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)*

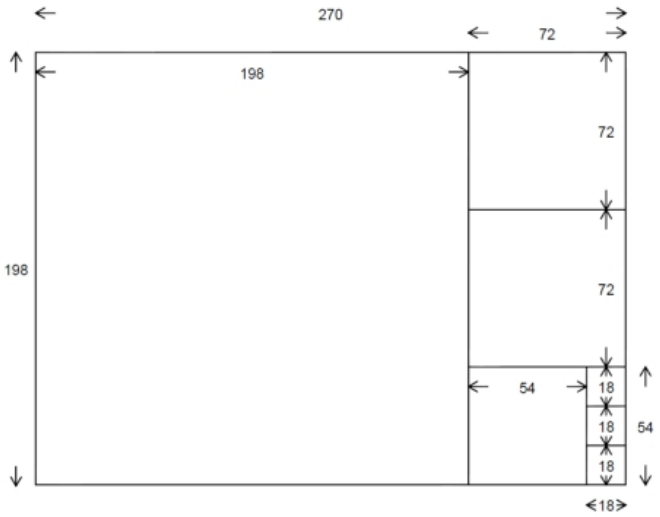
- eine **Menge von Objekten**, die durch den Algorithmus bearbeitet werden *(Zutaten, Geschirr, Herd, ...)*
- eine **Menge von Operationen**, die auf den Objekten ausgeführt werden *(waschen, schälen, ...)*
- ein **definierter Anfangszustand**, in dem sich die Objekte zu Beginn befinden *(Teller leer,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

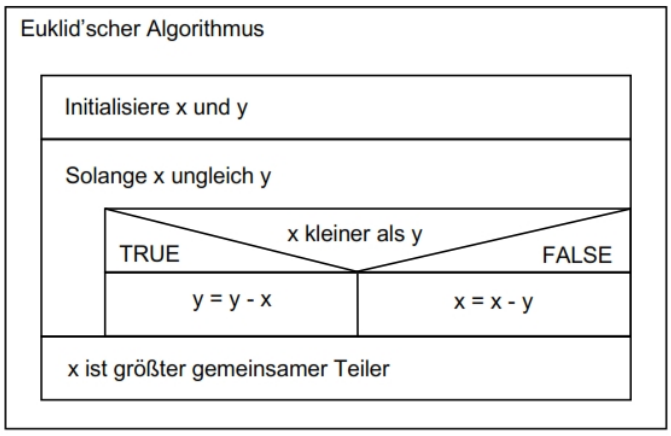
1.2.2 Der Algorithmus



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;
}
```

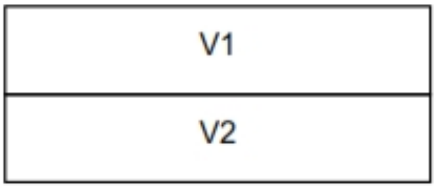
1.2.6 Trace-Tabelle Kapitel 1.2.4

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		

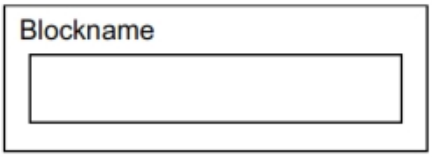
1.3 Nassi-Shneiderman-Diagramme Kapitel 1.3

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.

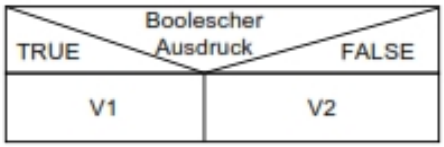
1.3.1 Sequenz



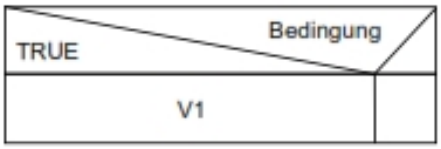
1.3.2 Block



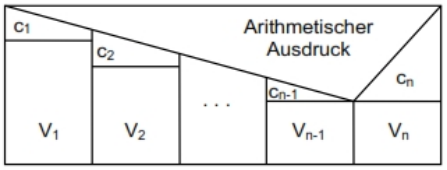
1.3.3 Einfache Alternative



1.3.4 Bedingte Anweisung

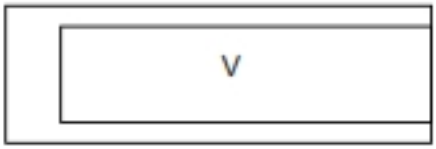


1.3.5 Mehrfache Alternative



1.3.6 Schleife mit vorheriger Prüfung

1.3.7 Endlosschlaufe



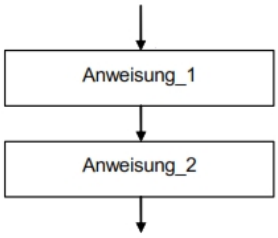
1.3.8 Schleife mit nachfolgender Prüfung

1.3.9 Abbruchanweisung

2 Kontrollstrukturen Kapitel 8

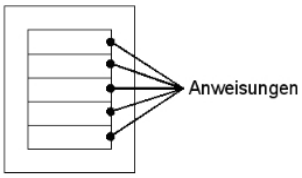
2.1 Sequenz Kapitel 8.1

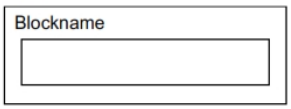
Die Sequenz ist eine zeitlich geordnete Abfolge von Anweisungen.



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
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

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 Standart 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;
```

3.2.2 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!

3.2.3 Manuelle Initialisierung von Variablen Kapitel 5.3.3

Jede einfache Variable kann bei ihrer Definition initialisiert werden:

```
int x = 5;
```

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.

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.

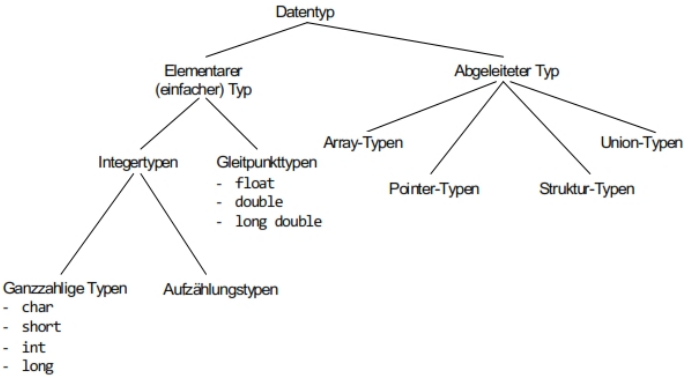
3.3 Typ-Attribute Kapitel 5.4

- 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)

3.4 Klassifikation von Datentypen Kapitel 5.5 und Kapitel 5.6



In der Programmiersprache C gibt es drei Klassen von Typen:

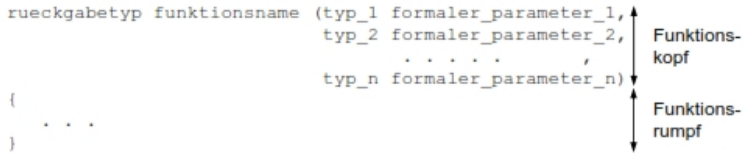
- Objekttypen (Datentypen): Objekttypen beschreiben Variablen,
  z.B. *int*
- Funktionstypen: Funktionstypen beschreiben Funktionen,
  z.B. *int f (void)*
- unvollständige Typen: Der Typ *void* ist ein unvollständiger Typ, der nicht vollständig gemacht werden kann. Er bezeichnet eine leere Menge und 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



- Funktionskopf: legt die Aufrufschnittstelle (Signatur) der Funktion fest. Er besteht aus Rückgabetyp, 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 Rückgabewer:t

```
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.

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 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.

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.