1 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 Problemes, 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)*

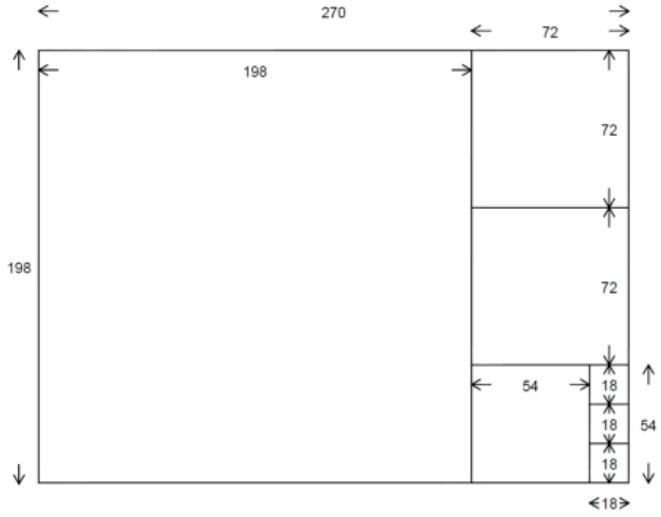
- eine **Menge von Objekten**, die durch den Algorithmus bearbeitet werden *(Zutaten, Geschirr, Herd, ...)*
- eine **Menge von Operationen**, die auf den Objekten ausgeführt werden *(waschen, schälen, ...)*
- ein **definierter Anfangszustand**, in dem sich die Objekte zu Beginn befinden *(Teller leer, 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?

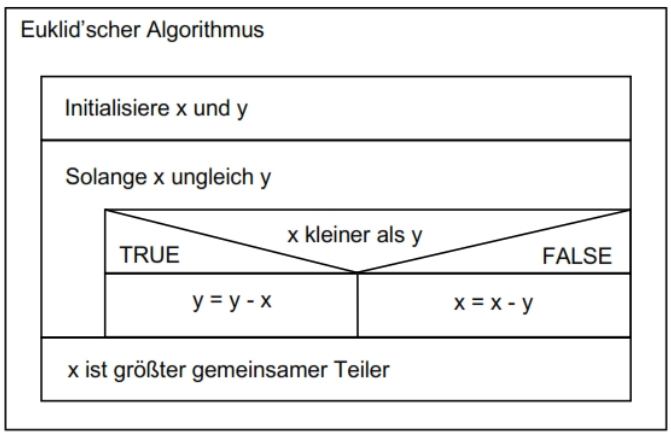
1.2.2 Der Algorithmus



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

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

1.2.3 Algorithmus-Beschreibung mit Struktogramm Kapitel 1.3



1.2.4 Algorithmus-Beschreibung mit Pseudocode Kapitel 1.2.1

```
Eingabe der Seitenlaengen: 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;
}
```

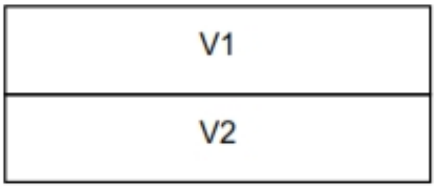
1.2.6 Trace-Tabelle Kapitel 1.2.4

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		

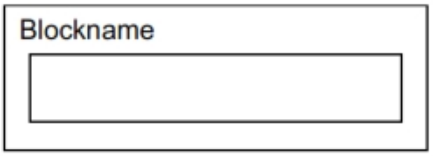
1.3 Nassi-Shneiderman-Diagramme Kapitel 1.3

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.

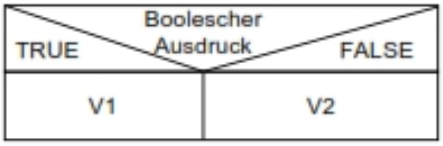
1.3.1 Sequenz



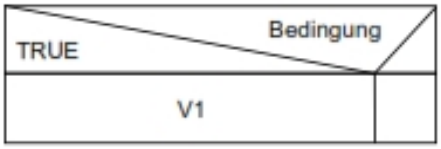
1.3.2 Block



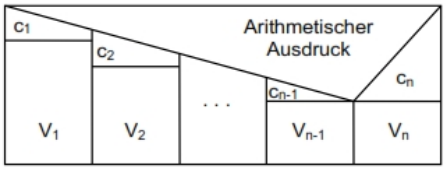
1.3.3 Einfache Alternative



1.3.4 Bedingte Anweisung

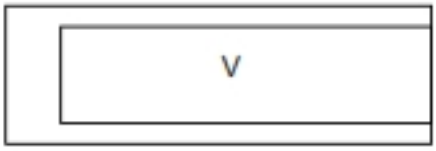


1.3.5 Mehrfache Alternative



1.3.6 Schleife mit vorheriger Prüfung

1.3.7 Endlosschlaufe

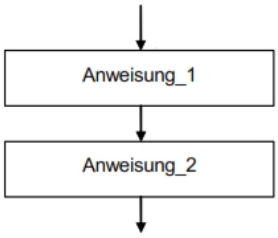


1.3.8 Schleife mit nachfolgender Prüfung

1.3.9 Abbruchanweisung

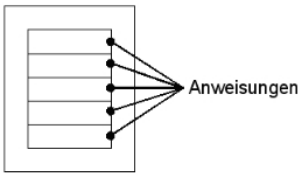
1.4 Sequenz Kapitel 8.1

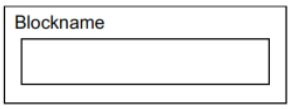
Die Sequenz ist eine zeitlich geordnete Abfolge von Anweisungen.



1.4.1 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.





1.5 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.

1.5.1 Einfache Alternative

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

1.5.2 Bedingte Anweisung

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

1.5.3 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);
```

1.5.4 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);
}
```

1.6 Iteration Kapitel 8.3

1.6.1 While

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

1.6.2 For-Schleife

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

1.6.3 Do-While

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

1.6.4 Endlosschleife

```
for (;;)
  Anweisung;
```

```
while (1)
  Anweisung;
```

1.6.5 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

1.7 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

2 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.

2.1 Übersicht über alle Standard-Datentypen Kapitel 5.2

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

2.1.1 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.

2.1.2 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 Standart IEEE 754 abgespeichert. Die Berechnung ist zu finden im Kapitel 5.2.3.

2.2 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

2.2.1 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;
```

2.2.2 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!

2.2.3 Manuelle Initialisierung von Variablen Kapitel 5.3.3

Jede einfache Variable kann bei ihrer Definition initialisiert werden:

```
int x = 5;
```

2.2.4 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.

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

2.2.5 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.

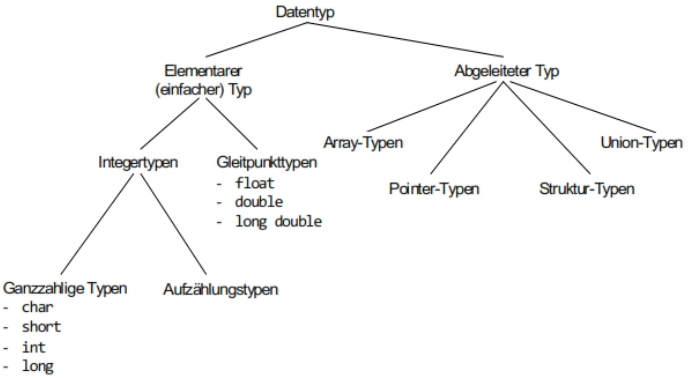
2.3 Typ-Attribute Kapitel 5.4

- 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)

2.4 Klassifikation von Datentypen Kapitel 5.5 und Kapitel 5.6



In der Programmiersprache C gibt es drei Klassen von Typen:

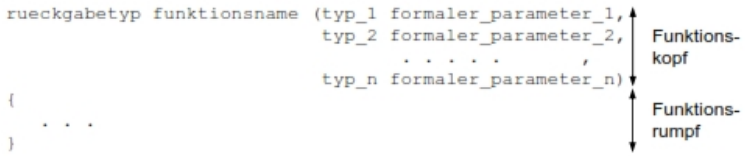
- Objekttypen (Datentypen): Objekttypen beschreiben Variablen,
  z.B. *int*
- Funktionstypen: Funktionstypen beschreiben Funktionen,
  z.B. *int f (void)*
- unvollständige Typen: Der Typ *void* ist ein unvollständiger Typ, der nicht vollständig gemacht werden kann. Er bezeichnet eine leere Menge und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

3 Funktionen

3.1 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.

3.2 Definition von Funktionen Kapitel 9.3.1



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

3.3 Eingaben/Ausgaben einer Funktion Kapitel 9.3

3.3.1 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

3.3.2 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

3.3.3 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
```

3.4 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.
Desweitern ist zu beachten, dass Parameternamen im Funktionsprototyp und in der Funktionsdefinition nicht übereinstimmen müssen. Es ist jedoch zu empfehlen.

3.4.1 Beispiel

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

3.4.2 Was passiert wenn der Prototyp vergessen geht?

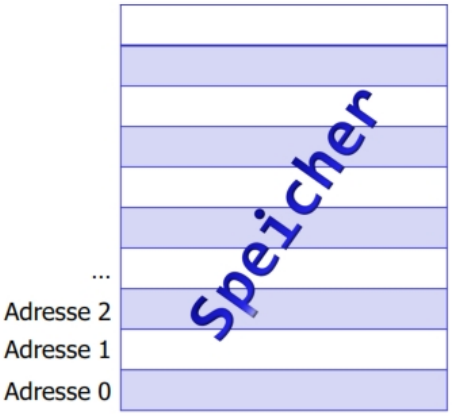
- Fehlt der Prototyp ganz, so wird die Funktion implizit (automatisch vom System) deklariert. Ihr Rückgabetyp wird als *int* angenommen, die Parameter werden nicht überprüft.
- Wenn die Funktion später definiert wird und nicht *int* als Rückgabetyp hat, bringt der Compiler eine Fehlermeldung.

3.4.3 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.

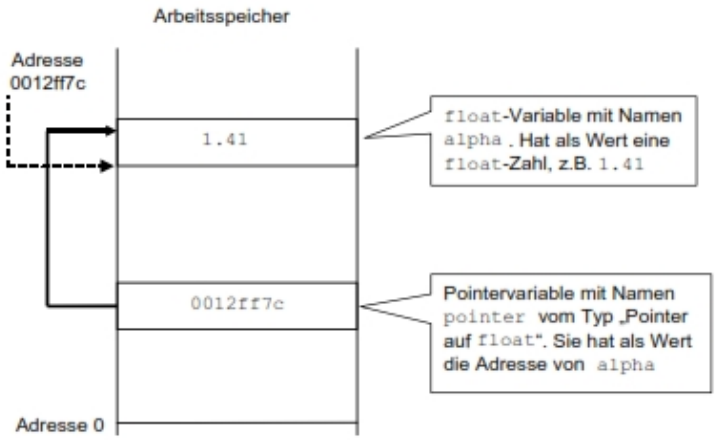
4 Pointer und Arrays Kapitel 6

4.1 Arbeisspeicher - Memory Map Kapitel 6.1



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

4.2 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).

4.2.1 Definition einer Pointervariablen Kapitel 6.1

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

4.2.2 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;
```

4.2.3 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.

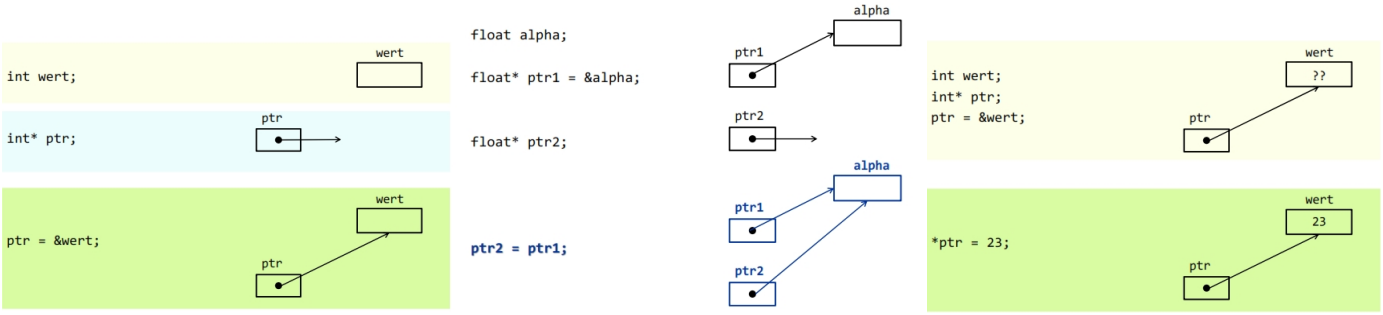
```
int wert;
  // Variable wert vom Typ int wird
  // definiert
int* ptr;
  // Pointer ptr auf den Typ int wird
  // definiert
  // ptr zeigt auf eine nicht definierte
  // Adresse
ptr = &wert;
  // ptr zeigt nun auf die Variable wert,
  // d.h. ptr enthaelt die Adresse der
  // Variablen wert
```

4.2.4 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;
```

4.2.5 Beispiele Darstellung in graphischer Pointernotation



4.2.6 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.

4.2.7 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.

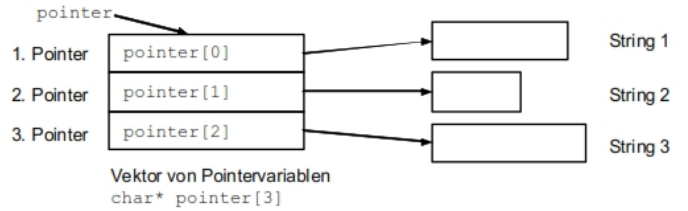
Addition und Subtraktion:

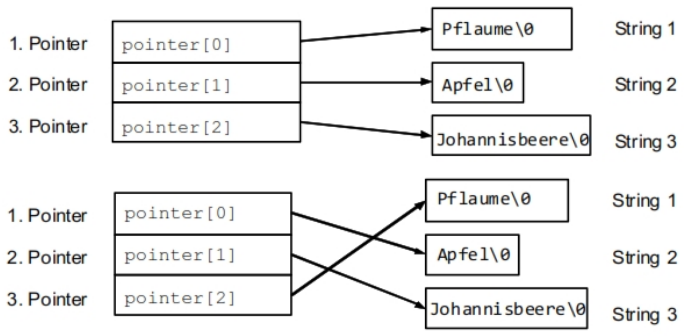
- Zu einem Pointer darf eine ganze Zahl oder ein anderer Pointer desselben Typs addiert werden.
- Von einem Pointer kann eine ganze Zahl oder ein anderer Pointer desselben Typs subtrahiert werden.
- Wenn eine ganze Zahl n addiert / subtrahiert wird, so bewegt sich der Pointer auf das nächste Element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

Andere Operationen:

- Andere Operationen sind nicht erlaubt!

4.2.8 Vektoren von Pointern Kapitel 10.7.1





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 vorherein 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.

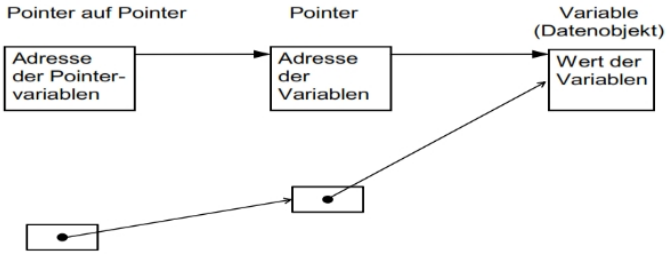
```
char* strTable[] = {"Pflaume",
                    "Apfel",
                    "Johannisbeere"};
```

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*.

4.2.9 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;
}
```

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.



4.2.10 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);
```

4.3 Arrays Kapitel 6.3

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

4.3.1 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
```

4.3.2 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!

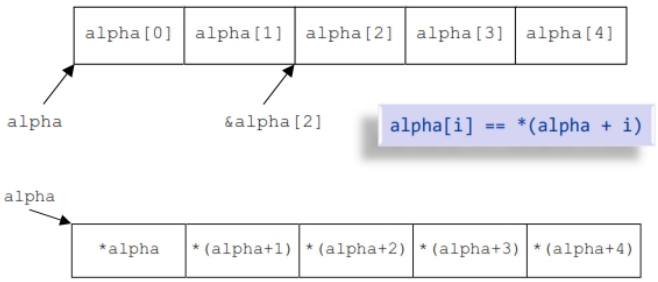
```
char name[15];
  // Das letzte Element name[14] muss immer mit
  // '\0' belegt sein.
```

4.3.3 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)
```

4.3.4 Äquivalenz von Array- und Pointernotation Kapitel 10.1



4.3.5 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 oder der Inhalt der beiden Speicherbereiche wird mit Hilfe der Funktion *memcmp*() byteweise verglichen.

4.3.7 Automatische Initialisierung Kapitel 10.1.3

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

4.3.8 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).

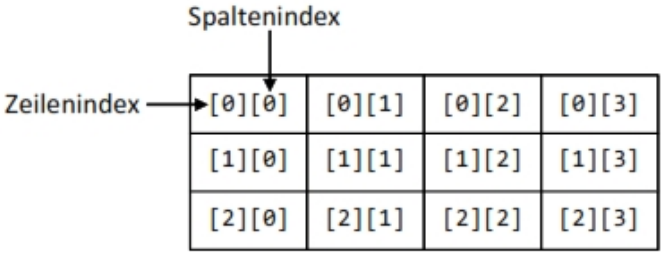
```
int alpha[] = {1, 2, 3, 4};
```

4.3.6 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 (einer Pointervariablen schon).

4.3.9 Mehrdimensionale Arrays Kapitel 10.1.4

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



4.3.10 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};
```

4.3.11 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
```

4.3.12 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;
}
```

4.3.13 Übergabe eines mehrdimensionalen Arrays

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

4.3.14 Vergleich von char-Array und Pointer auf Zeichenkette Kapitel 10.3

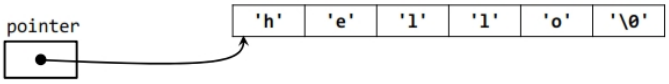
char-Array:

```
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.

4.4 Das Schlüsselweort const bei Pointern und Arrays Kapitel 10.4

4.4.1 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.

4.4.2 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.

4.4.3 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!

4.4.4 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.

5 Stringverarbeitung

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

5.1 Kopieren eines Strings Kapitel 10.5

5.1.1 Variante mit Laufvariable

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

5.1.2 Variante mit Pointer

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

5.2 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. Aus diesem Grund muss die Bufferlänge in Byte ebenfalls als Parameter übergeben werden.

5.2.1 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).

5.2.2 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).

5.2.3 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 $*s2$

5.2.4 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.

5.3 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.

5.3.1 Funktionen Kapitel 10.6.2.1 bis Kapitel 10.6.2.5

```c
#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.

6 Lexikalische Konventionen, Enum, Anweisungen und Operatoren

T. Andermatt

7 Strukturen und Unionen

7.1 Strukturen

7.1.1 Eigenschaften

- Daten, welche logisch zusammengehören, können zusammengenommen 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.

7.1.2 Definition von Strukturtypen

```
struct StructName
{
 FeldTyp1   feld1;
 FeldTyp2   feld2;
 FeldTyp3   feld3;
 ...
 FeldTypN   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

7.1.3 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;
};
```

7.1.4 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
```

7.1.5 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 &

7.1.6 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

7.1.7 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;
```

7.1.8 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


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

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

7.1.9 Übergabe und Rückgabe von Strukturvariablen

- Strukturvariablen können komplett an Funktionen übergeben werden
- Der Rückgabetyp 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 Pointeruebergabe, effizient

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

7.1.10 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]
 }
};
```

7.2 Unions

7.2.1 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

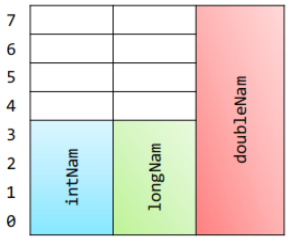
7.2.2 Definition von Uniontypen

```
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

7.2.3 Beispiel

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



7.3 Allgemeines zu Strukturen und Unions

7.3.1 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

7.3.2 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.

8 Komplexe Datentypen und Typennamen

8.1 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.

8.1.1 Array von Pointern

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

[ ] hat Vorrang vor *
alpha ist ein Array von 8 int-Pointern

8.1.2 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

8.1.3 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

8.1.4 Funktion mit Pointer als Rückgabewert

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

( ) hat immer Vorrang
foo ist eine Funktion, welche einen int-Pointer
zurückgibt

8.1.5 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

8.2 Typdefinitionen

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

8.2.1 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

8.2.2 Strukturdefinition mit typedef

Beispiel 1:

```
typedef struct
{
 int nummer;          // Artikelnummer
 char name[80];       // Artikelname (Bezeichnung)
 double preis;        // Artikelpreis
} Artikel;

int main(void)
{
 Artikel art;
 Artikel* pArt;
}
```

Beispiel 2:

```
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;
}
```

8.2.3 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;
}
```

9 Speicherklassen

T. Andermatt

10 Input/Output

T. Andermatt

11 Rekursion und Iteration Kapitel 9.8

11.1 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.

11.2 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

11.3 Beispiel anhand der Fakultätsberechnung Kapitel 9.8.2

11.3.1 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;
}
```

11.3.2 Rekursiv

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

11.4 Beispiel anhand der Binärdarstellung Kapitel 9.8.3

11.4.1 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]);
}
```

11.4.2 Rekursiv

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

12 Diverses

13 Anhang: Beispiele