

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*)

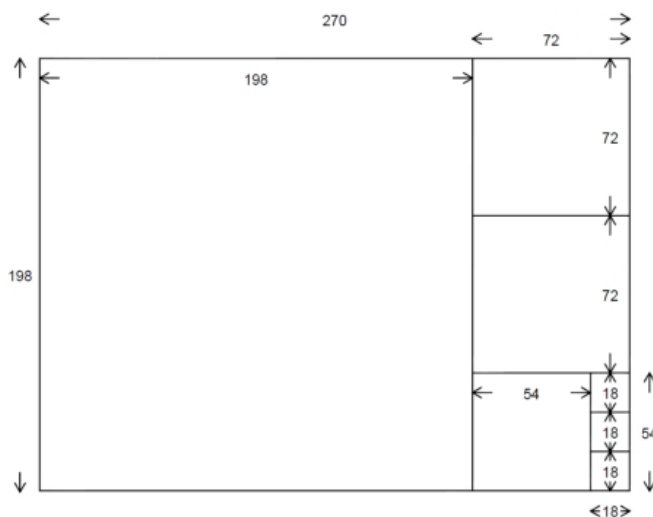
- eine **Menge von Objekten**, die durch den Algorithmus bearbeitet werden (*Zutaten, Geschirr, Herd, ...*)
- eine **Menge von Operationen**, die auf den Objekten ausgeführt werden (*waschen, schälen, ...*)
- ein **definierter Anfangszustand**, in dem sich die Objekte zu Beginn befinden (*Teller leer,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

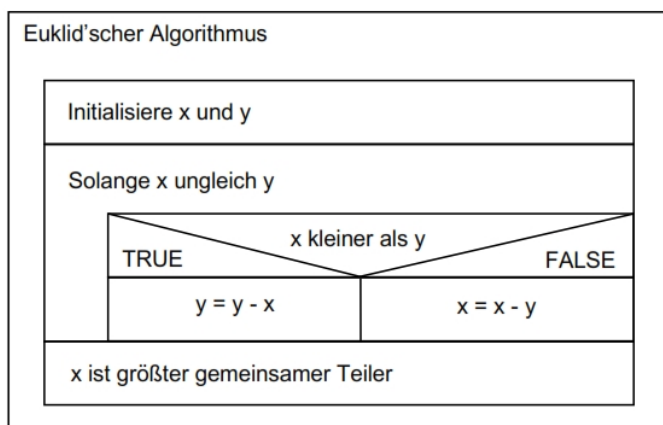
1.2.2 Der Algorithmus



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 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;
}
```

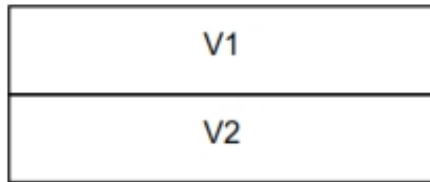
1.2.6 Trace-Tabelle **Kapitel 1.2.4**

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		

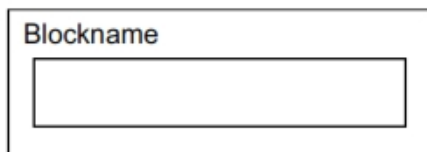
1.3 Nassi-Shneiderman-Diagramme Kapitel 1.3

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.

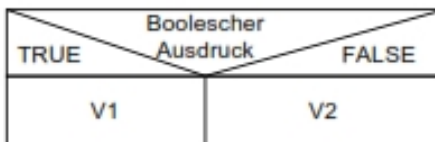
1.3.1 Sequenz



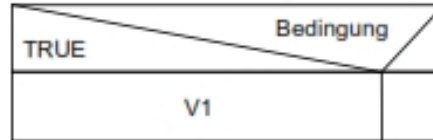
1.3.2 Block



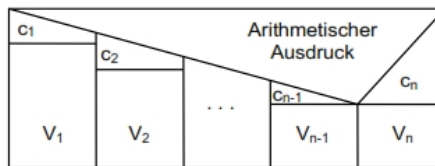
1.3.3 Einfache Alternative



1.3.4 Bedingte Anweisung



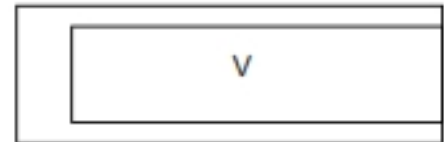
1.3.5 Mehrfache Alternative



1.3.6 Schleife mit vorheriger Prüfung



1.3.7 Endlosschleife



1.3.8 Schleife mit nachfolgender Prüfung



1.3.9 Abbruchanweisung



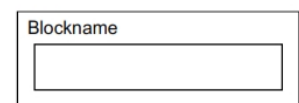
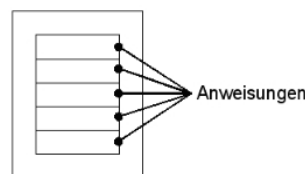
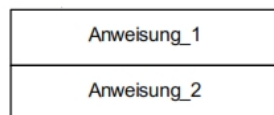
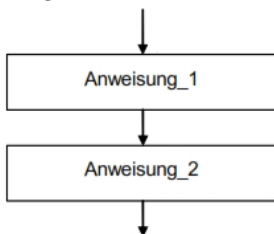
2 Kontrollstrukturen Kapitel 8

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 ³⁸ bis +3.4 * 10 ³⁸	Gleitpunkttyp	Gleitpunktzahl
<i>double</i>	8 (in der Regel)	−1.7 * 10 ³⁰⁸ bis +1.7 * 10 ³⁰⁸	Gleitpunkttyp	höhere Genauigkeit
<i>long double</i>	4 (in der Regel)	−1.1 * 10 ⁴⁹³² bis +1.1 * 10 ⁴⁹³²	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

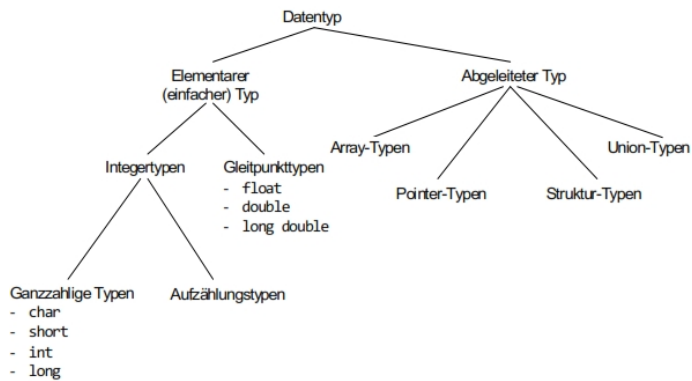
- Globale (externe) Variablen: Diese Variablen stehen allen Funktionen zur Verfügung und müssen ausserhalb von Funktionen definiert werden.
- Lokale (interne) Variablen: Diese Variablen stehen nur der Funktion zur Verfügung, in welcher die definiert wurden. Sie kann nicht von ausserhalb angesprochen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

```

rueckgabetyf 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?

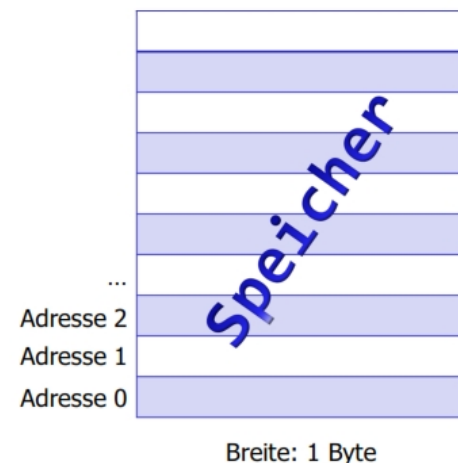
- Fehlt der Prototyp ganz, so wird die Funktion implizit (automatisch vom System) deklariert. Ihr Rückgabotyp wird als *int* angenommen, die Parameter werden nicht überprüft.
- Wenn die Funktion später definiert wird und nicht *int* als Rückgabotyp hat, bringt der Compiler eine Fehlermeldung.

4.4.3 Funktionsprototypen in der Praxis **Kapitel 9.4**

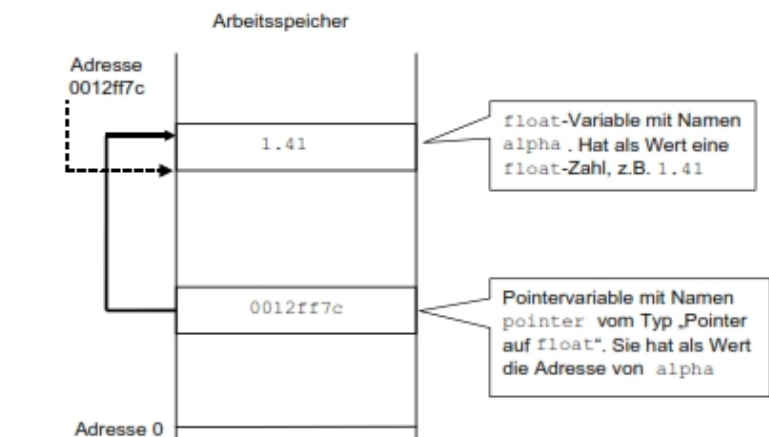
- Funktionsprototypen, welche die Schnittstelle der Unit beschreiben, kommen in das entsprechenden Headerfile.
- Jedes C-File, welches diese Schnittstelle nutzt, inkludiert dieses Headerfile und somit die Funktionsprototypen.
- Funktionsprototypen von internen Funktionen der Unit werden zuoberst im C-File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



- 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**

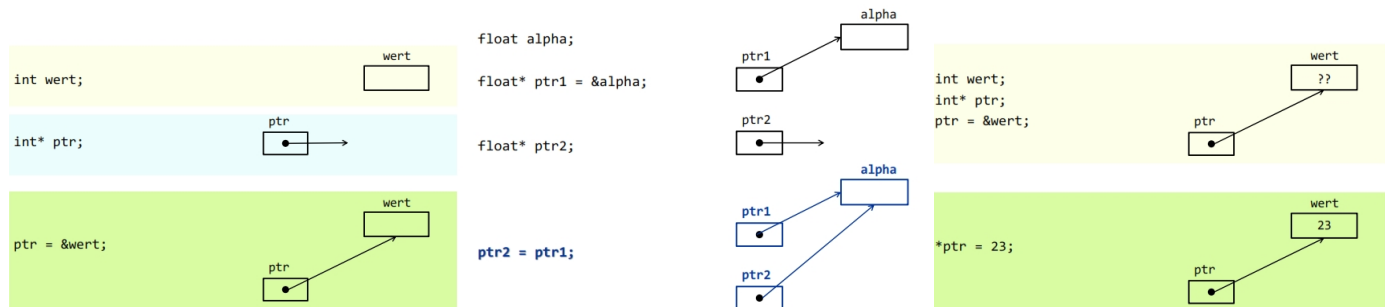
Ist ptr ein Pointer vom Typ $Typname$, so liefert der Ausdruck $*ptr$ den Inhalt der Speicherzelle, auf welche ptr zeigt.

```

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 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 $n * \text{sizeof}(Typ)$ Bytes.

Andere Operationen:

- Andere Operationen sind nicht erlaubt!

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

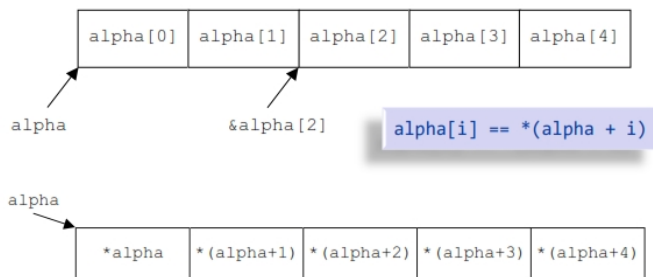
```
Typname arrayName[groesse];
int data[10];    // ein Array von 10 int-Werten
int data[1000]; // ein Array von 1000 int-Werten
double zahl[5]; // ein Array von 5 double-Werten
```

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.
```

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

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};
```

5.3.3 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)
```

5.3.6 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).

5.3.9 Mehrdimensionale Arrays Kapitel 10.1.4

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

		Spaltenindex			
Zeilenindex	[0]	[0][0]	[0][1]	[0][2]	[0][3]
	[1]	[1][0]	[1][1]	[1][2]	[1][3]
	[2]	[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}};
// äquivalent dazu ist die folgende
// Definition:
int alpha[3][4] = {1, 3, 5, 7, 2, 4,
                  6, 8, 3, 5, 7, 9};
```


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', '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

```