

# ProgGru

L. Leuenberger, M. Ehrler, T. Andermatt

4. Januar 2014

## Inhaltsverzeichnis

<b>1</b>	<b>Vom Problem zum Programm</b>	<b>Kapitel 1.2</b>	<b>3</b>
1.1	Algorithmus . . . . .		3
1.2	Der euklidische Algorithmus als Beispiel . . . . .		3
1.3	Nassi-Shneiderman-Diagramme	Kapitel 1.3	4
<b>2</b>	<b>Kontrollstrukturen</b>		<b>4</b>
2.1	Sequenz	Kapitel 8.1	4
2.2	Selektion	Kapitel 8.2	4
2.3	Iteration	Kapitel 8.3	5
2.4	Sprunganweisungen	Kapitel 8.4	5
<b>3</b>	<b>Typenkonzept</b>	<b>Kapitel 5</b>	<b>5</b>
3.1	Übersicht über alle Standard-Datentypen	Kapitel 5.2	5
3.2	Variablen	Kapitel 5.3	6
3.3	Typ-Attribute	Kapitel 5.4	6
3.4	Klassifikation von Datentypen	Kapitel 5.5 und Kapitel 5.6	7
<b>4</b>	<b>Funktionen</b>		<b>7</b>
4.1	Aufgaben einer Funktion . . . . .		7
4.2	Definition von Funktionen	Kapitel 9.3.1	7
4.3	Eingaben/Ausgaben einer Funktion	Kapitel 9.3	7
4.4	Deklaration von Funktionen	Kapitel 9.4	8
<b>5</b>	<b>Pointer und Arrays</b>	<b>Kapitel 6</b>	<b>8</b>
5.1	Arbeitsspeicher - Memory Map	Kapitel 6.1	8
5.2	Pointer	Kapitel 6.1	8
5.3	Arrays	Kapitel 6.3	11
5.4	Das Schlüsselwort const bei Pointern und Arrays	Kapitel 10.4	13
<b>6</b>	<b>Stringverarbeitung</b>		<b>13</b>
6.1	Kopieren eines Strings	Kapitel 10.5	13
6.2	Standardfunktionen für Strings und Speicher	Kapitel 10.6	13
6.3	Funktionen zur Speicherbearbeitung	Kapitel 10.6.2	14
<b>7</b>	<b>Lexikalische Konventionen, Enum</b>		<b>14</b>
7.1	Bestandteile des Sourcecodes . . . . .		14
7.2	Zeichenvorrat von C . . . . .		15
7.3	Lexikalische Analyse des Sourcecodes . . . . .		15
7.4	Lexikalische Einheiten (Token) . . . . .		15
7.5	Enumerations (Aufzählungstyp) . . . . .		16

<b>8</b>	<b>Anweisungen, Ausdrücke und Operatoren</b>	<b>16</b>
8.1	Operatoren und Operanden . . . . .	16
8.2	Postfix- und Präfixoperatoren . . . . .	16
8.3	Ausdrücke und Anweisungen . . . . .	17
8.4	Nebeneffekte . . . . .	17
8.5	Auswertungsreihenfolge . . . . .	17
8.6	L-Werte und R-Werte . . . . .	18
8.7	Operatoren im einzelnen . . . . .	18
8.8	Type Cast (Typumwandlungsoperator) . . . . .	19
8.9	Gültigkeitsbereiche von Namen (Scope) . . . . .	19
<b>9</b>	<b>Strukturen und Unionen</b>	<b>20</b>
9.1	Strukturen . . . . .	20
9.2	Unions . . . . .	22
9.3	Allgemeines zu Strukturen und Unions . . . . .	22
<b>10</b>	<b>Komplexe Datentypen und Typennamen</b>	<b>23</b>
10.1	Komplexere Vereinbarungen . . . . .	23
10.2	Typdefinitionen . . . . .	23
<b>11</b>	<b>Speicherklassen</b>	<b>24</b>
11.1	Adressraum eines Programms . . . . .	24
11.2	Programme aus mehreren Dateien . . . . .	25
11.3	Speicherklassen extern . . . . .	26
11.4	Speicherklassen static . . . . .	26
11.5	Speicherklassen bei lokalen Variablen . . . . .	26
<b>12</b>	<b>Rekursion und Iteration <span style="color: red;">Kapitel 9.8</span></b>	<b>27</b>
12.1	Unterschied von Rekursion und Iteration <span style="color: red;">Kapitel 9.8.1</span> . . . . .	27
12.2	Anwendung von rekursiven Funktionen <span style="color: red;">Kapitel 9.8.1</span> . . . . .	27
12.3	Beispiel anhand der Fakultätsberechnung <span style="color: red;">Kapitel 9.8.2</span> . . . . .	27
12.4	Beispiel anhand der Binärdarstellung <span style="color: red;">Kapitel 9.8.3</span> . . . . .	27
<b>13</b>	<b>Anhang: Beispiele</b>	<b>27</b>

# Vom Problem zum Programm **Kapitel 1.2**

## 1.1 Algorithmus

Der Begriff Programm ist eng mit dem Begriff Algorithmus verbunden. Algorithmen sind Vorschriften für die Lösung eines Problems, welches die Handlungen und ihre Abfolge, also die Handlungsweise, beschreiben. Abstrakt kann man sagen, dass die folgenden Bestandteile und Eigenschaften zu einem Algorithmus gehören: (*am Beispiel eines Kochrezeptes erklärt*)

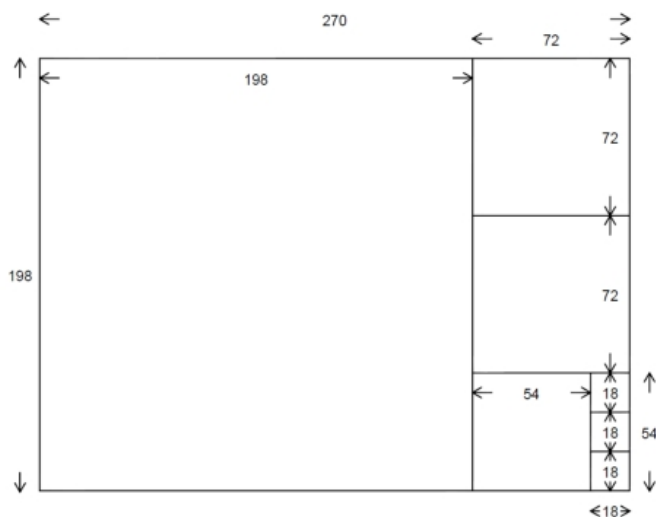
- eine **Menge von Objekten**, die durch den Algorithmus bearbeitet werden (*Zutaten, Geschirr, Herd, ...*)
- eine **Menge von Operationen**, die auf den Objekten ausgeführt werden (*waschen, schälen, ...*)
- ein **definierter Anfangszustand**, in dem sich die Objekte zu Beginn befinden (*Teller leer, Herd kalt, ...*)
- ein **gewünschter Endzustand**, in dem sich die Objekte nach der Lösung des Problems befinden sollen (*gekochtes Gemüse, ...*)

## 1.2 Der euklidische Algorithmus als Beispiel

### 1.2.1 Das Problem

Eine rechteckige Terrasse sei mit möglichst grossen quadratischen Platten auszulegen. Welche Kantenlänge haben die Platten?

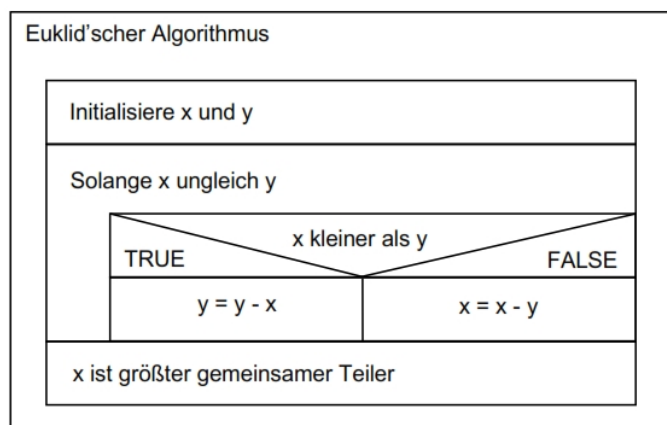
### 1.2.2 Der Algorithmus



Mit Abschneidetechnik nach Euklid. Entspricht der Ermittlung des grössten gemeinsamen Teilers (ggT):

$$\frac{x_{\text{ungekürzt}}}{y_{\text{ungekürzt}}} = \frac{\frac{x_{\text{ungekürzt}}}{\text{ggT}(x_{\text{ungekürzt}}, y_{\text{ungekürzt}})}}{\frac{y_{\text{ungekürzt}}}{\text{ggT}(x_{\text{ungekürzt}}, y_{\text{ungekürzt}})}} = \frac{x_{\text{gekürzt}}}{y_{\text{gekürzt}}}$$

### 1.2.3 Algorithmus-Beschreibung mit Struktogramm **Kapitel 1.3**



### 1.2.4 Algorithmus-Beschreibung mit Pseudocode

#### **Kapitel 1.2.1**

Eingabe der Seitenlängen: x, y (natuerliche Zahlen)  
solange x ungleich y ist, wiederhole  
wenn x groesser als y ist, dann  
ziehe y von x ab und weise das Ergebnis x zu  
andernfalls  
ziehe x von y ab und weise das Ergebnis y zu  
wenn x gleich y ist, dann ist x (bzw. y) der gesuchte ggT

### 1.2.5 Programm

```
#include <stdio.h>
int main(void)
{
    int x = 24;
    int y = 9;
    while (x != y)
    {
        if (x < y)
            y = y - x;
        else
            x = x - y;
    }
    printf("ggT ist: %d\n", x);
    return 0;
}
```

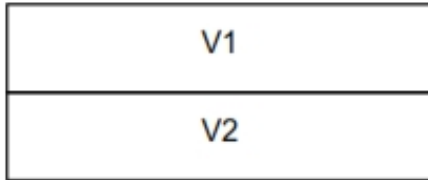
### 1.2.6 Trace-Tabelle **Kapitel 1.2.4**

Verarbeitungsschritt	x	y
Initialisierung x = 24, y = 9	24	9
x = x - y	15	9
x = x - y	6	9
y = y - x	6	3
x = x - y	3	3
ggT ist: 3		

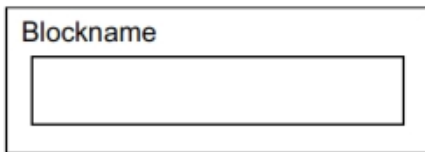
### 1.3 Nassi-Shneiderman-Diagramme Kapitel 1.3

Zur Visualisierung des Kontrollflusses von Programmen, das heisst, zur grafischen Veranschaulichung ihres Ablaufes, wurden 1973 von Nassi und Shneiderman grafische Strukturen, die sogenannten Struktogramme, vorgeschlagen. Entwirft man Programme mit Nassi-Shneiderman-Diagrammen, so genügt man automatisch den Regeln der Strukturierten Programmierung.

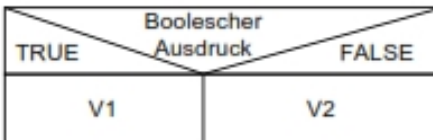
#### 1.3.1 Sequenz



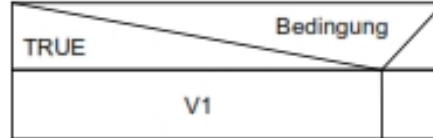
#### 1.3.2 Block



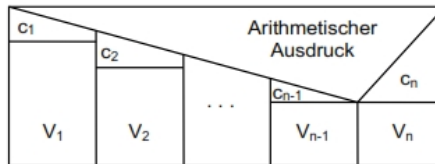
#### 1.3.3 Einfache Alternative



#### 1.3.4 Bedingte Anweisung



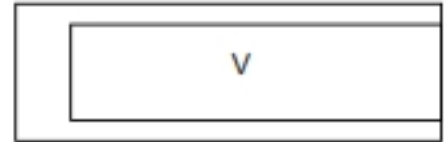
#### 1.3.5 Mehrfache Alternative



#### 1.3.6 Schleife mit vorheriger Prüfung



#### 1.3.7 Endlosschleife



#### 1.3.8 Schleife mit nachfolgender Prüfung



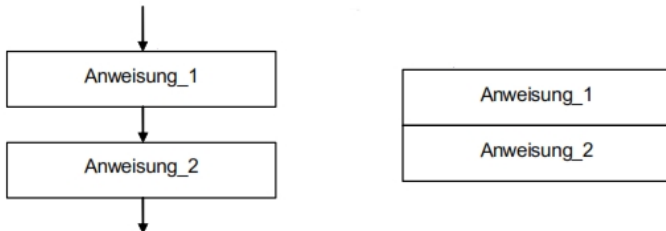
#### 1.3.9 Abbruchanweisung



## 2 Kontrollstrukturen

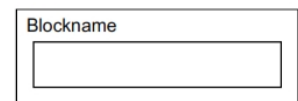
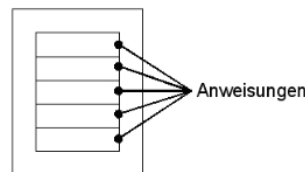
### 2.1 Sequenz Kapitel 8.1

Die Sequenz ist eine zeitlich geordnete Abfolge von Anweisungen.



#### 2.1.1 Block

- Erfordert die Syntax genau eine Anweisung, so können dennoch mehrere Anweisungen geschrieben werden, wenn man sie in Form eines Blocks zusammenfasst.
- Ein Block wird mit geschweiften Klammern eingefasst. {...} Ein Block zählt syntaktisch als eine einzige Anweisung.



### 2.2 Selektion Kapitel 8.2

Von **Selektion** spricht man zum einen, wenn man eine Anweisung nur dann ausführen will, wenn eine bestimmte Bedingung zutrifft. Zum anderen möchte man mit Selektionsanweisungen zwischen zwei Möglichkeiten (entweder/oder) bzw. zwischen mehreren Möglichkeiten genau eine auswählen.

#### 2.2.1 Einfache Alternative

```
if (Ausdruck)
  Anweisung wenn wahr;
else
  Anweisung wenn falsch;
```

#### 2.2.2 Bedingte Anweisung

```
if (Ausdruck)
  Anweisung wenn wahr;
```

#### 2.2.3 Mehrfache Alternative - else if

```
if (Ausdruck 1)
  Anweisung wenn Ausdruck 1 wahr;
else if (Ausdruck 2)
  Anweisung wenn Ausdruck 2 wahr;
else
  Anweisung wenn alle falsch (optional);
```

### 2.2.4 Mehrfache Alternative - switch case

- Für eine Mehrfach-Selektion, d.h. eine Selektion unter mehreren Alternativen, kann die *switch*-Anweisung verwendet werden, falls die Alternativen ganzzahligen Werten eines Ausdrucks von einem Integer-Typ entsprechen.
- Hat der Ausdruck der *switch*-Anweisung den gleichen Wert wie einer der konstanten Ausdrücke der *case*-Marken, wird die Ausführung des Programms mit der Anweisung hinter dieser *case*-Marke weitergeführt.
- Stimmt keiner der konstanten Ausdrücke mit dem *switch*-Ausdruck überein, wird zu *default* gesprungen.

```
switch (Ausdruck)
{
    case Wert 1:
        Anweisung 1;
        break;
    case Wert 2:
        Anweisung 2;
        break;
    default:
        Anweisung wenn nichts zutrifft
        (optional);
}
```

## 2.3 Iteration **Kapitel 8.3**

### 2.3.1 While

```
while (Ausdruck)
    Anweisung;
```

### 2.3.2 For-Schleife

```
for (Ausdruck_init; Ausdruck; Ausdruck_update)
    Anweisung;
```

### 2.3.3 Do-While

```
do
    Anweisung;
while (Ausdruck);
```

### 2.3.4 Endlosschleife

```
for (;;)
    Anweisung;

while (1)
    Anweisung;
```

### 2.3.5 Wann wird welche Schleife eingesetzt?

- For-Schleife: Bei Zählschleifen, d.h. wenn die Anzahl Durchläufe (kann auch variabel sein) im voraus feststeht.
- Do-While-Schleife: Wenn es keine Zählschleife ist, und die Schleife muss mindestens einmal durchlaufen werden
- While-Schleife: In allen anderen Fällen

## 2.4 Sprunganweisungen **Kapitel 8.4**

- *break*: *do – while*-, *while*-, *for*-Schleife und *switch*-Anweisung abbrechen
- *continue*: in den nächsten Schleifendurchgang (Schleifenkopf) springen bei *do – while*-, *while*- und *for*-Schleife
- *return*: aus Funktion an aufrufende Stelle zurückspringen
- *goto*: innerhalb einer Funktion an eine Marke (Label) springen

## 3 Typenkonzept **Kapitel 5**

In C wird verlangt, dass alle Variablen einen genau definierten, vom Programmierer festgelegten Typ haben. Der Typ bestimmt, welche Werte eine Variable annehmen kann und welche nicht.

### 3.1 Übersicht über alle Standard-Datentypen **Kapitel 5.2**

Datentyp	Anzahl Bytes	Wertebereich (dezimal)	Typ	Verwendung
<i>char</i>	1	−128 bis +127	Ganzzahltyp	speichern eines Zeichens
<i>unsigned char</i>	1	0 bis +255	Ganzzahltyp	speichern eines Zeichens
<i>signed char</i>	1	−128 bis +127	Ganzzahltyp	speichern eines Zeichens
<i>int</i>	4 (in der Regel)	−2'147'483'648 bis +2'147'483'647	Ganzzahltyp	effizienteste Grösse
<i>unsigned int</i>	4 (in der Regel)	0 bis +4'294'967'295	Ganzzahltyp	effizienteste Grösse
<i>short int</i>	2 (in der Regel)	−32'768 bis +32'767	Ganzzahltyp	kleine ganzzahlige Werte
<i>unsigned short int</i>	2 (in der Regel)	0 bis +65'535	Ganzzahltyp	kleine ganzzahlige Werte
<i>long int</i>	4 (in der Regel)	−2'147'483'648 bis +2'147'483'647	Ganzzahltyp	grosse ganzzahlige Werte
<i>unsigned long int</i>	4 (in der Regel)	0 bis +4'294'967'295	Ganzzahltyp	grosse ganzzahlige Werte
<i>float</i>	4 (in der Regel)	−3.4 * 10 <sup>38</sup> bis +3.4 * 10 <sup>38</sup>	Gleitpunkttyp	Gleitpunktzahl
<i>double</i>	8 (in der Regel)	−1.7 * 10 <sup>308</sup> bis +1.7 * 10 <sup>308</sup>	Gleitpunkttyp	höhere Genauigkeit
<i>long double</i>	4 (in der Regel)	−1.1 * 10 <sup>4932</sup> bis +1.1 * 10 <sup>4932</sup>	Gleitpunkttyp	noch höhere Genauigkeit

### 3.1.1 Ganzzahltypen (Integertypen) Kapitel 5.2

- Alle Integertypen ausser *char* sind per Default vorzeichenbehaftet.
- Bei *char* ist es compilerabhängig.
- Voranstellen des Schlüsselwortes *unsigned* bewirkt, dass alle Bits für eine positive Zahl verwendet werden. (keine negativen Zahlen möglich)
- Eine Überlaufproblematik (Overflow) bei *signed* und *unsigned* Typen ist vorhanden. Überläufe müssen vom Programmierer abgefangen werden!
- Die Werte werden bei *unsigned* Typen im Zweierkomplement abgespeichert.

### 3.1.2 Gleitpunkttypen Kapitel 5.2

- Gleitpunkttypen sind sehr viel aufwendiger in der Berechnung als Integertypen.
- Speziell bei kleinen Microcontrollern ohne FPU (floating point unit) sollte wenn möglich auf Gleitpunkttypen verzichtet werden.
- Die Werte werden gemäss Floating Point Standard IEEE 754 abgespeichert. Die Berechnung ist zu finden im Kapitel 5.2.3.

## 3.2 Variablen Kapitel 5.3

- Deklaration: legt nur die Art und den Typ der Variable, bzw. die Schnittstelle der Funktion fest ohne Speicherplatz zu reservieren
  - Definition: legt die Art und den Typ der Variablen bzw. Funktionen fest und reserviert Speicherplatz dafür
- Definition = Deklaration + Reservierung des Speicherplatzes**

### 3.2.1 Definition von Variablen Kapitel 5.3.1

Eine einzelne Variable wird definiert durch eine Vereinbarung der Form:

`datentyp name;`

also beispielsweise durch

`int x;`

Vom selben Typ können auch mehrere Variablen gleichzeitig definiert werden:

`int x, y, z;`

### 3.2.3 Manuelle Initialisierung von Variablen Kapitel 5.3.3

Jede einfache Variable kann bei ihrer Definition initialisiert werden:

`int x = 5;`

Es ist zu empfehlen, immer alle Variablen (lokal und global) vor dem ersten Lesezugriff manuell zu initialisieren.

### 3.2.5 Sichtbarkeit von Variablen Kapitel 9.2

Die Sichtbarkeit einer Variablen bedeutet, dass man auf sie über ihren Namen zugreifen kann:

- Variablen in inneren Blöcken sind nach aussen nicht sichtbar.
- Globale Variablen und Variablen in äusseren Blöcken sind in inneren Blöcken sichtbar.
- Wird in einem Block eine lokale Variable definiert mit demselben Namen wie eine globale Variable oder wie eine Variable in einem umfassenden Block, so ist innerhalb des Blocks nur die lokale Variable sichtbar. Die globale Variable in dem umfassenden Block wird durch die Namensgleichheit verdeckt.
- Wird in einem Block eine lokale Variable definiert mit demselben Namen wie eine Funktion, so ist innerhalb des Blockes nur die lokale Variable sichtbar. Die Funktion wird durch die Namensgleichheit verdeckt, da Funktionen denselben Namensraum wie Variablen haben.

## 3.3 Typ-Attribute Kapitel 5.4

- *const*: Die Variable kann nur initialisiert werden. Weitere Änderungen sind nicht mehr möglich.

`const double PI = 3.1415927;`

- *volatile*: Die Variable wird nicht (weg-)optimiert durch den Compiler, d.h. die Adressen der Variablen werden nicht geändert. Dies wird benötigt, wenn eine Variable auf einer definierten Adresse liegen muss (z.B. Memory-Mapped-Input/Output bei einem Mikrocontroller)

### 3.2.2 Interne und externe Variablen Kapitel 5.3.2

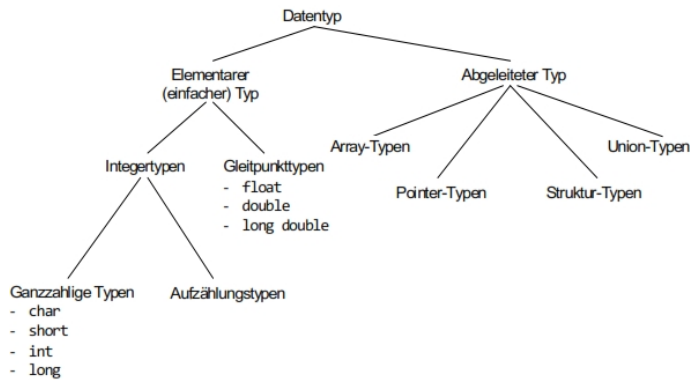
- Globale (externe) Variablen: Diese Variablen stehen allen Funktionen zur Verfügung und müssen ausserhalb von Funktionen definiert werden.
- Lokale (interne) Variablen: Diese Variablen stehen nur der Funktion zur Verfügung, in welcher die definiert wurden. Sie kann nicht von ausserhalb angesprochen werden.

**Grundsätzlich gilt: Variablen so lokal wie möglich definieren!**

### 3.2.4 Automatische Initialisierung von Variablen Kapitel 5.3.3

- Globale Variablen werden beim Programmstart immer auf Null gesetzt.
- Lokale Variablen werden **nicht** automatisch initialisiert und enthalten einen zufälligen Wert.

### 3.4 Klassifikation von Datentypen Kapitel 5.5 und Kapitel 5.6



In der Programmiersprache C gibt es drei Klassen von Typen:

- Objekttypen (Datentypen): Objekttypen beschreiben Variablen, z.B. *int*
- Funktionstypen: Funktionstypen beschreiben Funktionen, z.B. *int f(void)*
- unvollständige Typen: Der Typ *void* ist ein unvollständiger Typ, der nicht vollständig gemacht werden kann. Er bezeichnet eine leere Menge und wird beispielsweise verwendet, wenn eine Funktion keinen Rückgabewert oder keine Übergebeparameter hat.

## 4 Funktionen

### 4.1 Aufgaben einer Funktion

- Gleichartige, funktional zusammengehörende Programmteile unter einem eigenen Namen zusammenfassen. Der Programmteil kann mit diesem Namen aufgerufen werden.
- Einige Funktionen (im speziellen mathematische) sollen parametrisiert werden können, z.B. die Cosinusfunktion macht nur Sinn, wenn sie mit unterschiedlichen Argumenten aufgerufen werden kann.
- Divide et impera (divide and conquer, teile und herrsche): Ein grosses Problem ist einfacher zu lösen, wenn es in mehrere einfachere Teilprobleme aufgeteilt wird.

### 4.2 Definition von Funktionen Kapitel 9.3.1

```

rückgabetyf funktionsname (typ_1 formaler_parameter_1,
                           typ_2 formaler_parameter_2,
                           . . . . .
                           typ_n formaler_parameter_n)
{
    . . .
}
    
```

↑ Funktionskopf  
↓ Funktionsrumpf

- Funktionskopf: legt die Aufrufschnittstelle (Signatur) der Funktion fest. Er besteht aus Rückgabetyf, Funktionsname und Parameterliste.
- Funktionsrumpf: Lokale Vereinbarungen und Anweisungen innerhalb eines Blocks

### 4.3 Eingaben/Ausgaben einer Funktion Kapitel 9.3

#### 4.3.1 Eingabedaten

Es sind folgende Möglichkeiten vorhanden um Daten an Funktionen zu übergeben:

- Mithilfe von Werten, welche an die Parameterliste übergeben werden
- Mithilfe von globalen Variablen

#### 4.3.2 Ausgabedaten

Es sind folgende Möglichkeiten vorhanden um Daten zurückzugeben:

- Mithilfe des Rückgabewertes einer Funktion (*return*)
- Mithilfe von Änderungen an Variablen, deren Adresse über die Parameterliste an die Funktion übergeben wurde
- Mithilfe von Änderungen an globalen Variablen

#### 4.3.3 Beispiele

**Parameterlos und ohne Rückgabewert:**

```

void printGestrichelteLinie(void)
{
    printf("_____");
} ...
printGestrichelteLinie(); // Aufruf
    
```

**Parameter und ohne Rückgabewert:**

```

void printSumme(int a, int b)
{
    printf("%d", a + b);
} ...
int zahl = 14;
printSumme(zahl, 54); // Aufruf
    
```

**Parameter und Rückgabewert:**

```

int getSumme(int a, int b)
{
    return (a + b);
} ...
int summe;
summe = getSumme(13 54); // Aufruf
    
```



## 4.4 Deklaration von Funktionen Kapitel 9.4

Es ist festgelegt, dass die Konsistenz zwischen Funktionskopf und Funktionsaufrufen vom Compiler überprüft werden soll. Dazu muss beim Aufruf der Funktion die Schnittstelle der Funktion, d.h. der Funktionskopf, bereits bekannt sein. Steht aber die Definition einer Funktion im Programmcode erst nach ihrem Aufruf, so muss eine Vorwärtsdeklaration der Funktion erfolgen, indem vor dem Aufruf die Schnittstelle der Funktion mit dem Funktionsprototypen deklariert wird. Desweiteren ist zu beachten, dass Parameternamen im Funktionsprototyp und in der Funktionsdefinition nicht übereinstimmen müssen. Es ist jedoch zu empfehlen.

### 4.4.1 Beispiel

```
#include <stdio.h>
void init(int beta); /* Funktionsprototyp */
int main(void)
{
    ...
}
void init(int alpha) /* Funktionsdefinition */
{
    ...
}
```

### 4.4.2 Was passiert wenn der Prototyp vergessen geht?

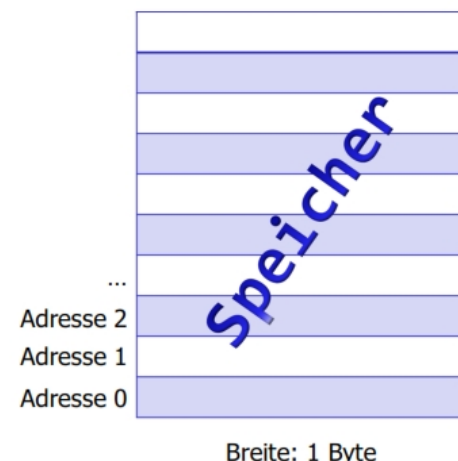
- Fehlt der Prototyp ganz, so wird die Funktion implizit (automatisch vom System) deklariert. Ihr Rückgabotyp wird als *int* angenommen, die Parameter werden nicht überprüft.
- Wenn die Funktion später definiert wird und nicht *int* als Rückgabotyp hat, bringt der Compiler eine Fehlermeldung.

### 4.4.3 Funktionsprototypen in der Praxis Kapitel 9.4

- Funktionsprototypen, welche die Schnittstelle der Unit beschreiben, kommen in das entsprechenden Headerfile.
- Jedes C-File, welches diese Schnittstelle nutzt, inkludiert dieses Headerfile und somit die Funktionsprototypen.
- Funktionsprototypen von internen Funktionen der Unit werden zuoberst im C-File aufgelistet und kommen nicht ins Headerfile.

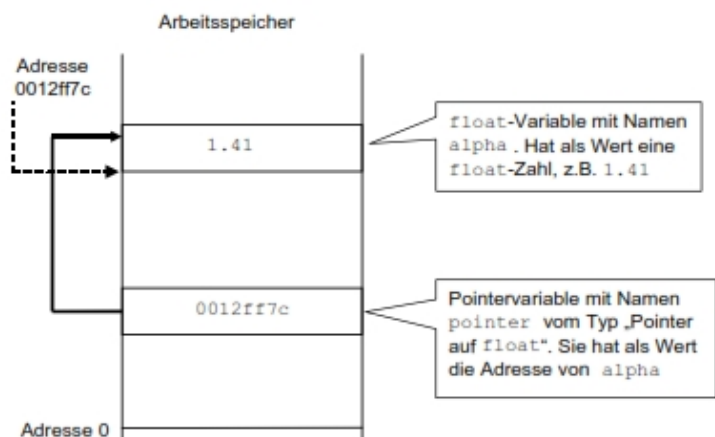
## 5 Pointer und Arrays Kapitel 6

### 5.1 Arbeitsspeicher - Memory Map Kapitel 6.1



- Der gesamte Speicher besteht aus einer Folge von einzelnen Bytes, welche durchnummeriert werden.
- Diese eindeutige Nummer einer Speicherzelle wird als Adresse bezeichnet.
- Bei einem byteweise adressierbaren Speicher (ist üblich) liegt an jeder Adresse genau 1 Byte.

### 5.2 Pointer Kapitel 6.1



- Ein Pointer ist eine Variable, welche die Adresse einer im Speicher befindlichen Variablen oder Funktion aufnehmen kann.
- Man sagt, der Pointer zeige (to point) auf diese Speicherzelle.
- Pointer in C sind typisiert, sie zeigen auf eine Variable des definierten Typs.
- Der Speicherbereich, auf den ein bestimmter Pointer zeigt, wird entsprechend des definierten Pointer-Typs interpretiert.
- Der Speicherbedarf einer Pointervariablen ist unabhängig vom Pointer-Typ. Er ist so gross, dass die maximale Adresse Platz findet (z.B. 32 Bit).



### 5.2.1 Definition einer Pointervariablen Kapitel 6.1

```

Typname* pointerName;
int* ptr1;    // ptr1 ist ein Pointer auf int
double* ptr2; // ptr2 ist ein Pointer auf double

```

### 5.2.3 Der Adressoperator (Referenzierung) Kapitel 6.1

Ist  $x$  eine Variable vom Typ  $Typname$ , so liefert der Ausdruck  $\&x$  einen Pointer auf die Variable  $x$ , d.h. er liefert die Adresse der Variablen  $x$ .

```

int wert;
// Variable wert vom Typ int wird
// definiert
int* ptr;
// Pointer ptr auf den Typ int wird
// definiert
// ptr zeigt auf eine nicht definierte
// Adresse
ptr = &wert;
// ptr zeigt nun auf die Variable wert,
// d.h. ptr enthaelt die Adresse der
// Variablen wert

```

### 5.2.2 Initialisierung mit Nullpointer Kapitel 6.1

NULL ist vordefiniert (in *stddef.h*) und setzt den Pointer auf einen definierten Nullwert. Besser ist es, statt NULL direkt 0 zu verwenden.

```
int* ptr = 0;
```

### 5.2.4 Der Inhaltsoperator \* (Dereferenzierung) Kapitel 6.1

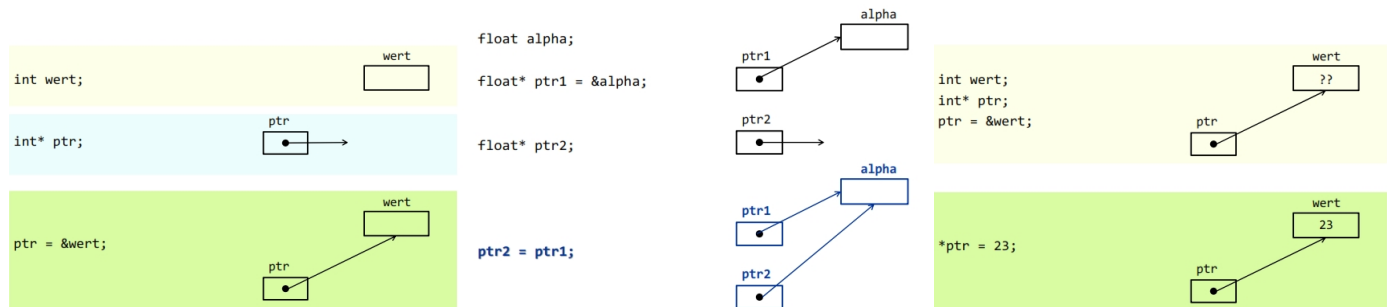
Ist  $ptr$  ein Pointer vom Typ  $Typname$ , so liefert der Ausdruck  $*ptr$  den Inhalt der Speicherzelle, auf welche  $ptr$  zeigt.

```

int wert;
// Variable wert vom Typ int wird definiert
int* ptr;
// Pointer ptr auf den Typ int wird definiert
// ptr zeigt auf eine nicht definierte
// Adresse
ptr = &wert;
// ptr zeigt nun auf die Variable wert, d.h.
// ptr enthaelt die Adresse der Variablen
// wert
*ptr = 23;
// in die Speicherzelle, auf welche ptr
// zeigt(hier: auf die Variable wert),
// wird 23 geschrieben. Aequivalent:
// wert = 23;

```

### 5.2.5 Beispiele Darstellung in graphischer Pointernotation



### 5.2.6 Pointer auf void

- Wenn bei der Definition des Pointers der Typ der Variablen, auf die der Pointer zeigen soll, noch nicht feststeht, wird ein Pointer auf den Typ *void* vereinbart.
- Ein Pointer auf *void* umgeht die Typenprüfung des Compilers. Er kann einem typisierten Pointer zugewiesen werden und er kann eine Zuweisung von einem typisierten Pointer erhalten.
- Abgesehen von einem Pointer auf *void*, darf ohne explizite Typenkonvertierung kein Pointer auf einen Datentyp an einem Pointer mit einem anderen Datentyp zugewiesen werden.
- Jeder Pointer kann durch Zuweisung in den Typ *void\** und zurück umgewandelt werden, ohne dass Informationen verloren gehen.

### 5.2.7 Pointerarithmetik Kapitel 10.1.1

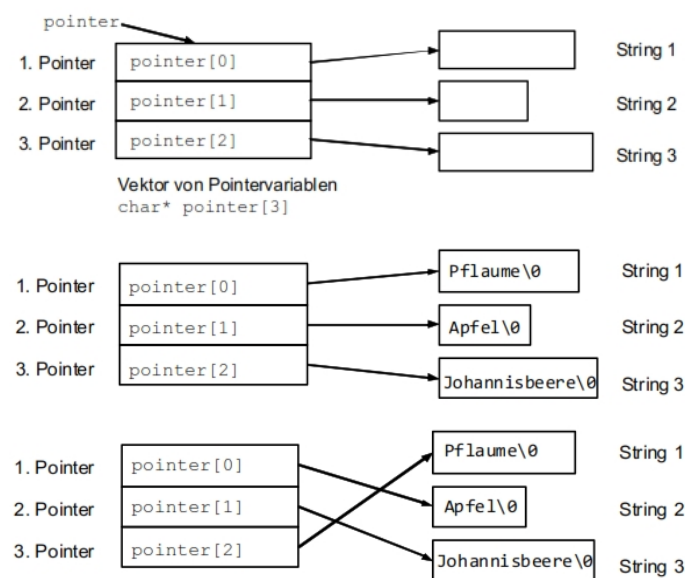
#### Zuweisung:

- Pointer unterschiedlicher Datentypen dürfen einander nicht zugewiesen werden (Schutzmechanismus).
- Einem Pointer eines bestimmten Typs dürfen Pointer dieses Typs oder *void*-Pointer zugewiesen werden.
- Einem *void*-Pointer dürfen beliebige Pointer zugewiesen werden (nützlich aber gefährlich).

#### Vergleiche:

- Bei Pointern desselben Typs funktionieren Vergleiche wie `==`, `!=`, `<`, `>`, `>=`, etc.
- Hintergrund: ein Pointer ist eine Adresse, d.h. die Vergleiche passieren mit den Adressen. Daraus ist klar, was die Vergleiche bewirken.

### 5.2.8 Vektoren von Pointern Kapitel 10.7.1



```
char* strTable[] = {"Pflaume",
                   "Apfel",
                   "Johannisbeere"};
```

### 5.2.9 Pointer auf Pointer Kapitel 10.7.2

```
#include <stdio.h>
void textAusgabe(char** textPointer,
                 int zeilen)
{
    int i;
    for (i = 0; i < zeilen; i++)
        printf("%s\n", *textPointer++);
}
int main(void)
{
    char* strTable[] = {"Pflaume",
                       "Apfel",
                       "Johannisbeere"};
    textAusgabe(strTable, 3);
    return 0;
}
```

#### Addition und Subtraktion:

- Zu einem Pointer darf eine ganze Zahl oder ein anderer Pointer desselben Typs addiert werden.
- Von einem Pointer kann eine ganze Zahl oder ein anderer Pointer desselben Typs subtrahiert werden.
- Wenn eine ganze Zahl *n* addiert / subtrahiert wird, so bewegt sich der Pointer auf das nächste Element des Pointertyps. Die Zahl *n* wird also nicht als Byte interpretiert, der Pointer bewegt sich um  $n * \text{sizeof}(\text{Typ})$  Bytes.

#### Andere Operationen:

- Andere Operationen sind nicht erlaubt!

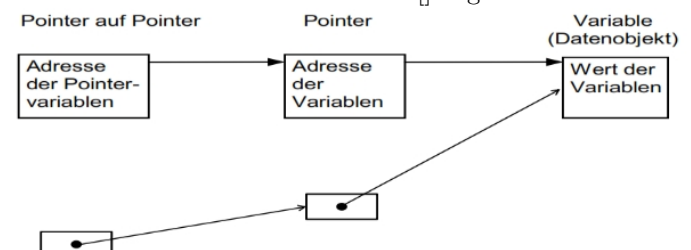
Ein Pointer ist eine Variable, in der die Adresse eines anderen Speicherobjektes gespeichert ist. Entsprechend einem eindimensionalen Vektor von gewöhnlichen Variablen kann natürlich auch ein eindimensionaler Vektor von Pointervariablen gebildet werden.

Arbeitet man mit mehreren Zeichenketten, deren Länge nicht von vornherein bekannt ist, so verwendet man ein Array von Pointern auf *char*.

Will man nun beispielsweise diese Strings sortieren, so muss dies nicht mit Hilfe von aufwändigen Kopieraktionen für die Strings durchgeführt werden. Es werden lediglich die Pointer so verändert, dass die geforderte Sortierung erreicht wird.

Formale Parameter für die Übergabe eines Arrays können in der Notation eines offenen Arrays ohne Längenangabe geschrieben werden. *strPointer*[] ist demzufolge ein Vektor. Der Vektor besteht aus Pointern auf *char*.

Die Schreibweisen *char \*\*textPointer* und *char \*textPointer*[] sind bei formalen Parametern gleichwertig. Bei der Übergabe eines Arrays wird als aktueller Parameter ein Pointer auf das erste Element eines Arrays übergeben, daher sind bei Übergabeparametern sowohl *\*textPointer* als auch *textPointer*[] zugelassen.



### 5.2.10 Pointer auf Funktionen **Kapitel 10.8**

- Jede Funktion befindet sich an einer definierten Adresse im Codespeicher.
- Diese Adresse kann ebenfalls ermittelt werden.
- Interessant wäre, dynamisch zur Laufzeit in Abhängigkeit des Programmablaufs eine unterschiedliche Funktion über einen Funktionspointer aufzurufen (z.B. um unterschiedliche Integrale zu berechnen).

```
#include <stdio.h>
int foo(char ch)
{
    int i;
    for (i = 1; i <= 10; i++)
        printf("%c_", ch);
    return i;
}
int main(void)
{
    int (*p)(char);
    // Deklaration des Funktionspointers
    int ret;
    p = foo;
    // ermittle Adresse der Funktion foo()
    ret = p('A');
    // Aufruf von foo() ueber Funktionspointer
    return 0;
}
```

#### Vereinbarung eines Pointers

```
int (*p)(char);
```

*ptr* ist hier ein pointer auf eine Funktion mit Rückgabewert vom Typ *int* und einem Übergabeparameter vom Typ *char*. Die Klammern müssen unbedingt gesetzt werden.

#### Zuweisung einer Funktion

```
p = funktionsname;
    \ \ oder
p = &funktionsname;
```

#### Aufruf einer Funktion

```
a = (*p) (Uebergabeparameter);
    \ \ oder
a = p (Uebergabeparameter);
```

## 5.3 Arrays **Kapitel 6.3**

Ein Array bietet eine kompakte Zusammenfassung von mehreren Variablen des gleichen Typs.

### 5.3.1 Definition eines Arrays **Kapitel 6.3**

```
Typname arrayName[groesse];
int data[10]; // ein Array von 10 int-Werten
int data[1000]; // ein Array von 1000 int-Werten
double zahl[5]; // ein Array von 5 double-Werten
```

### 5.3.2 Zeichenketten (Strings) **Kapitel 6.3**

- Ein String ist in C ein Array von Zeichen (*char*-Array).
- Ein String muss in C immer mit dem Nullzeichen '\0' abgeschlossen werden. Dieses benötigt eine Stelle im Array!

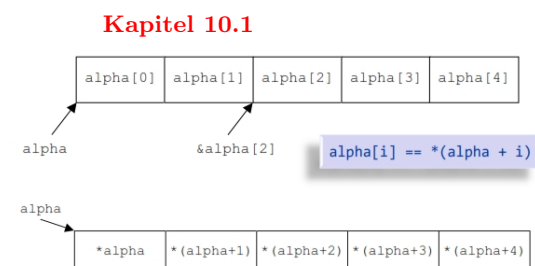
```
char name[15];
// Das letzte Element name[14] muss immer mit
// '\0' belegt sein.
```

### 5.3.3 Zugriff auf ein Arrayelement **Kapitel 6.3**

Der Zugriff auf ein Element eines Arrays erfolgt über den Array-Index. Ist ein Array mit *n* Elementen definiert, so ist darauf zu achten, dass in C der Index mit 0 beginnt und mit *n* - 1 endet.

```
int alpha[5];
// der Array 'alpha' mit 5 Elementen
// vom Typ int wird definiert
alpha[0] = 14;
// 1. Element (Index 0) wird auf 14
// gesetzt
alpha[4] = 3;
// das letzte Element (Index 4)
```

### 5.3.4 Äquivalenz von Array- und Pointernotation **Kapitel 10.1**



### 5.3.5 Vergleichen von Arrays **Kapitel 10.1**

- In C gibt es keinen Operator `==`, der zwei Arrays miteinander vergleicht.
- Arrayvergleiche müssen explizit Element um Element durchgeführt werden (z.B. mit Hilfe der Funktion `memcmp()`).

### 5.3.6 Der Arrayname **Kapitel 10.1**

- Der Arrayname ist ein nicht modifizierbarer L-Wert.
- Der Arrayname ist ein konstanter Pointer auf das erste Element des Arrays und kann nicht verändert werden.
- Auf den Arraynamen können nur die beiden Operatoren `sizeof` und `&` angewandt werden.
- Der Arrayname (z.B. `arr` bei `int arr[5]`), als auch der Adressoperator angewandt auf den Arraynamen (`&arr`) ergeben einen konstanten Pointer auf das erste Element des Arrays, d.h. sie ergeben dieselbe Adresse. Der Datentyp ist allerdings unterschiedlich:  
Der Typ von `arr` ist `int*`  
Der Typ von `&arr` ist `int (*)[5]` (Pointer auf Array mit 5 `int`'s)
- Einem Arraynamen kann kein Wert zugewiesen werden.

**5.3.7 Automatische Initialisierung Kapitel 10.1.3**

- Globale Arrays werden automatisch mit 0 initialisiert.
- Lokale Arrays werden nicht automatisch initialisiert.

**5.3.8 Explizite Initialisierung Kapitel 10.1.3**

- Bei der Definition eines Arrays kann ein Array explizit ("manuell") initialisiert werden.
- Nach der Initialisierung können die Elemente nur noch einzeln geändert werden.

```
int alpha[3] = {1, 2*5, 3};
```

- Werden bei der Initialisierung weniger Werte angegeben als der Array Elemente hat, so werden die restlichen Elemente mit 0 belegt.

```
int alpha[200] = {3, 105, 17};
// alpha[3] bis alpha[199] werden
// gleich 0 gesetzt
```

- Wird bei der Definition keine Arraygrösse angegeben, so zählt der Compiler die Anzahl Elemente automatisch (offenes Array oder Array ohne Längenangabe).

```
int alpha[] = {1, 2, 3, 4};
```

**5.3.11 Initialisierung von Zeichenketten Kapitel 10.1.5 und Kapitel 10.1.6**

```
char str[20] = {'Z', 'e', 'i', 'c', 'h', 'e', 'n', 'k', 'e', 't', 't', 'e', '\0'};
// umstaendlich
char str[20] = "Zeichenkette"; // bevorzugt
char str[20] = {"Zeichenkette"}; // unueblich
char str[] = "Zeichenkette"; // haeufig, Compiler soll chars zaehlen
```

**5.3.12 Übergabe von Arrays und Zeichenketten Kapitel 10.2**

- Bei der Übergabe eines Arrays an eine Funktion wird als Argument der Arrayname übergeben (i.e. Pointer auf erstes Element des Arrays).
- Der formale Parameter für die Übergabe eines eindimensionalen Arrays kann ein offenes Array sein oder ein Pointer auf den Komponententyp des Arrays.
- Die Grösse des Arrays muss immer explizit mitgegeben werden.
- Zeichenketten sind *char*-Arrays und werden deshalb gemäss der oben erwähnten Punkte gehandhabt.

```
enum {groesse = 3};
void init(int* alpha, int dim); /* hier ist alpha ein Pointer auf ein Array */
void ausgabe(int alpha[], int dim); /* hier ist alpha vom Typ eines offenen Arrays */
int main(void)
{
    int arr[groesse];
    init(arr, groesse);
    ausgabe(arr, groesse);
    return 0;
}
```

**5.3.13 Übergabe eines mehrdimensionalen Arrays**

```
void printMat(double mat[][cols], // Matrix
              int m,              // Anzahl Zeilen
              int n);             // Anzahl Spalten
```

**5.3.9 Mehrdimensionale Arrays Kapitel 10.1.4**

Das Array *int alpha[3][4]* kann folgendermassen aufgezeichnet werden:

	Spaltenindex			
Zeilenindex →	[0][0]	[0][1]	[0][2]	[0][3]
	[1][0]	[1][1]	[1][2]	[1][3]
	[2][0]	[2][1]	[2][2]	[2][3]

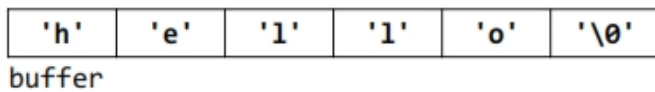
**5.3.10 Initialisierung eines mehrdimensionalen Arrays Kapitel 10.1.4**

```
int alpha[3][4] = {{1, 3, 5, 7},
                   {2, 4, 6, 8},
                   {3, 5, 7, 9}};
// aequivalent dazu ist die folgende
// Definition:
int alpha[3][4] = {1, 3, 5, 7, 2, 4,
                  6, 8, 3, 5, 7, 9};
```

### 5.3.14 Vergleich von char-Array und Pointer auf Zeichenkette Kapitel 10.3

char-Array:

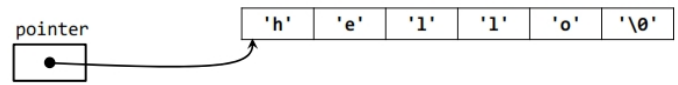
```
char buffer[] = "hello";
```



*buffer* entspricht der konstanten Anfangsadresse des Arrays.

Pointer auf Zeichenkette:

```
char* pointer = "hello";
```



*pointer* kann auf eine andere Adresse zeigen (ist nicht konstant). Zugriff auf "hello" könnte dadurch verloren gehen.

## 5.4 Das Schlüsselwort const bei Pointern und Arrays Kapitel 10.4

### 5.4.1 const bei Arrays

```
const int arr[] = {14, -2, 456};
```

*arr*[0], *arr*[1] und *arr*[2] sind alle konstant und können somit nach der Initialisierung nicht mehr abgeändert werden.

### 5.4.2 const bei Pointer - konstanter String

```
char str[] = "Ein_String";
const char* text = str;
// erlaubt
char ch = text[1];
text = "Ein_anderer_String";
str[4] = 'A';
// nicht erlaubt
text[1] = 's';
```

Dies bedeutet nicht, dass der Pointer *text* konstant ist, sondern dass *text* auf einen konstanten String zeigt.

### 5.4.3 const bei Pointer - konstanter Pointer

```
char str[] = "Ein_String";
char* const text = str;
// erlaubt
char ch = text[1];
text[1] = 's';
str[4] = 'A';
// nicht erlaubt
text = "Ein_anderer_String";
```

Hier ist nun der Pointer *text* konstant. Die Position von *const* ist sehr relevant!

### 5.4.4 const bei Pointer - konstanter Pointer auf konstanten String

```
char str[] = "Ein_String";
const char* const text = str;
// erlaubt
char ch = text[1];
str[4] = 'A';
// nicht erlaubt
text[1] = 's';
text = "Ein_anderer_String";
```

Bei dieser Variante ist sowohl der Pointer *text* als auch der String, auf welchen *text* zeigt, konstant.

## 6 Stringverarbeitung

- Üblicherweise werden Stringfunktionen aus Bibliotheken verwendet.
- Bei Speicherknappheit, lohnt es sich aber unter Umständen die Funktionen selber zu programmieren.

### 6.1 Kopieren eines Strings Kapitel 10.5

#### 6.1.1 Variante mit Laufvariable

```
char alpha[30] = "zu_kopierender_String";
char beta[30];
int i;
for (i = 0; beta[i] = alpha[i]; i++);
```

#### 6.1.2 Variante mit Pointer

```
char alpha[30] = "zu_kopierender_String";
char beta[30];
char* palpha = alpha;
char* pbeta = beta;
while (*pbeta++ = *palpha++);
```

### 6.2 Standardfunktionen für Strings und Speicher Kapitel 10.6

- Funktionen für die String- und Speicherverarbeitung sind prinzipiell dasselbe.
- Diese Funktionen werden in der Bibliothek *string.h* zur Verfügung gestellt.
- Funktionen die mit *str* beginnen, dienen der Stringverarbeitung und erkennen das '\0'-Zeichen.
- Funktionen die mit *mem* beginnen, dienen der Speicherverarbeitung und erkennen das '\0'-Zeichen nicht.

### 6.2.1 String kopieren **Kapitel 10.6.1.1**

```
#include <string.h>
char* strcpy(char* dest, const char* src);
```

- Dies Funktion kopiert einen String von *src* nach *dest* inklusive `'\0'`.
- Hat als Rückgabewert den Pointer auf *dest*.
- *dest* muss auf einen Bereich zeigen, der genügend gross ist. Ist der zu kopierende Buffer grösser als der Zielbuffer, dann werden nachfolgende Speicherbereiche überschrieben (Buffer overflow).

### 6.2.2 Strings zusammenfügen **Kapitel 10.6.1.2**

```
#include <string.h>
char* strcat(char* dest, const char* src);
```

- Diese Funktion hängt einen String *src* an *dest* an, inklusive `'\0'`. Das ursprüngliche `'\0'` von *dest* wird überschrieben.
- Hat als Rückgabewert den Pointer auf *dest*.
- *dest* muss auf einen Bereich zeigen, der genügend gross ist. Ist der zu kopierende Buffer grösser als der Zielbuffer, dann werden nachfolgende Speicherbereiche überschrieben (Buffer overflow).

## 6.3 Funktionen zur Speicherbearbeitung **Kapitel 10.6.2**

Die grundsätzlichen Unterschiede zu den Stringfunktionen sind:

- Formelle Parameter sind vom Typ *void\** statt *char\**.
- Die mem-Funktionen arbeiten byteweise.
- Im Gegensatz zu den *str*-Funktionen wird das `'\0'`-Zeichen nicht speziell behandelt.
- Die Bufferlänge muss als Parameter übergeben werden.

### 6.3.1 Funktionen **Kapitel 10.6.2.1 bis Kapitel 10.6.2.5**

```
#include <string.h>
// Speicherbereich kopieren
void* memcpy(void* dest, const void* src, size_t n);
// Speicherbereich verschieben
void* memmove(void* dest, const void* src, size_t n);
// Speicherbereiche vergleichen
void* memcmp(const void* s1, const void* s2, size_t n);
// Zeichen in Speicherbereich suchen
void* memchr(const void* s, int c, size_t n);
// Speicherbereich mit Wert belegen
void* memset(const void* s, int c, size_t n);
```

Bei *memcpy()* dürfen sich die Buffer nicht überlappen, *memmove()* kann auch mit überlappenden Buffern umgehen.

## 7 Lexikalische Konventionen, Enum

### 7.1 Bestandteile des Sourcecodes

#### 7.1.1 Lexikalische Elemente (“Text“)

- Befehle
- Variablen
- Deklarationen
- ...

#### 7.1.2 Trennzeichen

- Leerzeichen
- Tabulatoren
- Newline

### 6.2.3 Strings vergleichen **Kapitel 10.6.1.3**

```
#include <string.h>
int strcmp(const char* s1, const char* s2);
int strncmp(const char* s1, const char* s2,
size_t n);
```

- Dies Funktion vergleicht die beiden Strings, die auf *s1* und *s2* zeigen. Bei der Funktion *strncmp* werden nur die ersten *n* Zeichen verglichen.
- Dies Funktionen hat die folgenden Rückgabewerte:  
`< 0` : *s1* ist lexikographisch kleiner als *s2*  
`== 0` : *s1* und *s2* sind gleich  
`> 0` : *s1* ist lexikographisch grösser als *s2*

### 6.2.4 Stringlänge bestimmen **Kapitel 10.6.1.5**

```
#include <string.h>
size_t strlen(const char* s);
```

- Diese Funktion bestimmt die Länge von *s*, d.h. die Anzahl der *char*-Zeichen. Das `'\0'`-Zeichen wird dabei nicht mitgezählt.
- Hat als Rückgabewert die Länge von *s*.



## 7.2 Zeichenvorrat von C

Der Zeichenvorrat für einen Quelltext in C umfasst:

- Grosse und kleine Buchstaben, keine Umlaute: A - Z, a - z
- Unterstrich: `_`
- Ziffern: 0 - 9
- Leerzeichen (Blank, Space), Tabulator
- Strichpunkt (Semikolon): `;`

- Punkt (Dezimaltrennung, Selektionsoperator): `.`
- Sonderzeichen für Operatoren: `() [] <> + - * / % ^ ~ & | = ! ? :`
- Hochkomma, Doppelhochkomma: `' "`
- Geschweifte Klammern: `{ }`
- (Hash), Sharp, "Gartenhag": `#`
- Backslash: `\`

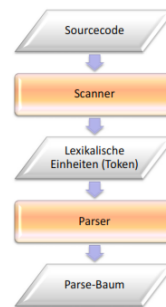
## 7.3 Lexikalische Analyse des Sourcecodes

### 7.3.1 Scanner

Zeichengruppen finden. Alle zusammengehörenden Zeichengruppen, welche nicht durch einen Trenner (z.B. Operator oder Trennzeichen) unterbrochen werden, gelten als lexikalische Einheiten (Token).

### 7.3.2 Parser

lexikalische Einheiten werden auf korrekte Syntax überprüft



## 7.4 Lexikalische Einheiten (Token)

### 7.4.1 Gross- und Kleinschreibung

- C unterscheidet Gross- und Kleinschreibung, d.h. C ist case-sensitive
- Die reservierten Wörter sind immer klein geschrieben
- *alpha* und *Alpha* sind unterschiedliche Namen
- Solche Feinheiten, bei denen sich zwei Namen nur in der Gross- bzw. Kleinschreibung einzelner Buchstaben unterscheiden, sollen unbedingt vermieden werden (fehleranfällig)

### 7.4.2 Kommentare

- Kommentare sollen die Verständlichkeit des Programms erhöhen
- Sie werden vom Preprocessor (allererster Compile-Schritt) entfernt. Sie sind im ausführbaren Code nicht mehr enthalten.
- Ein Kommentar muss mit `/*` und `*/` eingefasst werden
- Kommentare können auch über mehrere Zeilen reichen, sie dürfen aber nicht verschachtelt werden
- Zeilenkommentare beginnen mit einem Doppelslash `//`. Sie reichen ab dem Doppelslash bis zum Ende der Zeile

### 7.4.3 Namen

Namen bezeichnen in C:

- Variablen
- Funktionen
- Tags von Strukturen, Unions, Bitfeldern, Enumerations
- Komponenten von Strukturen
- Enum Konstanten
- Typnamen (typedef)
- Marken (Label)
- Makronamen (`#define`)

Namen können bestehen aus:

- Buchstaben a-z, A-Z
- Ziffern 0-9
- Underscore `_`

Das erste Zeichen eines Namens darf keine Ziffer sein!

Styleguide Variablen & Funktionen:

- mit Kleinbuchstaben beginnen
- erster Buchstaben von zusammengesetzten Wörtern ist gross
- keine Underscores

Beispiele: *counter*, *maxSpeed*, *getCount()*, *init()*, *setMaxSpeed()*

### 7.4.4 Reservierte Wörter

In ANSI C sind 32 reservierte Schlüsselwörter definiert. Sie sind stets klein geschrieben und dürfen nicht als Namen (z.B. für Variablen) verwendet werden.

- |                |                   |                 |                |                   |                 |                  |                   |
|----------------|-------------------|-----------------|----------------|-------------------|-----------------|------------------|-------------------|
| • <i>auto</i>  | • <i>const</i>    | • <i>double</i> | • <i>float</i> | • <i>int</i>      | • <i>short</i>  | • <i>struct</i>  | • <i>unsigned</i> |
| • <i>break</i> | • <i>continue</i> | • <i>else</i>   | • <i>for</i>   | • <i>long</i>     | • <i>signed</i> | • <i>switch</i>  | • <i>void</i>     |
| • <i>case</i>  | • <i>default</i>  | • <i>enum</i>   | • <i>goto</i>  | • <i>register</i> | • <i>sizeof</i> | • <i>typedef</i> | • <i>volatile</i> |
| • <i>char</i>  | • <i>do</i>       | • <i>extern</i> | • <i>if</i>    | • <i>return</i>   | • <i>static</i> | • <i>union</i>   | • <i>while</i>    |



### 7.4.5 Symbolische Konstanten

Haben Namen, der ihren Wert repräsentiert.

- Können mit dem Präprozessor-Befehl `#define` eingeführt werden:  

```
#define PI 3.1415
#define LIST_LENGTH 40
```
- Verhindern von "Magic numbers"
- Vereinfacht Änderungen
- Styleguide: Symbolische `#define`-Konstanten werden aus Grossbuchstaben und Underscores gebildet (keine Kleinbuchstaben)

### 7.4.7 Konstante Zeichenkette (String)

- begrenzt mit `""`
- (automatisch) abgeschlossen mit Nullzeichen `'\0'`

## 7.5 Enumerations (Aufzählungstyp)

```
enum State {idle,           // 0
            standby = 34,   // 34
            startingUp,     // 35
            running,        // 36
            blocked = 897,   // 897
            shuttingDown};   // 898
```

```
enum State s = idle;
```

### 7.5.1 Anonyme Enumerations

*enums* können auch verwendet werden, um ganzzahlige symbolische Konstanten zu definieren. Der *enum* erhält dann keinen Namen, er wird nur dazu verwendet, die einzelnen Konstanten festzulegen. Bessere Alternative zu `#define` für ganzzahlige Konstanten!!

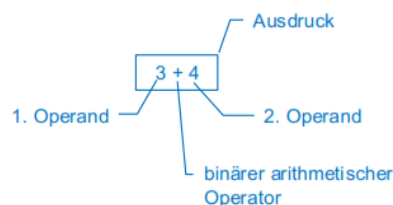
```
enum {listLength = 40,
      commandLength = 8,
      dateLength = 128};
```

## 8 Anweisungen, Ausdrücke und Operatoren

### 8.1 Operatoren und Operanden

#### 8.1.1 Stelligkeit der Operatoren

- Ein einstelliger (unärer) Operator hat einen einzigen Operanden wie z.B. der Minusoperator als Vorzeichenoperator.
- Benötigt ein Operator 2 Operanden für die Verknüpfung, so spricht man von einem zweistelligen (binären) Operator.



### 8.2 Postfix- und Präfixoperatoren

#### 8.2.1 Postfixoperatoren

Postfix-Operatoren sind unäre Operatoren, die nach (post) ihrem Operanden stehen.

```
printf("%d", i++);
```

*i* wird auf den Bildschirm geschrieben, anschliessend inkrementiert.

### 7.4.6 Literale Konstanten

Haben keinen Namen. Werden durch ihren Wert repräsentiert.

- Ganzzahlige Konstanten (default: int):  
254 (*dez*), 035 (*okt*), 0x3f (*hex*), -34, 14L (*long*), 14U (*unsigned*), 14UL (*unsignedlong*)
- Zeichenkonstanten:  
'*c*', '\n', '\x4a' (*ASCII hex*), '\014' (*ASCII okt*), '\\', *L'a*' (*DoubleByte*)
- Gleitpunktkonstanten (default: double):  
254.89, -13.0, 3.45e23 (*exp.Schreibweise*), 4.65f (*float – Konstante*), 3.14159L (*long double*)
- Aufzählungskonstanten

Beispiel "Ritchie"

'R'	'i'	't'	'c'	'h'	'i'	'e'	'\0'
0	1	2	3	4	5	6	7

- Aufzählungskonstanten haben einen konstanten ganzzahligen Wert.
- Die erste Konstante erhält den Wert 0, die zweite 1, etc.
- Werte können auch explizit zugewiesen werden

## 8.3 Ausdrücke und Anweisungen

### 8.3.1 Ausdrücke

- hat immer einen Rückgabewert (Typ ist durch Operanden bestimmt)
- kann Teil eines grösseren Ausdrucks sein
- kann eine Anweisung werden

Beispiel:  $3 \text{ (int)} + 4.5 \text{ (double)}$

### 8.3.2 Anweisungen

- kann keinen Rückgabewert haben
- kann nicht Teil eines grösseren Ausdrucks sein
- Selektions-, Iterations-, Sprung-, Ausdrucksanweisung

Beispiel: `while (a > 14)`

## 8.5 Auswertungsreihenfolge

### 8.5.1 Regeln

1. Wie in der Mathematik werden als erstes Teilausdrücke in Klammern ausgewertet.
2. Dann werden Ausdrücke mit unären Operatoren ausgewertet. Unäre Operatoren werden von rechts nach links angewendet, d.h.
  - (a) zuerst werden die Postfix-Operatoren auf ihre Operanden
  - (b) und dann die Präfix-Operatoren auf ihre Operanden angewendet.
3. Abschliessend werden Teilausdrücke mit mehrstelligen Operatoren gemäss der Priorität der Operatoren ausgewertet.
4. Bei gleicher Priorität der Operatoren entscheidet die Assoziativität (von links nach rechts oder von rechts nach links)

### 8.5.2 Prioritätstabelle

Priorität	Operatoren		Assoziativität
Priorität 1	()	Funktionsaufruf	links
	[]	Array-Index	links
	-> .	Komponentenzugriff	links
Priorität 2	! ~	Negation (logisch, bitweise)	rechts
	++ --	Inkrement, Dekrement	rechts
	sizeof		rechts
	+ -	Vorzeichen (unär)	rechts
	(typename)	cast	rechts
	* &	Dereferenzierung, Adresse	rechts
Priorität 3	* /	Multiplikation, Division	links
	%	modulo	links
Priorität 4	+ -	Summe, Differenz (binär)	links
Priorität 5	<< >>	bitweises Schieben	links
Priorität 6	< <=	Vergleich kleiner, kleiner gleich	links
	> >=	Vergleich größer, größer gleich	links
Priorität 7	== !=	Gleichheit, Ungleichheit	links
Priorität 8	&	bitweises UND	links
Priorität 9	^	bitweises Exklusives-ODER	links
Priorität 10		bitweises ODER	links
Priorität 11	&&	logisches UND	links
Priorität 12		logisches ODER	links
Priorität 13	?:	bedingte Auswertung	rechts
Priorität 14	=	Wertzuweisung	rechts
	+=, -=, *=, /=, %=, &=, ^=,  =, <<=, >>=	kombinierte Zuweisungsoperatoren	rechts
Priorität 15	,	Komma-Operator	links

## 8.4 Nebeneffekte

- Bei Nebeneffekten wird nebst der eigentlichen Aufgabe noch eine weitere Operation in dieselbe Anweisung gepackt (häufig mit Präfix- und Postfix Operatoren)
- zurückhaltend einsetzen!!

Beispiel:

$j = i + ++;$  ist äquivalent zu

$j = i;$

$i = i + 1; \quad //j ++;$

### 8.5.3 Beispiele zur Auswertungsreihenfolge

1.  $*p ++$

Zuerst wird der Pointer  $p$  inkrementiert, der Rückgabewert von  $p ++$  ist  $p!$  Dieser Rückgabewert (das noch nicht inkrementierte  $p$ ) wird nun dereferenziert.

2.  $* ++p$

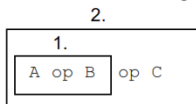
Zuerst wird der Pointer inkrementiert, der Rückgabewert von  $++p$  ist der inkrementierte Pointer, dann wird (das inkrementierte)  $p$  dereferenziert.

3.  $(*p) ++$

Zuerst wird der Pointer dereferenziert, anschliessend wird  $*p$ , d.h. der Inhalt der Speicherzelle, auf welche  $p$  zeigt, inkrementiert.

### 8.5.4 Assoziativität: links nach rechts oder rechts nach links?

- Ist meistens aufgrund der Operation intuitiv klar (von links nach rechts ist häufiger)
- Achtung: Assoziativität legt die Reihenfolge der Operatoren fest, sie sagt aber nichts über die Reihenfolge der Auswertung der Operanden aus



- Nebeneffekte sehr behutsam nützen!!

### 8.5.5 Beispiel Pointer mit Post- /Prefix- Inkrement

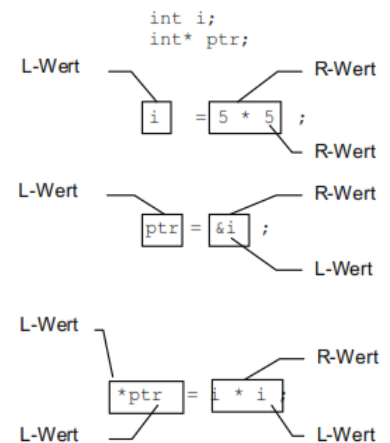
```
int zahl[4] = {4, 7, 66, 235};
int* p = &zahl[0];
```

Operation	*p	p
Start	4	0x22cca0
*p++	4	0x22cca4
*++p	66	0x22cca8
*(p++)	66	0x22ccac
(*p)++	234	0x22ccac
*pp	235	0x22ccac

## 8.6 L-Werte und R-Werte

- Ausdrücke haben eine unterschiedliche Bedeutung, je nachdem, ob sie links oder rechts vom Zuweisungsoperator stehen.
- Ein Ausdruck stellt einen L-Wert (lvalue oder left value) dar, wenn er sich auf ein Speicherobjekt bezieht. Ein solcher Ausdruck kann links (und rechts) des Zuweisungsoperators stehen.
- Ein Ausdruck, der sich nicht auf ein Speicherobjekt bezieht, kann nur rechts des Zuweisungsoperators stehen. Er wird als R-Wert (rvalue oder right value) bezeichnet. Einem R-Wert kann nichts zugewiesen werden.

```
a = 5 * a;    // ok
6 = a;       // nicht zulaessig, 6 ist kein lvalue
a * b = c;   // nicht zulaessig, (a*b) ist kein lvalue
b = 3++;     // nicht zulaessig, 3 ist kein lvalue
```



### 8.6.1 Zugriff auf L- und R-Werte

- Ein lvalue erfordert immer Schreibzugriff
- Auf einen rvalue wird nur lesend zugegriffen
- Es gibt auch nicht modifizierbare lvalues. Auf diese kann auch nur lesend zugegriffen werden.

## 8.7 Operatoren im einzelnen

### 8.7.1 Unäre arithmetische Operatoren

- Positiver Vorzeichenoperator  $+A$
- Negativer Vorzeichenoperator  $-A$
- Postfix-Inkrementoperator  $A++$
- Präfix-Inkrementoperator  $++A$
- Postfix-Dekrementoperator  $A--$
- Präfix-Dekrementoperator  $--A$

### 8.7.3 Zuweisungsoperatoren

- Zuweisungsoperator  $A = B$
- Kombinierte Zuweisungsoperatoren
  - Alle arithmetischen und logischen Operatoren haben zusammen mit dem Zuweisungsoperator eine verkürzte Form, die das Schreiben verkürzt (mehr nicht)
  - Beispiel:

```
a = a/b;
```

kann verkürzt geschrieben werden als

```
a /= b;
```

### 8.7.2 Binäre arithmetische Operatoren

- Additionsoperator  $A + B$
- Subtraktionsoperator  $A - B$
- Multiplikationsoperator  $A * B$
- Divisionsoperator  $A / B$
- Modulooperator  $A \% B$

### 8.7.4 Relationale Operatoren (Vergleichsoperatoren)

- Gleichheitsoperator  $A == B$
- Ungleichheitsoperator  $A != B$
- Grösseroperator  $A > B$
- Kleineroperator  $A < B$
- Grössergleichoperator  $A >= B$
- Kleinergleichoperator  $A <= B$

### 8.7.5 Logische Operatoren

- Logisch UND (AND)  $A \& B$
- Logisch ODER (OR)  $A || B$
- Logisch NICHT (NOT)  $!A$

0 = false, falsch

1 = true, wahr (genauer: ungleich 0)

### 8.7.7 Schiebe- (Shift-) Operatoren

- Rechts-Shift um n Bits  $A >> n$
- Links-Shift um n Bits  $A << n$

### 8.7.8 Bedingungsoperator (Ternärer Operator)

$A ? B : C$

Ist eine verkürzte Schreibweise für

```
if (A)
    B;
else
    C;
```

### 8.7.6 Bit-Operatoren

- Bitweises AND  $A \& B$
- Bitweises OR  $A | B$
- Bitweises NOT (Inverter)  $\sim A$
- Bitweises XOR  $A \wedge B$

Beispiel Maximum von zwei Zahlen a, b ermitteln:

```
int maximum = a > b ? a : b;
```

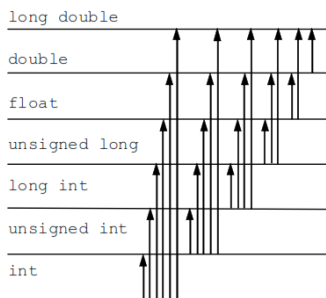
entspricht:

```
if (a > b)
    maximum = a;
else
    maximum = b;
```

## 8.8 Type Cast (Typumwandlungsoperator)

### 8.8.1 Implizite Typumwandlung

Bei der impliziten Typumwandlung wird Umwandlung nicht im Code aufgeführt. Sie wird vom Compiler automatisch anhand der Datentypen von Variablen bzw. Ausdrücken erkannt und durchgeführt.



Beispiel:

```
int i = 5; // i = 5
float f = i; // f = 5.000000
```

### 8.8.2 Explizite Typumwandlung

- Nebst den impliziten (automatischen) Typumwandlungen kann mit Hilfe des cast-Operators eine explizite Typumwandlung bewirkt werden.
- Der Programmierer ist verantwortlich, dass die Umwandlung keine Probleme ergibt. (z.B. Umwandlung von grosser Zahl in kleineren Typ)
- Syntax: (Zieltyp)Ausdruck

Beispiel:

```
int a = (int)4.6; // a == 4
double d = (double)a; // d == 4.0
```

### 8.8.3 Erlaubte Typenumwandlung

- Zwischen skalaren Typen (Integer, Floating Points, Pointer)
- Von skalarem Typ in void

## 8.9 Gültigkeitsbereiche von Namen (Scope)

- Compiler arbeitet dateiweise
- Namen in einer anderen Datei sind dem Compiler nicht bekannt
- (Globale) Variablen, welche in einer anderen Datei definiert werden, können mit Hilfe des extern-Statements bekannt gemacht werden
- Durch das extern-Statement wird kein Speicherplatz reserviert.  
*extern int Foo\_globalVariable;*
- Funktionsprototypen und Definitionen, die von anderen Modulen genutzt werden können (Schnittstellen), werden in einer Headerdatei definiert
- Durch *#include* der Headerdatei wird der Header geladen und die Namen somit bekannt gemacht.

## 9 Strukturen und Unionen

### 9.1 Strukturen

#### 9.1.1 Eigenschaften

- Daten, welche logisch zusammengehören, können zusammengefasst werden
- Die Struktur ist ein zusammengesetzter Datentyp, sie setzt sich aus den Feldern zusammen
- Die einzelnen Felder der Strukturen können (müssen aber nicht) unterschiedliche Typen haben
- Jedes Feld wird mit einem innerhalb der Struktur eindeutigen Namen versehen → Strukturspezifische Präfixe für die Feldnamen (z.B. *Angestellter\_Vorname*) sind deshalb sinnlos.

#### 9.1.2 Definition von Strukturtypen

```
struct StructName
{
    FieldType1    feld1;
    FieldType2    feld2;
    FieldType3    feld3;
    ...
    FieldTypeN    feldN;
};
```

- *StructName* kann frei gewählt werden
- *struct StructName* ist hier ein selbst definierter Typ, der weiter verwendet werden kann
- Der Datentyp ist definiert durch den Inhalt der geschweiften Klammer
- Der Feldtyp kann wiederum eine Struktur sein

#### 9.1.3 Beispiel

```
struct Adresse
{
    char strasse[20];
    int  hausnummer;
    int  plz;
    char ort[20];
};

struct Angestellter
{
    int  personalnummer;
    char name[20];
    char vorname[20];
    struct Adresse wohnort;
    struct Adresse arbeitsort;
    float gehalt;
};
```

#### 9.1.4 Beispiele für die Definition von Strukturvariablen

```
struct Angestellter mueller;
struct Angestellter bonderer;
```

```
struct Angestellter vertrieb[20];
// ein Array von 20 Strukturvariablen des Typs struct Angestellter
```

#### 9.1.5 Operationen auf Strukturvariablen

- Zuweisung: liegen zwei Strukturvariablen *a* und *b* vom gleichen Strukturtyp vor, so kann der Wert der einen Variablen der anderen zugewiesen werden → *a = b*;
- Ermittlung der Grösse der Struktur: mit *sizeof*-Operator
- Ermittlung der Adresse der Strukturvariablen: mit Adressoperator *&*

#### 9.1.6 Zugriff auf eine Strukturvariable und deren Felder

Der Zugriff auf ein Feld einer Strukturvariablen erfolgt über

- den Namen der Strukturvariablen,
- gefolgt von einem Punkt
- und dem Namen des Feldes

... wenn der Zugriff über einen Pointer auf eine Strukturvariable erfolgt, über

- den Namen des Pointers,
- gefolgt von einem Pfeil (*->*)
- und dem Namen des Feldes

### 9.1.7 Beispiele für den Zugriff auf eine Strukturvariable

```

struct Angestellter mueller;
struct Angestellter bonderer;

struct Angestellter vertrieb[20];

struct Angestellter* pMitarbeiter = &bonderer;

mueller.personalnummer = 34259;
bonderer.wohnort.plz = 7208;
strcpy(mueller.vorname, "Fritz");
printf("%s\n", vertrieb[14].name);

pMitarbeiter->personalnummer = 65433; // einfache Form bei Pointer
(*pMitarbeiter).personalnummer = 65433; // alternative Form

pMitarbeiter->arbeitsort.plz = 8640;

```

### 9.1.8 Lage im Speicher

- Die Felder einer Strukturvariablen werden nacheinander gemäss der Definition in den Speicher gelegt.
- Gewisse Datentypen verlangen unter Umständen, dass sie auf eine Wortgrenze (gerade Adresse) gelegt werden. Dies nennt man Alignment.
- Durch das Alignment kann es vorkommen, dass einzelne Bytes nicht verwendet werden, d.h. im Speicher ausgelassen werden.
- Die Grösse einer Strukturvariablen kann nicht durch Addieren der Grössen der Felder ermittelt werden, nur *sizeof()* liefert den genauen Wert

11	zahlLoLo
10	zahlLoHi
9	zahlHiLo
8	zahlHiHi
7	Not Used
6	text[2]
5	text[1]
4	text[0]
3	wertLoLo
2	wertLoHi
1	wertHiLo
0	wertHiHi

```

struct B
{
    int wert;
    char text[3];
    int zahl;
};

```

Das `int`-Feld `zahl` muss auf einer geraden Adresse beginnen!

### 9.1.9 Übergabe und Rückgabe von Strukturvariablen

- Strukturvariablen können komplett an Funktionen übergeben werden
- Der Rückgabetyt einer Funktion kann eine Struktur sein. Dabei wird die Strukturvariable direkt komplett übergeben
- Zu beachten ist der Kopieraufwand bei der Übergabe, bzw. Rückgabe eines Wertes. In der Praxis soll deshalb mit Pointern gearbeitet werden!

```

void foo(struct Angestellter a);
// grosser Kopieraufwand, nicht ideal

void foo(struct Angestellter* pa);
// nur Pointerübergabe, effizient

void fooRead(const struct Angestellter* pa)
// nur Pointerübergabe, read only durch const

```

### 9.1.10 Initialisierung einer Strukturvariablen

Eine Initialisierung einer Strukturvariablen kann direkt bei der Definition der Strukturvariablen mit Hilfe einer Initialisierungsliste durchgeführt werden (Reihenfolge beachten). Natürlich muss der Datentyp *struct Angestellter* bereits bekannt sein.

```
struct Angestellter maier =
{
    56321,           // personalnummer
    "Maier",         // name[20]
    "Hans",          // vorname[20]
    {
        "Schillerplatz", // strasse [20]
        14,              // hausnummer
        75142,           // plz
        "Esslingen"     // ort[20]
    }
};
```

## 9.2 Unions

### 9.2.1 Eigenschaften

- ähnlich wie Struktur
- beinhaltet auch mehrere Felder unterschiedlichen Typs
- im Gegensatz zur Struktur ist aber nur ein einziges Feld jeweils aktiv (abhängig vom Typ)
- Die Grösse einer *Union* ist so gross wie das grösste Feld der *Union*
- Bei der Union sind dieselben Operationen wie bei einer Struktur definiert

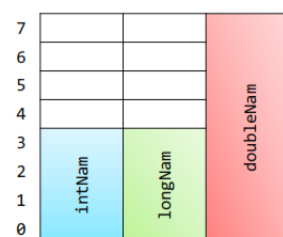
### 9.2.2 Definition von Uniontypen

```
union UnionName
{
    FeldTyp1   feld1;
    FeldTyp2   feld2;
    FeldTyp3   feld3;
    ...
    FeldTypN   feldN;
};
```

- *UnionName* kann frei gewählt werden
- *union UnionName* ist ein hier selbst definierter Typ, der weiter verwendet werden kann
- Der Datentyp ist definiert durch den Inhalt der geschweiften Klammer
- Der Feldtyp kann wiederum eine Union oder auch eine Struktur sein

### 9.2.3 Beispiel

```
union Vario
{
    int intNam;
    long longNam;
    double doubleNam;
}
```



## 9.3 Allgemeines zu Strukturen und Unions

### 9.3.1 Codierstil

- *Strukturname* und *Unionname* mit einem grossen Buchstaben beginnen!  
*struct Angestellter;*  
*union Vario;*
- Struktur- und Unionvariablen mit einem kleinen Buchstaben beginnen
- Bei Feldern von *Strukturen* und *Union* soll kein Präfix bei den Feldnamen verwendet werden

### 9.3.2 Vorsicht bei Unions

- Der Programmierer muss verfolgen, welcher Typ jeweils in der *Union* gespeichert ist. Der Datentyp, der entnommen wird, muss der sein, der zuletzt gespeichert wurde.



## 10 Komplexe Datentypen und Typennamen

### 10.1 Komplexere Vereinbarungen

Vereinbarungen in C können im Allgemeinen nicht stur von links nach rechts gelesen werden. Stattdessen muss man die Vorrang-Reihenfolge der Operatoren (siehe Kapitel 7) beachten.

#### 10.1.1 Array von Pointern

```
int* alpha[8];
```

[ ] hat Vorrang vor \*

*alpha* ist ein Array von 8 *int*-Pointern

#### 10.1.2 Pointer auf ein Array

```
int (*alpha)[8];
int* (*beta)[8];
```

( ) hat immer Vorrang

*alpha* ist ein Pointer auf ein Array von 8 *int*-Werten

*beta* ist ein Pointer auf ein Array von 8 *int*-Pointern

#### 10.1.3 Wie viel höher ist die Adresse von (alpha+1) ?

```
int main(void)
{
    int arr[5];
    int (*alpha)[5] = &arr;
    printf("Adresse_von_alpha: %p\n", alpha);
    printf("Adresse_von_(alpha+1): %p\n", alpha+1);
    return 0;
}
```

*alpha* ist ein Pointer auf einen Array von 5 *ints*,  
d.h. *alpha* zeigt auf einen Bereich, der  $5 * 4 = 20$  Bytes gross ist.

(*alpha*+1) liegt demnach 20 Bytes höher

#### 10.1.4 Funktion mit Pointer als Rückgabewert

```
int* foo(...);
```

( ) hat immer Vorrang

*foo* ist eine Funktion, welche einen *int*-Pointer zurückgibt

#### 10.1.5 Pointer auf eine Funktion

```
int (*pFunc)(...);
int* (*pFoo)(...);
```

( ) hat immer Vorrang, dann Assoziativität

*pFunc* ist ein Pointer auf eine Funktion, die ein *int* zurückgibt

*pFoo* ist ein Ponter auf eine Funktion, die einen *int*-Pointer zurückgibt

```
char* (*delta(...))[10];
```

( ) hat immer Vorrang, [ ] hat Vorrang vor \*

*delta* ist eine Funktion mit der Parameterliste (...) und gibt einen Pointer auf ein Array von 10 Pointern *char* zurück

### 10.2 Typdefinitionen

Bei der Definition einer Strukturvariablen muss immer das Schlüsselwort *struct* vorangestellt werden. Dies ist mühsam  
→ mit *typedef* elegantere Lösung

#### 10.2.1 Eigenschaften

- Vor allem bei komplexeren Typen (z.B. *structs*) macht es Sinn, einen eigenen Namen für den Typ zu definieren. Das Schlüsselwort *struct* kann anschliessend bei der Definition von Strukturvariablen weggelassen werden
- Eigene Typen können mit dem Befehl *typedef* definiert werden
- Zwischen dem Schlüsselwort *struct* und { wird üblicherweise kein zusätzlicher Name definiert
- *typedefs* werden im Notfall global, d.h. ausserhalb einer Funktion definiert
- eigene Typen sollen mit einem Grossbuchstaben beginnen

### 10.2.2 Strukturdefinition mit typedef

```
typedef enum {black, red, green, yellow, blue} Color;
typedef struct
{
    int x;
    int y;
} Point;
typedef struct
{
    Point p1;
    Point p2;
    Color col;
} Line;
int main (void)
{
    Line myLine = {{12, -34}, // p1
                  {783, 12}, // p2
                  yellow}; // col
    printf ("Anfangspunkt: (%d, %d)\n", myLine.p1.x, myLine.p1.y);
    printf ("Endpunkt: (%d, %d)\n", myLine.p2.x, myLine.p2.y);
    printf ("Farbe: %d\n", myLine.col);
    return 0;
}
```

### 10.2.3 Wie setzt der Compiler ein typedef um?

Ein *typedef* ist eine reine Textersetzung.

```
typedef struct
{
    int x;
    int y;
} Point;
```

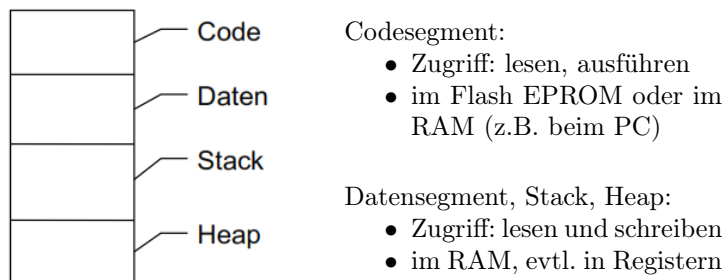
Überall im Code wo nun das Wort *Point* gefunden wird, ersetzt der Compiler dieses in einem ersten Durchgang mit dem Text

```
struct
{
    int x;
    int y;
}
```

## 11 Speicherklassen

### 11.1 Adressraum eines Programms

Der Adressraum eines ablauffähigen C-Programms - also einer Datei programm.exe - besteht aus den vier Segmenten: Code, Daten, Stack und Heap.



#### 11.1.1 Code

- Programm in Maschinencode

#### 11.1.2 Daten

- Globale Variablen, static Variablen

#### 11.1.3 Stack

- Lokale Variablen
- Parameter einer Funktion
- Rücksprungadressen

#### 11.1.4 Heap

- Dynamische Variablen (Speicherplatz wird erst zu Laufzeit angefordert)

#### 11.1.5 Anforderungen in grösseren Projekten

- Der gesamte Code soll in mehrere Dateien aufgeteilt werden können.
- Der Gültigkeitsbereich von Namen (Variablen, Typen, Funktionen) soll möglichst klein sein, die Namen sollen nicht im ganzen Projekt abgestimmt werden müssen.
- Die Schnittstellen zwischen den einzelnen Teilen (Dateien) sollen möglichst klein sein → stark entkoppeln

#### 11.1.6 Information Hiding

- Grundsatz: nur das gegen aussen preisgeben, das auch notwendig ist
- Interne Implementationsdetails nicht veröffentlichen → Funktionen, die in keiner weiteren Unit verwendet werden, sollen nur lokal bekannt sein
- Schnittstelle möglichst klein halten

## 11.2 Programme aus mehreren Dateien

### 11.2.1 Übersetzungsvorgang

- Quelldateien werden getrennt übersetzt
- Der Compiler erzeugt aus jeder Quelldatei eine Objektdaten mit Maschinencode
- Eine Objektdaten ist nicht ausführbar, da die Adressen der Funktionen (und globalen Variablen), die in einer anderen Datei definiert sind, noch nicht bekannt sind
- Der Linker löst genau diese noch offenen Referenzen auf, indem er alle zum ausführbaren Programm gehörenden Objektdaten linked (deutsch: bindet)
- Der Linker meldet genau dann einen Fehler, wenn er eine noch offene Referenz nirgends gefunden hat, bzw. nicht auflösen konnte

### 11.2.2 Sourcedatei (nur) compilieren

```
gcc -c foo.c
```

- Dadurch entsteht noch kein ausführbares Programm, sondern nur die *Datei foo.o*, der Objectcode
- Dies muss mit allen \*.c - Dateien gemacht werden

### 11.2.4 Buildprozess

Der Buildprozess beinhaltet alle Schritte, um ein ausführbares Programm zu erhalten.

```
gcc -c foo.c
gcc -c goo.c
gcc -c main.c
gcc -o prog.exe foo.o goo.o main.o
```

Es wäre mühsam, wenn diese Befehle jedesmal neu eingetippt werden müssten. Deshalb wird in der Praxis oft ein Buildscript oder ein Buildtool eingesetzt, z.B. make oder ant.

### 11.2.6 make-File

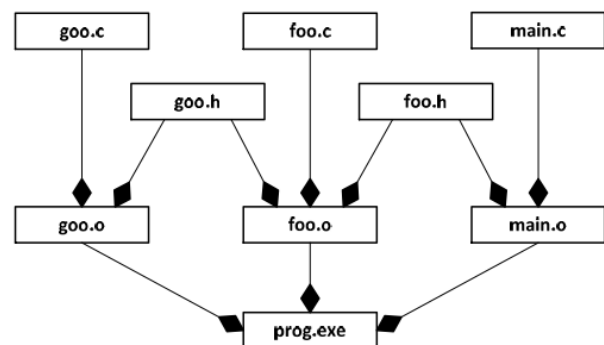
- In einem make-File können Abhängigkeiten definiert werden
- Wenn eine Datei geändert wurde, dann werden alle Operationen ausgeführt mit den Dateien, welche von dieser geänderten Datei abhängen
- Der Befehl (gcc) wird z.B. nur dann ausgeführt, wenn sich an den Dateien, zu denen eine Abhängigkeit besteht, etwas geändert hat

### 11.2.3 Objektdaten linken

```
gcc -o prog.exe foo.o goo.o main.o
```

- Alle Objektdaten müssen gelinkt werden

### 11.2.5 Abhängigkeiten zwischen Dateien



```
# makefile
cc=gcc
CFLAGS= -c -Wall
LFLAGS= -Wall
OBJ=foo.o goo.o main.o
EXE=prog.exe

$(EXE): $(OBJ)
$(cc) $(LFLAGS) -o $(@) $(OBJ)

foo.o: foo.h goo.h foo.c
$(cc) $(CFLAGS) -o $(@) foo.c

goo.o: goo.h goo.c
$(cc) $(CFLAGS) -o $(@) goo.c

main.o: foo.h main.c
$(cc) $(CFLAGS) -o $(@) main.c

clean:
rm -f $(OBJ) $(EXE)
```

**Targets:** Diese können mit make angesprungen werden, z.B. make goo.o

**Abhängigkeitsliste, d.h. von diesen Dateien ist das Target abhängig**

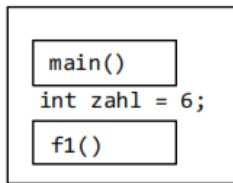
**Befehl, der ausgeführt werden muss, falls etwas geändert hat. Wichtig: erstes Zeichen muss ein TAB sein.**

### 11.2.7 Feste vs. relocative Adressen

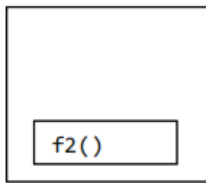
- Der Linker kann den Code auf feste physikalische (fixe) Adressen legen, oder
- auf virtuelle Adressen, welche erst beim Programmstart auf eine physikalische Adresse gelegt werden

## 11.3 Speicherklassen extern

Datei ext1.c



Datei ext2.c



```

// ext2.c
extern int zahl;    // zahl wird bekannt gemacht
...

```

## 11.4 Speicherklassen static

### 11.4.1 static Variablen

- static Variablen sind im Datenbereich, nicht auf dem Stack
- Gültigkeitsbereich ist der Block, in dem die Variable definiert ist
- static Variablen, welche ausserhalb einer Funktion definiert sind (globale Variablen), sind nur in der Datei gültig, in der sie definiert werden
- static Variablen sind nur einmal vorhanden (auch in Multithreading Umgebungen), d.h. ihr Wert wird erhalten, auch wenn die Funktion beendet ist. Beim nächsten Aufruf der Funktion geht es mit dem alten Wert weiter.
- Nur einsetzen, wenn man das will!  
`static int zahl = 0;`

## 11.5 Speicherklassen bei lokalen Variablen

### 11.5.1 Speicherklasse auto

- Variablen der Speicherklasse *auto* werden auf dem Stack angelegt
- Wenn bei einer Variablen keine Speicherklasse angegeben wird, so hat sie "automatisch" die Speicherklasse *auto*, d.h. *auto* ist der Default
- Die Speicherklasse *auto* wird praktisch nie explizit angegeben
- Automatische Variablen werden beim Verlassen des Blocks, in dem sie definiert wurden, vom System automatisch aus dem Speicher entfernt

### 11.5.2 Speicherklasse register

- Bei Angabe der Speicherklasse *register* wird dem Compiler empfohlen, für diese Variable ein Register (schnellste Speicherzelle) zu verwenden
- Der Compiler wird dies versuchen, eine Garantie besteht aber nicht, da dem Compiler z.B. u.U. gar nicht genügend Register zur Verfügung stehen
- Bei formalen Parametern ist das die einzig mögliche explizite Speicherklasse
- Der Compiler optimiert meist gut, *register* soll deshalb üblicherweise nicht angegeben werden

### 11.5.3 Speicherklasse static

- Wie bei allen Variablen der Speicherklasse *static* werden diese Variablen im globalen Datenbereich (nicht auf dem Stack) angelegt
- Das einzige spezielle von lokalen *static* Variablen gegenüber globalen *static* Variablen ist, dass die Sichtbarkeit auf den Block beschränkt ist, in welchem die Variable definiert ist
- Diese Einschränkung der Sichtbarkeit ist sehr zu begrüssen

### 11.3.1 externe Variablen

- Eine externe Variable kann nur in einer einzigen Datei definiert werden (ohne extern)
- In den anderen Dateien wird sie mit extern deklariert (bekannt gemacht)
- Eine manuelle Initialisierung ist nur bei der Definition möglich
- Globale Variablen, welche nicht manuell initialisiert werden, werden automatisch mit 0 initialisiert
- extern-Deklarationen werden üblicherweise in einer Headerdatei deklariert. Am Beginn der .c - Datei wird die Headerdatei mit *#include* eingefügt

### 11.4.2 static Funktionen

- static Funktionen sind nur in der Datei, in welcher sie definiert sind, sichtbar
- Alle Funktionen, welche nicht aussen (für andere Units) sichtbar sein sollen, sollten deshalb als static definiert werden
- Mit anderen Worten: alle Funktionen *static* definieren, ausser jene, welche die Schnittstelle nach aussen bilden

### 11.5.4 Initialisierung

- Automatische Variablen werden nicht durch das System initialisiert. Ihr Wert ist deshalb vor der manuellen Initialisierung undefiniert.
- Externe und statische lokale Variablen werden automatisch mit 0 initialisiert. Es ist guter Programmierstil, auch diese Variablen explizit zu initialisieren.

## 12 Rekursion und Iteration **Kapitel 9.8**

### 12.1 Unterschied von Rekursion und Iteration **Kapitel 9.8.1**

- Rekursion: Die Funktion enthält Abschnitte, in der sie selbst direkt oder indirekt wieder aufgerufen wird.
- Iteration: Ein Algorithmus enthält Abschnitte, die innerhalb einer Ausführung mehrfach durchlaufen werden (Schleife).
- Jeder rekursive Algorithmus kann auch iterativ formuliert werden.
- Die rekursive Form kann eleganter sein, ist aber praktisch immer ineffizienter als die iterative Form.
- Das Abbruchkriterium ist bei beiden Formen zentral.

### 12.2 Anwendung von rekursiven Funktionen **Kapitel 9.8.1**

- Wachstums-Vorgänge
- Backtracking-Algorithmen: z.B. Finden eines Weges durch ein Labyrinth (zurück aus Sackgasse und neuen Weg prüfen)
- Traversierungen von Suchbäumen
- In Mathematik rekursiv definierte Algorithmen

### 12.3 Beispiel anhand der Fakultätsberechnung **Kapitel 9.8.2**

#### 12.3.1 Iterativ

```
unsigned long faku(unsigned int n)
{
    unsigned long fak = 1UL;
    unsigned int i;
    for (i = 2; i <= n; ++i)
        fak = fak * i;
    return fak;
}
```

#### 12.3.2 Rekursiv

```
unsigned long faku(unsigned int n)
{
    if (n > 1)
        return n * faku(n-1);
    else
        return 1UL;
}
```

### 12.4 Beispiel anhand der Binärdarstellung **Kapitel 9.8.3**

#### 12.4.1 Iterativ

```
void binaerZahlIter (unsigned int zahl1)
{
    // Anzahl Bytes des Typs int
    int array [sizeof(int)*8] = {0};
    int zahl2;
    for (zahl2 = 0; zahl1; zahl2++, zahl1 /= 2)
        array[zahl2] = zahl1 % 2;
    for (--zahl2; zahl2 >= 0; zahl2--)
        printf ("%d_", array[zahl2]);
}
```

#### 12.4.2 Rekursiv

```
void binaerZahlReku (unsigned int zahl)
{
    if (zahl > 0)
    {
        binaerZahlReku (zahl / 2);
        printf ("%d_", zahl % 2);
    }
}
```

## 13 Anhang: Beispiele