⟨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 = (struct datum) {.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);
}
```

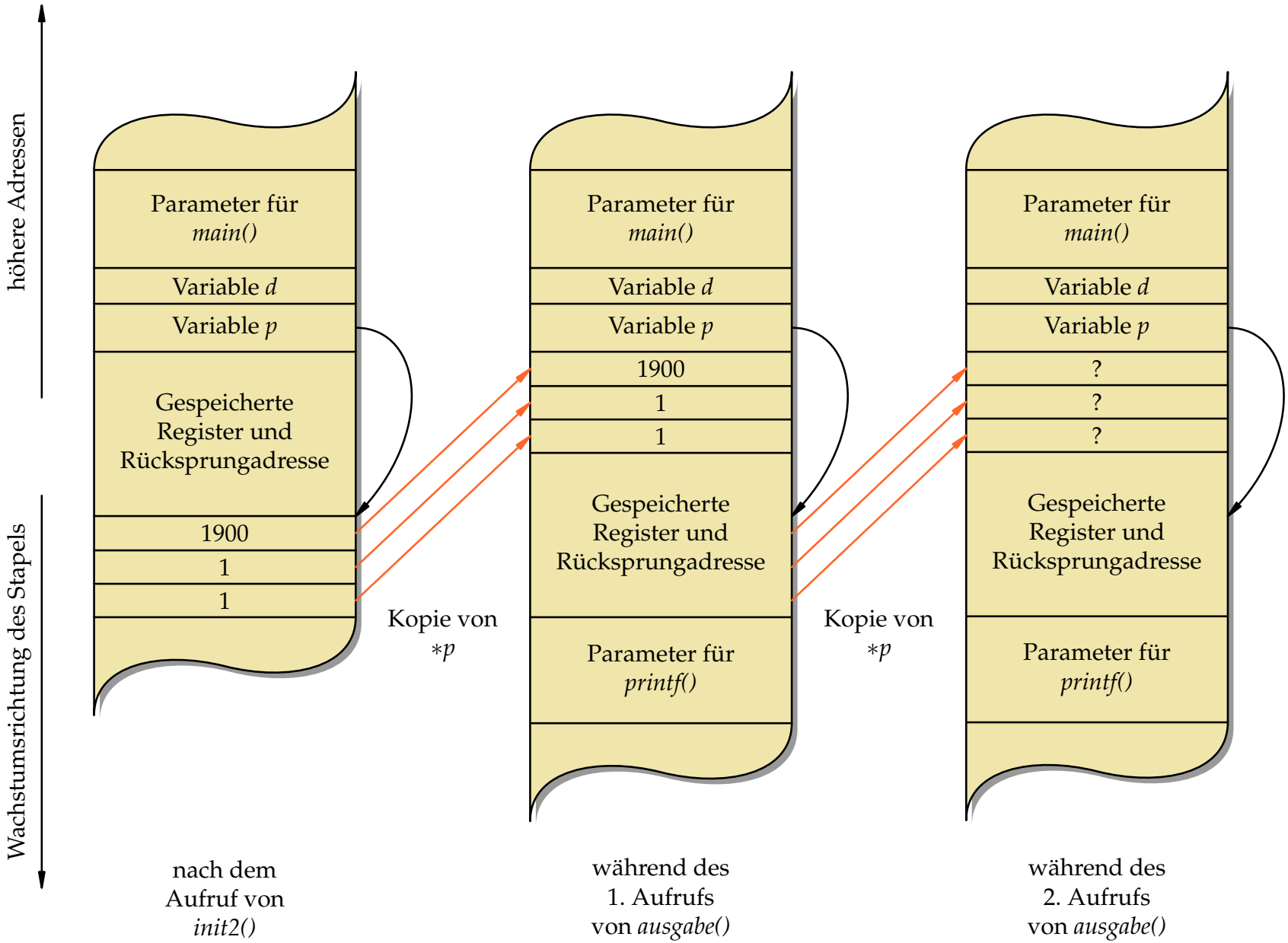
- Funktionen können als Ergebnistyp auch einen Verbundtyp verwenden.
- Hingegen ist Vorsicht angebracht, wenn Zeiger auf Verbunde zurückgegeben werden:

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()
Parameter für main()
Variable d
Variable p
1900
1
1
Gespeicherte Register und Rücksprungadresse
Kopie von *p
Parameter für printf()
Parameter für main()
Variable d
Variable p
?
?
?
Gespeicherte Register und Rücksprungadresse
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.

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

- 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.

```
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 176

```
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* intpap; // intpap = Zeiger auf intpa
typedef intpap (*fp)(); // f = Zeiger auf eine Funktion, die intpap liefert
fp y;
```

Verwendung von Klammern bei Typdeklarationen 179

```
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**.

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.

- Anders als in Java gibt es keine automatisierte Speicherverwaltung, die unbenötigte Speicherflächen automatisch freigibt.
- Entsprechend muss in C Speicher nicht nur explizit belegt, sondern auch explizit freigegeben werden.
- Dies ist recht fehleranfällig.
- Hinzu kommt, dass Speicherflächen zunächst nicht mit Datentypen verbunden sind. Sie müssen aufgeteilt, verwaltet und Zeiger müssen je nach Bedarf konvertiert werden. Entsprechend fehlt hier die Sicherheit des Typsystems.
- Zum Ausgleich dafür lässt sich eine Speicherverwaltung in C selbst schreiben.

void∗ *calloc*(*size_t nelem*, *size_t* elsize)

Belegt Speicher für *nelem* Elemente der Größe *elsize* und initialisiert diesen mit 0. Im Erfolgsfall wird der Zeiger darauf geliefert, ansonsten 0.

void∗ *malloc*(*size_t size*)

Belegt Speicher für ein Objekt, das *size* Bytes benötigt. Im Erfolgsfall wird der Zeiger darauf geliefert, ansonsten 0.

void free(**void**∗ *ptr*)

Hier muss *ptr* auf eine zuvor belegte, jedoch noch nicht freigegebene Speicherfläche verweisen. Dann gibt *free* diese Fläche zur andersweitigen Nutzung wieder frei.

void∗ *realloc*(**void**∗ *ptr*, *size_t size*)

Versucht, die Größe der Speicherfläche, auf die *ptr* verweist, auf *size* Bytes anzupassen. Im Erfolgsfalle wird ein Zeiger auf die (möglicherweise neue) Speicherfläche zurückgeliefert, ansonsten 0.

void∗ *aligned_alloc*(*size_t* alignment, *size_t* size)

Neu eingeführt in C11, berücksichtigt Adresskanten.

Eine einfache lineare Liste

reverse.c

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

/* lineare Liste ganzer Zahlen */
typedef struct element {
   int i;
   struct element* next;
} element;

int main() {
   element* head = 0;
   int i;

   /* Zahlen einlesen und in der Liste
      in umgekehrter Reihenfolge ablegen */
   while ((scanf("%d", &i)) == 1) {
      element* last = (element*) calloc(1, sizeof(element));
      if (last == 0) {
         fprintf(stderr, "out of memory!\n"); exit(1);
      }
      last->i = i; last->next = head; head = last;
   }
   /* Zahlen aus der Liste wieder ausgeben */
   while (head != 0) {
      printf("%d\n", head->i);
      head = head->next;
   }
}
```

reverse.c

```
element* last = (element*) calloc(1, sizeof(element));
if (last == 0) {
   fprintf(stderr, "out of memory!\n"); exit(1);
}
```

- *calloc* wird hier darum gebeten, für ein Element der Größe **sizeof**(*element*) Speicher zu belegen.
- Das entspricht *element last* = **new** *element*() in Java.
- Falls der gewünschte Speicher nicht belegt werden kann, wird ein 0-Zeiger zurückgeliefert. Entsprechend ist in C immer ein anschließender Test auf 0 erforderlich.
- Wenn es klappt, wird durch *calloc* die Speicherfläche mit Nullen initialisiert.
- *calloc* liefert den generischen Zeiger **void**∗ zurück. Dieser ist zu allen anderen Zeigern kompatibel. Der Cast-Operator (*element*∗) macht diese Typkonvertierung hier explizit.

reverse.c

```
element* last = (element*) malloc(sizeof(element));
if (last == 0) {
   fprintf(stderr, "out of memory!\n"); exit(1);
}
```

- Alternativ kann auch *malloc* aufgerufen werden.
- *malloc* erwartet jedoch nur die Gesamtgröße der belegenden Speicherfläche in Bytes und unterlässt die Initialisierung.
- Der Inhalt des neuen Objekts ist deswegen im Erfolgsfall vollkommen uninitialisiert.

```
void* my_calloc(size_t nelem, size_t elsize) {
   void* ptr = calloc(nelem, elsize);
   if (ptr) return ptr; /* alles OK */
   /* Fehlerbehandlung: */
   fprintf(stderr, "out of memory -- aborting!\n");
   /* Termination mit core dump */
   abort();
}
```

- Die Behandlung des Falls, dass ein 0-Zeiger zurückgeliefert wird, sollte nie vergessen werden.
- Wem das zu mühsam erscheint, kann diese Überprüfung in eine separate Funktion auslagern wie *my_calloc* in diesem Fall.