register und Rücksprungadresse
1900
1
1
Kopie von *p
Parameter für printf()
?
?
?
nach dem Aufruf von init2()
während des 1. Aufrufs von ausgabe()
während des 2. Aufrufs von ausgabe()

- Die Variable *d* in der Funktion *init2()* ist eine lokale Variable, die auf dem Laufzeit-Stapel für Funktionen (im Englischen *runtime stack* genannt) lebt.
- Sie existiert nur solange, wie diese Funktion ausgeführt wird. Danach wird dieser Speicherplatz evtl. anderweitig verwendet.
- Nach dem Aufruf von *init2()* ist zwar die Lebenszeit der Daten hinter *p* zwar vorbei, aber sie liegen typischerweise immer noch intakt auf dem Laufzeit-Stapel.
- Entsprechend werden beim ersten Aufruf von *ausgabe()* die Daten noch korrekt kopiert.
- Allerdings werden die von *p* referenzierten Daten dann während des ersten Aufrufs von *ausgabe()* überschrieben. Deswegen werden beim folgenden zweiten Aufruf von *ausgabe()* vollkommen undefinierte Werte bei der Parameterübergabe kopiert.

$$\langle \text{union-type-specifier} \rangle \longrightarrow \textbf{union} \; [\; \langle \text{identifier} \rangle \;] \; „\{“ \; \langle \text{struct-declaration-list} \rangle \; „\}“$$
$$\longrightarrow \textbf{union} \; \langle \text{identifier} \rangle$$

- Syntaktisch gleichen variante Verbunde den regulären Verbunden – es wird nur das Schlüsselwort **union** anstelle von **struct** verwendet.
- Im Vergleich zu den regulären Verbunden liegen alle Variablen eines varianten Verbunds an der gleichen Position im Speicher.

In folgenden Szenarien kann der Einsatz von Verbunden sinnvoll sein:

- Variante Verbunde sparen Speicherplatz ein, wenn immer nur eine Variante benötigt wird. In diesem Falle muss (außerhalb des varianten Verbunds) ein Status verwaltet werden, der festhält, welche Variante gerade in Benutzung ist.
- Durch variante Verbunde sind zwei (oder mehr) Sichten durch verschiedene Datentypen auf ein gemeinsames Stück Speicher möglich, ohne dass hierfür jeweils umständliche Konvertierungen notwendig wären. Allerdings ist hier Vorsicht geboten, da dies sehr von der jeweiligen Plattform abhängen kann.

union.c

```
union IPAddr {
   unsigned int ip;
   unsigned char b[4];
};
```

- Alle Komponenten eines Verbunds liegen an der gleichen Speicheradresse.
- Der Speicherbedarf der größten Komponente bestimmt den Speicherbedarf für den gesamten varianten Verbund.
- In diesem Beispiel sind *ip* und *b* zwei Sichten auf das gleiche Stück Speicher: Einerseits kann eine IP-Adresse als ganze Zahl betrachtet werden, andererseits aber auch als Sequenz von vier Bytes.
- Der Unterschied zwischen *big* und *little endian* ist hier wieder relevant.

Beispiel eines varianten Verbunds

union.c

```
int main() {
   union IPAddr a;

   a.ip = 0x863c4205;  /* bel. IP-Adresse in int-Darst. zuweisen */
   /* Zugriff auf a ueber die Komponente ip */
   printf("%u [%x]\n", a.ip, a.ip);
   /* Zugriff auf a ueber die Komponente b */
   printf("%hhu.%hhu.%hhu.%hhu ", a.b[0], a.b[1], a.b[2], a.b[3]);
   printf("[%02hhx.%02hhx.%02hhx.%02hhx]\n",
      a.b[0], a.b[1], a.b[2], a.b[3]);
   puts("");
   printf("Speicherplatzbedarf: %zd\n", sizeof(a));

   puts("");   /* Anordung im Speicher analysieren */
   puts("Position im Speicher:");
   printf("a:    %p\n", &a);
   printf("ip:   %p\n", &a.ip);
   printf("b[0]: %p\n", &a.b[0]);
   printf("b[1]: %p\n", &a.b[1]);
   printf("b[2]: %p\n", &a.b[2]);
   printf("b[3]: %p\n", &a.b[3]);
}
```

```
clonard$ uname -m
sun4u
clonard$ gcc -Wall -std=gnu99 union.c -o union
clonard$ union
2252096005 [863c4205]
134.60.66.5 [86.3c.42.05]

Speicherplatzbedarf: 4

Position im Speicher:
a:    ffbff6ac
ip:   ffbff6ac
b[0]: ffbff6ac
b[1]: ffbff6ad
b[2]: ffbff6ae
b[3]: ffbff6af
clonard$
```

- Ausführung auf einer *big endian*-Maschine.

Beispiel eines varianten Verbunds


```
thales$ uname -m
i86pc
thales$ gcc -Wall -std=gnu99 union.c -o union
thales$ union
2252096005 [863c4205]
5.66.60.134 [05.42.3c.86]

Speicherplatzbedarf: 4

Position im Speicher:
a:    804729c
ip:   804729c
b[0]: 804729c
b[1]: 804729d
b[2]: 804729e
b[3]: 804729f
thales$
```

- Ausführung auf einer *little endian*-Maschine.

$$
\begin{array}{rcl}
\langle \text{typedef-name} \rangle & \longrightarrow & \langle \text{identifier} \rangle \\
\langle \text{storage-class-specifier} \rangle & \longrightarrow & \textbf{typedef} \\
& \longrightarrow & \textbf{extern} \\
& \longrightarrow & \textbf{static} \\
& \longrightarrow & \textbf{auto} \\
& \longrightarrow & \textbf{register}
\end{array}
$$

- Einer Deklaration kann das Schlüsselwort **typedef** vorausgehen. Dann wird der Name, der sonst ein Variablen- oder Funktionsname geworden wäre, stattdessen zu einem neu definierten Typnamen. Dieser Typname kann anschließend überall dort verwendet werden, wo die Angabe eines ⟨type-specifier⟩ zulässig ist.

Ein einfaches Beispiel einer Typdefinition


```
typedef int Laenge; /* Vereinbarung des eigenen Typnames "Laenge" */

/* ... */

Laenge i, j; /* Vereinbarung der Variablen i und j vom Typ Laenge */
```

- Hier ist *Laenge* zu einem Synonym für **int** geworden.
- Damit sind **int** *i*, *j*; und *Laenge i*, *j*; äquivalente Vereinbarungen.
- Hier bieten Typdefinitionen die Flexibilität, einen Typ an einer zentralen Stelle zu vereinbaren, um ihn dann bequem für das gesamte Programm verändern zu können.
- Das ist insbesondere sinnvoll bei der Verwendung numerischer Datentypen. Synonyme können auch zur Lesbarkeit beitragen, wenn besonders „sprechende“ Namen verwendet werden.

```
typedef char* CharPointer;
typedef int TenIntegers[10];
CharPointer cp1, cp2; // beide sind vom Typ char*
char* cp3, cp4; // cp4 hat nur den Typ char!
TenIntegers a, b; // beides sind Vektoren
int c[10], d; // d hat nur den Typ int!
```

- Typdefinitionen ermöglichen es, komplexere Typen in einen ⟨type-specifier⟩ zu integrieren, die sich sonst nur im Rahmen einer ⟨declaration⟩ formulieren liessen.
- Das betrifft insbesondere Zeiger und Arrays.

```
typedef struct datum {
   unsigned short tag, monat, jahr;
} datum;
datum geburtsdatum; // aequivalent zu struct datum geburtsdatum
datum heute, morgen;
```

- Bei Verbunden werden ebenfalls Typdefinitionen verwendet, um anschließend nur den Namen ohne das Schlüsselwort **struct** verwenden zu können.
- Die Verwendung von Typnamen aus Typdefinitionen bleibt – abgesehen von den syntaktischen Unterschieden – äquivalent zur Verwendung des ursprünglichen Datentyps. Entsprechend entsteht durch eine Typdefinition kein neuer Typ, der nicht mehr mit dem alten Typ kompatibel wäre.

- Durch die unglückliche Aufteilung von Typ-Spezifikationen in ⟨type-specifier⟩ (links stehend) und ⟨declarator⟩ (rechts stehend, sich um den Namen anordnend), werden komplexere Deklarationen rasch unübersichtlich.
- Die Motivation für diese Syntax kam wohl aus dem Wunsch, dass die Deklaration einer Variablen ihrer Verwendung gleichen solle.
- Entsprechend hilft es, sich bei komplexeren Deklarationen die Vorränge und Assoziativitäten der zugehörigen Operatoren in Erinnerung zu rufen.

```
char* x[10];
```

- Der Vorrangtabelle lässt sich entnehmen, dass der []-Operator einen höheren Vorrang (16) im Vergleich zum $*$-Operator (15) hat.
- Entsprechend handelt es sich bei x um ein Array mit 10 Elementen des Typs Zeiger auf **char**.
- Im Einzelnen:

$x[10]$	Array mit 10 Elementen
$* x[10]$	Array mit 10 Zeigern
char$* x[10]$	Array mit 10 Zeigern auf Zeichen

```
char (*x)[10];
```

- Wenn der $*$-Operator Vorrang erhalten soll, so ist in Kombination mit dem []-Operator eine Klammerung notwendig.
- Mit der Klammerung wird x als Zeiger auf ein Array mit 10 Elementen des Types **char** deklariert.
- Im Einzelnen:

$(*x)$	Zeiger
$(*x)[10]$	Zeiger auf ein Array mit 10 Elementen
char $(*x)[10]$	Zeiger auf ein Array mit 10 Elementen des Typs **char**

Zweites Beispiel für die Analyse einer Deklaration 170

```
int* (*(*x)())[5];
```

- Die Analyse beginnt hier wieder beim Variablennamen x in der Mitte der Deklaration:

$*x$	ein Zeiger
$(*x)()$	ein Zeiger auf eine Funktion
$*(*x)()$	ein Zeiger auf eine Funktion, die einen Zeiger liefert
$(*(*x)())[5]$	ein Zeiger auf eine Funktion, die einen Zeiger auf ein 5-elementiges Array liefert
$* (*(*x)())[5]$	ein Zeiger auf eine Funktion, die einen Zeiger auf ein 5-elementiges Array aus Zeigern liefert
int$* (*(*x)())[5]$	ein Zeiger auf eine Funktion, die einen Zeiger auf ein 5-elementiges Array aus Zeigern auf **int** liefert

```
int* (*(*x)())[5];
```

- An zwei Stellen waren hier Vorränge relevant: Im zweiten Schritt war wesentlich, dass Funktionsaufrufe (Vorrangstufe 16) Vorrang haben vor der Dereferenzierung (Vorrangstufe 15) und im vierten Schritt hatte die Indizierung (Vorrangstufe 16) ebenfalls Vorrang vor der Dereferenzierung.
- Zusammenfassend:
  - [] und () haben einen höheren Rang als $*$.
  - [] und () assoziieren von *links nach rechts*, während $*$ von *rechts nach links* gruppiert.

```
int* (*(*x)())[5];
```

- Lesbarer wird dies durch einen stufenweisen Aufbau mit Typdefinitionen:

```
typedef int* intp; // intp = Zeiger auf int
typedef intp intpa[5]; // intpa = Vektor mit 5 Zeigern auf int
typedef intpa f(); // f = Funktion, die intpa liefert
typedef f* fp; // Zeiger auf eine Funktion
fp x;
```

```
int (*x[10])();
```

- Klammern können verwendet werden, um die Operatoren anders zu gruppieren und damit den Typ entsprechend zu verändern.
- Hier ist x ein 10-elementiges Array von Zeigern auf Funktionen mit Rückgabewerten des Typs **int**. Im Einzelnen:

$x[10]$	x als 10-elementiges Array
$(*x[10])$	x als 10-elementiges Array von Zeigern
$(*x[10])()$	x als 10-elementiges Array von Zeigern auf Funktionen
int $(*x[10])()$	x als 10-elementiges Array von Zeigern auf Funktionen, mit Rückgabewerten des Typs **int**.

Beispiele für unzulässige Typdeklarationen

int *af*[]()	Array von Funktionen, die Rückgabewerte des Typs **int** liefern.
int *fa*()[]	Funktion, die ein Array von ganzen Zahlen liefert; hier wäre **int**∗ *fa*() akzeptabel gewesen.
int *ff*()()	Funktion, die eine Funktion liefert, welche wiederum **int** liefert.