

## 4 Aufbau eines Programmes in C



Die Programmiersprache C gehört zu der Klasse der **prozeduralen Programmiersprachen**.


Ein prozedurales **Programm** besteht aus Funktionen, die auf Daten arbeiten.



In diesem Kapitel wird auf die Frage eingegangen, was genau Daten sind, von einzelnen Zeichen der Tastatur bis hin zu Werten mit unterschiedlichen Datentypen, und wie sie gespeichert werden in Variablen im Arbeitsspeicher.

Auch wird gezeigt, wie schlussendlich all diese Werte und Variablen in Funktionen übergeben, verarbeitet und wieder zurückgegeben werden, und wie ein Programm insgesamt aufgebaut sein muss, damit alle Teile zusammenpassen und korrekt miteinander interagieren können.

### 4.1 Zeichen und Codierungen

Wenn ein Programm mit Hilfe eines Texteditors geschrieben wird, werden Zeichen über die Tastatur eingegeben. Einzelne oder mehrere aneinandergereihte Zeichen haben im Programm eine spezielle Bedeutung. So repräsentierten beispielsweise die Zeichen x und y bei der Implementierung des euklidischen Algorithmus in Kapitel  3.4 die Namen von Variablen.

Ein „**Zeichen**“ ist ein von anderen Zeichen unterscheidbares Objekt, welches in einem bestimmten Zusammenhang eine definierte Bedeutung trägt.



**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-45209-4\\_4](https://doi.org/10.1007/978-3-658-45209-4_4).

Zeichen können beliebige Symbole (beispielsweise Buchstaben), Bilder (beispielsweise Hieroglyphen), Töne (beispielsweise Morsezeichen) oder gar Objekte (beispielsweise Rauchzeichen) sein. Zeichen derselben Art sind Elemente eines Zeichenvorrats. So sind beispielsweise die Zeichen I, V, X, L, C, D und M Elemente des Zeichenvorrats der römischen Zahlen.

Eine arabische Ziffer ist ein Zeichen, das die Bedeutung einer Zahl hat. Diese Zeichen werden als Dezimalziffern 0, 1, ... 9 verwendet. Jedoch können durchaus auch Buchstaben die Rolle von Ziffern einnehmen, wie das Beispiel der römischen Zahlen zeigt.

Von einem „Alphabet“ spricht man, wenn der Zeichenvorrat eine strenge Ordnung aufweist.



So stellen beispielsweise folgende geordnete Folgen unterschiedliche Alphabete dar:

0, 1	das Binäralphabet,
a, b, c, ... , z	die Kleinbuchstaben ohne Umlaute und ohne ß,
0, 1, ... , 9	das Dezimalalphabet

#### 4.1.1 Rechnerinterne Darstellung von Zeichen

Ein Computer funktioniert mit Strom und besitzt als Basiseinheit nur zwei Zustände: Strom aus oder Strom an. Diese zwei Zustände werden in einem sogenannten „**Bit**“ gespeichert, der kleinsten Speichereinheit eines Computers. Bit ist eine Abkürzung für den englischen Ausdruck „binary digit“.

Rechnerintern werden Zeichen durch Bits dargestellt. Ein Bit kann den Wert 0 oder 1 annehmen.



Mit einem Bit können also gerade mal zwei verschiedene Fälle dargestellt werden. Gruppiert man jedoch mehrer Bits zu einer Einheit, so entstehen schnell mehr Kombinationsmöglichkeiten. Mit einer Gruppe von 2 Bits sind  $2 * 2 = 4$  Kombinationen möglich, mit einer Gruppe von 3 Bits können bereits  $2 * 2 * 2 = 8$  verschiedene Fälle dargestellt werden. Die Anzahl an Möglichkeiten wächst exponentiell:

1 Bit	$2^1 = 2 = \mathbf{2}$ Möglichkeiten
2 Bits	$2^2 = 2 * 2 = \mathbf{4}$ Möglichkeiten
3 Bits	$2^3 = 2 * 2 * 2 = \mathbf{8}$ Möglichkeiten
4 Bits	$2^4 = 2 * 2 * 2 * 2 = \mathbf{16}$ Möglichkeiten
5 Bits	$2^5 = 2 * 2 * 2 * 2 * 2 = \mathbf{32}$ Möglichkeiten
6 Bits	$2^6 = 2 * 2 * 2 * 2 * 2 * 2 = \mathbf{64}$ Möglichkeiten
7 Bits	$2^7 = 2 * 2 * 2 * 2 * 2 * 2 * 2 = \mathbf{128}$ Möglichkeiten
8 Bits	$2^8 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = \mathbf{256}$ Möglichkeiten

...

Folgende Kombinationen sind mit 3 Bits möglich:

000   001   010   011   100   101   110   111

Um aus diesen binären Zahlen ein Alphabet zu erstellen, ordnet ein Computer jeder dieser Bitgruppen ein Zeichen zu, das heißt, dass jede dieser Bitkombinationen ein Zeichen eines Alphabets repräsentieren kann. Im folgenden Beispiel wird jeder dieser Bitkombinationen eine Farbe zugeordnet:

000	Schwarz
001	Rot
010	Grün
011	Gelb
100	Blau
101	Magenta
110	Cyan
111	Weiß

Bei der rechnerinternen Darstellung von Zeichen durch Bits braucht man nur eine eindeutig umkehrbare Zuordnung (beispielsweise erzeugt durch eine Tabelle) und kann dann jedem Zeichen eine Bitkombination und jeder Bitkombination ein Zeichen zuordnen.



Mit anderen Worten, man bildet die Elemente eines Zeichenvorrats auf die Elemente eines anderen Zeichenvorrats ab.

Die Abbildung der Elemente eines Zeichenvorrats auf die Elemente eines anderen Zeichenvorrats bezeichnet man als „Codierung“.



Ein Beispiel einer häufig verwendeten Codierung ist dasjenige der positiven ganzen Zahlen. Folgende Tabelle weist jeder 8-Bit Kombination eine positive Dezimalzahl zu:

00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
...	
11111101	253
11111110	254
11111111	255

Ein Computer speichert somit Zahlen intern nicht mit Dezimalziffern, sondern binär mit der eben gezeigten Codierung. Für negative Zahlen gibt es wiederum spezielle Codierungen, genauso wie auch für Gleitpunkt-Zahlen. Eine ausführliche Behandlung dieser Codierungen wird in Anhang [→ E.5](#) stattfinden.


Wichtig ist eine solche Codierung schlussendlich für die Person, welche ein Programm schreiben möchte. Während zu Anfangszeiten des digitalen Computerzeitalters tatsächlich einzelne Bits oder Bitfolgen (beispielsweise mittels Lochkarten) dem Computer eingespeist werden mussten, kann heutzutage ganz einfach eine Text-Datei geschrieben werden. Der Compiler wandelt sodann sämtliche Zeichen und Zahlen mittels der entsprechenden Codierung in Bitfolgen um.

### 4.1.2 Relevante Text-Codierungen für Rechner



Damit einem Computer mitgeteilt werden kann, was ein Programm ausführen soll, muss der Programmtext in einer **Codierung** geschrieben werden, welche der Computer versteht. Heutzutage sind insbesondere folgende Codierungen für die Sprache C relevant:

- **ISO 646:** Die Codierung ISO/IEC 646 wird als „Invariant Code Set“ bezeichnet und definiert 110 global festgelegte Zeichen sowie 18 länderspezifische Zeichen. Dieser Zeichensatz beinhaltet die lateinischen Buchstaben, die arabischen Ziffern, einige Interpunktionszeichen (Prozentzeichen, Punkt, Fragezeichen etc.) sowie einige Steuerzeichen. Dieser Zeichensatz war ursprünglich die minimale Anforderung, die ein C-Compiler verstehen muss. Heutige Compiler erwarten jedoch mindestens den ASCII-Zeichensatz.
- **ASCII:** Der ASCII-Zeichensatz definiert 128 Zeichen als 7-Bit-Code, wobei diese heutzutage in 8 Bit verpackt werden und das achte Bit auf 0 gesetzt wird. ASCII ist die Abkürzung für „American Standard Code for Information Interchange“. Der ASCII-Zeichensatz ist identisch mit der US-länderspezifischen Variante von ISO 646. ISO 646 ist zudem vorwärtskompatibel zu ASCII.
- **UTF-8:** Mit der Verbreitung von **Unicode** (auch bekannt als **UCS** = universal character set) wurde UTF-8 (UTC transformation format for 8 bit) immer populärer. Ab dem Standard C11 sind in C auch Unicode-Zeichen möglich. ASCII ist vorwärtskompatibel zu UTF-8, doch UTF-8 erlaubt es zusätzlich, sämtliche Zeichen des Unicode (mit weit über einer Million Zeichen) abzubilden.
- **UTF-16:** UTF-16 ist ähnlich wie UTF-8 eine Codierung, welche Unicode abzubilden vermag. Leider aber benötigt ein Zeichen 16 und nicht 8 Bits. Deswegen ist ASCII nicht vorwärtskompatibel zu UTF-16, sondern erfordert eine Umkonvertierung.
- **UTF-32:** Diese Codierung ist genauso wie UTF-16 zu betrachten, jedoch benötigt ein Zeichen 32 Bits.

Die verschiedenen Codierungen, insbesondere die Umwandlungen von Unicode mittels UTF, werden in Anhang  detailliert beschrieben.

Im Rahmen dieses Buches kann vorerst davon ausgegangen werden, dass Zeichen in C normalerweise 8 Bits benötigen. Damit kann in der Regel dem ASCII-Standard gefolgt werden. Es sei jedoch angemerkt, dass manche Betriebssysteme intern UTF-16 oder gar UTF-32 als Codierung verwenden. Diese Zeichen müssen jedoch durch einen speziellen Typ und entsprechende Umwandlungsfunktionen angesprochen werden. In Kapitel [→ 6.1](#) wird genauer darauf eingegangen.

**Zeichen** sind zunächst Buchstaben, Ziffern oder Sonderzeichen. Zu diesen Zeichen können auch noch Steuerzeichen hinzukommen.



Ein Steuerzeichen ist beispielsweise `^C`, das durch das gleichzeitige Anschlagen der Taste `Strg` (Steuerung, oder `Ctrl` (Control) auf englischen Tastaturen) und der Taste `C` erzeugt wird. Die Eingabe von `^C` in einer Konsole kann dazu dienen, ein Programm abzubrechen.

## 4.2 Variablen und Werte

Bei imperativen Sprachen wie C besteht ein Programm aus einer Folge von Anweisungen, wie beispielsweise „Wenn `x` größer als `y` ist, dann ziehe `y` von `x` ab und weise das Ergebnis `x` zu“. Wesentlich an diesen Sprachen ist das Variablenkonzept.

Eine **Variable** ist im Gegensatz zu einer Konstanten eine veränderliche Größe. In der Programmierung werden Variablen benötigt, um in ihnen **Werte** abzulegen, die Werte wieder abzurufen, zu verarbeiten und erneut zu speichern.

Eine Variable hat in C vier Eigenschaften:

- Variablennamen
- Datentyp
- Wert
- Adresse



Eine Variable ist eine benannte Speicherstelle (Adresse) des Arbeitsspeichers. Über den Variablennamen kann auf die entsprechende Speicherstelle und somit auch auf den Wert, welcher dort gespeichert ist, zugegriffen werden.



Es ist auch möglich, dass Variablen in Registern des Prozessors liegen. Darauf wird in Kapitel [→ 15.6](#) eingegangen.

An so einer Speicherstelle steht eine Folge von Bits, welche mithilfe einer passenden Codierung einen gewünschten Wert speichern. Um dem Compiler klarzumachen, welche Codierung er für die gespeicherten Werte nutzen soll, wird jede Variable in C mit einem sogenannten „Typ“ deklariert.

### 4.2.1 Datentypen

Ein **Datentyp** – oder kurz **Typ** – ist der Bauplan für eine Variable. Ein Datentyp legt fest, welche **Operationen** auf einer Variablen möglich sind und wie die Codierung der Variablen im Speicher erfolgt.



Mit der Codierung wird festgelegt, wie viele Bytes die Variable im Speicher einnimmt und welche Bedeutung ein jedes Bit dieser Darstellung hat. In der Programmiersprache C gibt es viele vorgegebene Datentypen mit einer genau festgelegten Codierung. Beim Schreiben eines Programmes ist es zudem erlaubt, weitere Typen zu definieren. All diese Typen werden mittels eines Namens angesprochen. Nach einmaliger Definition kann ein solcher Typname in der vorgesehenen Bedeutung ohne weitere Maßnahmen verwendet werden.

Eine Variable wird vereinbart, indem mittels eines Typnamens die gewünschte Codierung festgelegt wird. Dadurch ist für den Compiler klar, wieviele Bytes eine Variable benötigt und wie sie zu interpretieren ist.

Der in einer Variablen gespeicherte Wert muss der Variablen explizit zugewiesen werden. Wenn einer Variablen nie ein Wert zugewiesen wird, sind die Bits an der Speicherstelle der Variablen mehr oder weniger zufällig angeordnet und ergeben sinnfreie Werte. Dies kann zu einer Fehlfunktion des Programms führen. Daher darf nicht versäumt werden, den Variablen die gewünschten Startwerte (Initialwerte) zuzuweisen, das heißt die Variablen zu **initialisieren**.

Beim Schreiben eines Programmes ist man selbst verantwortlich dafür, dass Variablen initialisiert werden.



Die Sprache C definiert mehrere Datentypen, welche unterteilt werden können in einfache und zusammengesetzte Datentypen.

Ein **einfacher Datentyp** ist nicht aus noch einfacheren Datentypen zusammengesetzt. Die Sprache C stellt standardmäßig einige einfachen Datentypen bereit wie `int` zur Darstellung von Integern oder `double` zur Darstellung von Gleitpunktzahlen. Beide Datentypen wurden bereits in Programmierbeispielen in Kapitel [→ 2](#) verwendet. Eine ausführliche Beschreibung dieser Typen findet sich in Kapitel [→ 7](#).

Ein Datentyp wie `int` oder `double` ist in C jedoch nicht nur definiert durch seine Codierung, sondern auch durch die zulässigen Operationen, welche mit diesem Datentyp ausgeführt werden können.

In der Programmierung bezeichnet eine Operation die Verknüpfung von **Operanden** mit einem **Operator**. Beispielsweise bezeichnet die Verknüpfung der Operanden 1 und 2 mit dem Operator `+` eine Operation, welche die beiden Zahlen addiert.

Man kann sich vorstellen, dass die Sprache C eine sehr große Tabelle angelegt hat, welche für alle möglichen Kombinationen aus Datentyp und Operation festlegt, was genau ausgeführt werden soll:

Operator	Operanden	Operation	Ergebnis
-	<code>int</code>	Negation	<code>int</code>
*	<code>(int, int)</code>	Multiplikation	<code>int</code>
<	<code>(float, float)</code>	Vergleich	<code>int</code> (Wahrheitswert)
=	<code>(double, int)</code>	Zuweisung	<code>double</code>

Diese Liste ist natürlich nicht vollständig, soll jedoch veranschaulichen, dass die Sprache C genau festlegt, wie Operatoren mit den zugrundeliegenden Werten verfahren sollen.

Beispielsweise ist der Multiplikations-Operator `*` folgendermaßen zu lesen: Der Multiplikations-Operator `*` verknüpft zwei `int`-Werte zu einem `int`-Wert als Ergebnis, indem er sie miteinander multipliziert.



Der Compiler nutzt diese Tabelle, um dann die genau zu dem Typ passenden Maschinenbefehle zu generieren.

**Zusammengesetzte Datentypen** erlauben es, Aufbau und Struktur von Variablen angepasst an eine gegebene Problemstellung zu definieren, indem man aus mehreren einfachen Datentypen einen neuen Datentyp konstruiert.

C bietet für zusammengesetzte Datentypen das Sprachkonstrukt der **Struktur** (struct).



Eine Struktur bildet ein Objekt der realen Welt in ein Schema ab, das der Compiler versteht, wobei ein Objekt beispielsweise ein Haus, ein Vertrag oder eine Firma sein kann – also prinzipiell jeder Gegenstand, der für einen Menschen eine Bedeutung hat und den er sprachlich beschreiben kann. Will man beispielsweise eine Software für das Personalwesen einer Firma schreiben, so ist es beispielsweise zweckmäßig, einen Datentyp mittels einer Struktur namens Mitarbeiter einzuführen.

Eine Struktur, die einen Mitarbeiter abbildet, könnte in C wie folgt deklariert werden:

```
struct Mitarbeiter {  
    char vorname[25];  
    char name[25];  
    int alter;  
    int gehalt;  
};
```

Auch Bibliotheken können solche Datentypen definieren, beispielsweise der Typ struct tm in der Bibliothek <time.h> (siehe Kapitel [→ 13.1.10](#)).

Auf zusammengesetzte Typen wird in Kapitel [→ 13.1](#) noch detailliert eingegangen.

Zusammengesetzte Typen sind sogenannten „selbst definierte“ Typen – sie werden nicht von der Sprache vorgegeben. Sie erlauben es, nützliche Typen bereitzustellen, welche genau auf das vorliegende Programm zugeschnitten sind.

Nebst den Strukturen zählen zudem die Enumerationen und Unions zu den selbst definierten Datentypen. Auf Enumerationen wird in Kapitel [→ 7.2.5](#) eingegangen und auf Unionen in Kapitel [→ 13.3](#).

## 4.3 Funktionen in prozeduralen Programmen

Ein Programm soll in kleinere Einheiten aufgeteilt werden können, die überschaubar sind, was ganz allgemein als **Modularisierung** bezeichnet wird. Riesengroße Programme, die sich über mehrere tausend Codezeilen erstrecken, sollen vermieden werden, da sie unübersichtlich und damit schwer wartbar sind.

Prozedurale Programme bestehen deswegen aus einem **Hauptprogramm** und mehreren **Unterprogrammen**.

Ein Programm soll verständlich, testbar und wartbar sein. Deswegen wird es in kleinere Unterprogramme unterteilt. Ein Unterprogramm umfasst eine Folge von Anweisungen und stellt eine Programmeinheit dar.



Diese Unterprogramme werden **Prozeduren** und **Funktionen** genannt. Jedes dieser Unterprogramme hat die Aufgabe, eine Teillösung in einen abgeschlossenen Rahmen zu verpacken.

Dadurch, dass ein Unterprogramm eine bestimmte Berechnung kapselt, kann eine Berechnung relativ leicht ausgetauscht und ein alternativer Algorithmus verwendet werden, ohne dass der Rest des Programms von dieser Änderung betroffen ist.



Prozeduren und Funktionen sind ein Mittel zur Strukturierung eines Programms.



Prozeduren werden traditionell eher als ausführende Programmteile betrachtet, wohingegen Funktionen eher den Charakter eines Unterprogramms haben, welches ein Resultat liefert. Die Programmiersprache C macht hierbei keine Unterscheidung und definiert jedes Unterprogramm als eine Funktion. Ein Programm besteht also in C stets aus Funktionen, wovon eine Funktion die Rolle des Hauptprogrammes übernimmt und alle anderen Funktionen Unterprogramme darstellen.

Das Hauptprogramm in C ist die Funktion mit dem Namen **main()**. Mit ihr beginnt ein Programm seine Ausführung.



Der Name `main()` ist exklusiv für das Hauptprogramm reserviert und darf in einem C-Programm nicht ein zweites Mal verwendet werden.



Auch andere Funktionen haben einen Namen. Somit können Funktionen per Name und mit wechselnden aktuellen Parametern aufgerufen werden. Dies kann den Programmtext erheblich verkürzen, indem an den entsprechenden Stellen anstelle des gesamten Codes der Funktion einfach nur noch der Aufruf der Funktion notiert werden muss.

Funktionen haben die Aufgabe, Teile eines Programms unter einem eigenen Namen zusammenzufassen.



Es ist somit möglich, die Funktionalität einer Funktion an beliebig vielen Stellen im Programm zu nutzen.

Funktionen sind wiederverwendbar. Kann ein und dasselbe Unterprogramm mehrfach in einem Programm aufgerufen werden, so wird das Programm insgesamt kürzer und es ist auch einfacher zu testen.



Funktionen können auch Ergebnisse zurückliefern. Die Programmausführung wird nach Abarbeitung einer Funktion an der Stelle des Aufrufs fortgesetzt.

### 4.3.1 Unterprogramme und Bibliotheken

Viele Unterprogramme sind nicht nur in einem einzigen Programm verwendbar, sondern lösen eine Aufgabe, die in vielen Programmen zu erledigen ist.

Unterprogramme aus zusammenhängenden Aufgabengebieten werden in sogenannten **Bibliotheken** (auf englisch „**libraries**“) zusammengefasst und anderen Codeteilen mittels Einbinden der passenden Datei zur Verfügung gestellt.



Bibliotheken werden vom Compilerhersteller zusammen mit dem Compiler ausgeliefert, können aber auch auf dem freien Markt bezogen oder selbst erstellt werden.

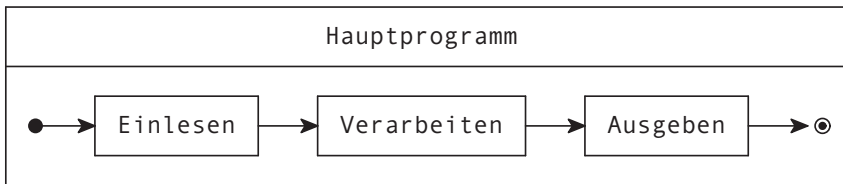
So wird beispielsweise in der Programmiersprache C das Programmieren durch eine ganze Reihe von Standardbibliotheken erleichtert, die jeder C-Compiler anbieten muss. Im „Hello World“-Programm in Kapitel [→ 1.1](#) wurde beispielsweise die Bibliothek `<stdio.h>` eingebunden, welche die Funktionalität der Standardein- und -ausgabe beinhaltet.

Die standardisierten Bibliotheken und deren verfügbare Funktionen sind im Anhang [→ A](#) aufgelistet.

### 4.3.2 Aufrufhierarchie von Unterprogrammen

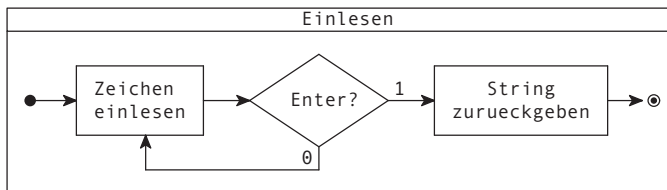
Genauso wie das Hauptprogramm kann jedes Unterprogramm wiederum Unterprogramme aufrufen. Durch diese Aufrufe entsteht eine Beziehung zwischen den verschiedenen Programmeinheiten, die als „**Aufrufhierarchie**“ bezeichnet wird.

Im Folgenden wird ein einfaches Beispiel für ein Programm betrachtet, welches aus einem Hauptprogramm und drei Unterprogrammen besteht. Welche Unterprogramme wann vom Hauptprogramm aufgerufen werden, kann wie in einem Flussdiagramm dargestellt werden:



Im Diagramm wird offensichtlich: Das gezeigte Hauptprogramm ruft die Unterprogramme Einlesen, Verarbeiten und Ausgeben hintereinander auf.

Was genau hinter einem dieser Unterprogramme steckt, ist jedoch in diesem ersten Diagramm noch nicht ersichtlich. Für ein Unterprogramm wird ein neues Diagramm erstellt. Beispielsweise könnte das erste Unterprogramm Einlesen folgendermaßen aussehen:



Hier zeigt sich, wie praktisch Flussdiagramme sind: Jedes Diagramm wird eindeutig mit seinem Anfang (ausgefüllter Punkt) und Ende (umrahmter Punkt) dargestellt, so dass jedes Sinnbild nach außen hin genau einen Eingang und einen Ausgang hat und somit als abgeschlossene Einheit betrachtet werden kann. Von außen betrachtet nennt sich dies eine sogenannte „Black Box“.

Innerhalb des Diagramms kann der **Programmfluss** erneut beliebig zerlegt werden. Jede Einheit (beispielsweise „Zeichen einlesen“) kann erneut ein vollständiges Flussdiagramm enthalten.

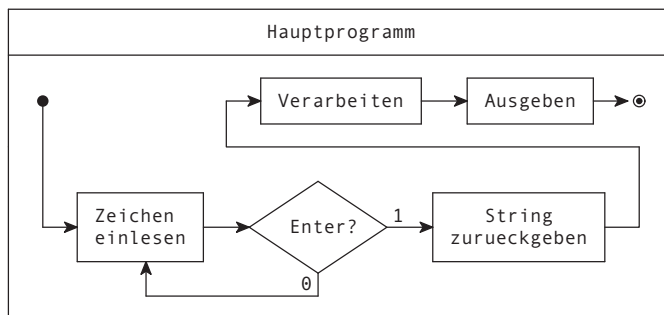
Ein Programm wird nicht durch ein einzelnes Diagramm beschrieben, sondern durch eine Menge von Diagrammen – für jedes Unterprogramm ein Diagramm.



Jedes Programm besitzt einen Zeiger (der sogenannte **Befehlszeiger**, oder auf Englisch „**instruction pointer**“), welcher dem Computer sagt, wo innerhalb des Codes gerade Anweisungen ausgeführt werden. Im Diagramm kann dies einfach nachvollzogen werden, indem ein Zeiger auf die verschiedenen Kästchen zeigt. Betrachtet man nun nur das Hauptprogramm, dann startet der Zeiger zunächst beim Startpunkt, dann zeigt er auf Einlesen, dann auf Verarbeiten, dann auf Ausgeben und dann geht der Zeiger ans Ende.

Gelangt ein Programm jedoch an ein Unterprogramm, so verschiebt sich der Befehlszeiger temporär an die Startstelle des Unterprogrammes. In der Programmiersprache C werden Unterprogramme mittels Funktionen realisiert. Bei einem sogenannten „Funktionsaufruf“ verschiebt sich der Befehlszeiger an den Startpunkt der Funktion. Man sagt auch, es wird in die Funktion „gesprungen“. Der Befehlszeiger arbeitet sodann die Anweisungen gemäß dem Diagramm des Unterprogramms ab und gelangt irgendwann zum Ende der Funktion. Dann wird zu dem Punkt im Hauptprogramm „zurückgesprungen“, an welchem das Unterprogramm aufgerufen wurde, und der Zeiger wird auf die nächste Anweisung dahinter gesetzt.

Würde man diesen Fluss in einem Diagramm darstellen, so sähe dies etwa so aus:



Dieses Flussdiagramm hat exakt die gleiche Funktionalität. Nur ist hier das Einlesen nicht mehr in eine Funktion verpackt. Es stellt sich nun natürlich die Frage, ab wann denn nun einzelne Anweisungen überhaupt in Funktionen verpackt werden sollen.

## 4.4 Schrittweise Verfeinerung beim Top-Down-Design

Wann führt man ein Unterprogramm ein und wann genügt es, die Verarbeitungsschritte direkt hinzuschreiben? Diese sehr wichtige Frage kann nicht einfach pauschal beantwortet werden. Viele Aspekte spielen hier eine Rolle, die zum Teil auch schon erwähnt wurden, wie etwa die Übersichtlichkeit oder die Wiederverwendbarkeit. In den folgenden Abschnitten wird eine Vorgehensweise gezeigt, welche die Aufteilung eines Programms in einzelne Unterprogrammeinheiten unterstützt.

Beim Entwurf eines neuen Programms geht man in der Regel **top-down** vor. Das bedeutet, dass man von groben Strukturen (top) ausgeht, die dann schrittweise in feinere Strukturen (bottom) zerlegt werden. Dies ist das Prinzip der **schrittweisen Verfeinerung**.

Bei der schrittweisen Verfeinerung wird das vorgegebene Problem in Teilprobleme und in die Beziehungen zwischen diesen Teilproblemen top-down so zerlegt, dass jede Teilaufgabe weitgehend unabhängig von den anderen Teilaufgaben gelöst werden kann.



Ist die Lösung eines Teilproblems nicht komplex, werden die notwendigen Schritte einfach hingeschrieben. Ist ein Teilproblem komplex, wird das Prinzip der schrittweisen Verfeinerung wiederum auf dieses Teilproblem angewandt. Man kann für jeden Teil entscheiden, ob es sich lohnt, diesen Teil als Unterprogramm zu realisieren.

Im Kontext einer prozeduralen Programmiersprache wie C werden zur Lösung von Teilproblemen Unterprogramme eingeführt.



#### 4.4.1 Das EVA-Prinzip

Eine der wichtigsten Methoden, um Programmeinheiten (also Hauptprogramm und Unterprogramme) in weitere Programmeinheiten aufzuteilen, wurde bereits angesprochen:

Ein Datenverarbeitungsproblem kann sehr oft in die Teilprobleme **E**inlesen, **V**erarbeiten und **A**usgeben aufgespalten werden.



Dieses Prinzip der Zerlegung wird nach den Anfangsbuchstaben der Tätigkeiten auch als **EVA-Prinzip** bezeichnet. Daten müssen zuerst eingelesen werden. Dann werden diese Daten verarbeitet und am Ende müssen die Ergebnisse ausgegeben werden.



Nicht immer ist das EVA-Prinzip sinnvoll, jedoch ist es ein gutes Grundprinzip, wie das folgende Beispiel zeigt.

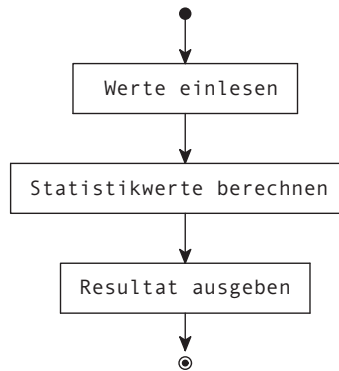
#### 4.4.2 Exemplarische Durchführung eines Top-Down-Design

In diesem Kapitel wird ein Programm betrachtet, welches die Aufgabe hat, Durchschnittswerte zu berechnen. Das Programm wird top-down in seine Teilaufgaben aufgespalten:

- Ein Programm liest Zahlen ein. Sobald eine Null gelesen wird, wird die Eingabe abgebrochen.
- Das Programm errechnet daraufhin den mathematischen Durchschnitt und die Standardabweichung der eingegebenen Werte.
- Das Programm gibt die errechneten Werte aus.

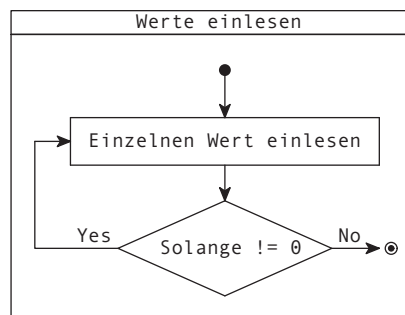
Alleine schon aus der Problemstellung lässt sich das EVA-Prinzip klar erkennen: Einlesen – Verarbeiten – Ausgeben. Das Diagramm sieht in diesem konkreten Beispiel somit folgendermaßen aus:



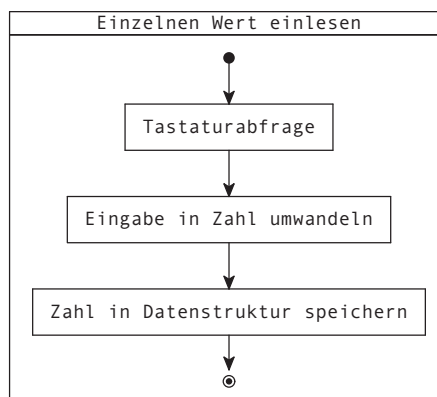


Bereits zu Beginn wird nun festgestellt, dass das Ausgeben der Resultate wohl keine Herausforderung darstellt, sodass die Ausgabe nicht weiter als Unterprogramm ausgeführt, sondern direkt als Anweisung geschrieben werden kann.

Für das Einlesen jedoch wird ein Unterprogramm top-down entworfen:



Die Anweisung „Einzelnen Wert einlesen“ wird nun wiederum mittels des EVA-Prinzips verfeinert:



Dieses Diagramm ist nun genügend fein ausgearbeitet, es werden somit keine weiteren Unterprogramme mehr definiert. Das Diagramm ist bereits auf der Beschreibungsebene von Anweisungen in der Programmiersprache angelangt. Mit anderen Worten: Die Verarbeitungsschritte entsprechen bereits einzelnen Anweisungen.

Es ist möglich, Flussdiagramme bis auf die Programmcode-Ebene zu verfeinern.

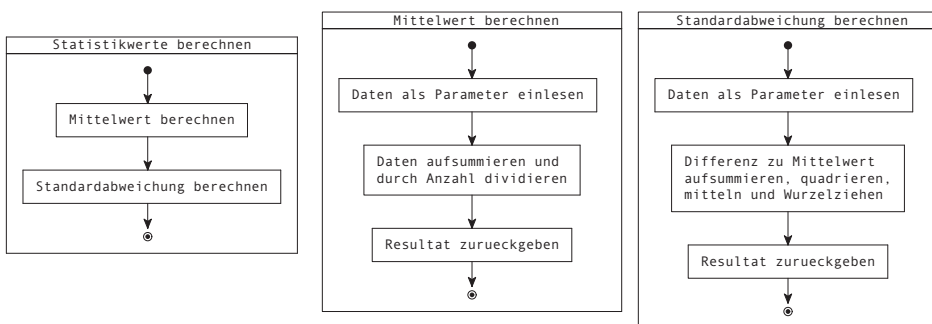


Dann entspricht jeder Verarbeitungsschritt einer Anweisung des Programms. Bei komplexen Programmen kommt man aber erst nach mehrfachen Verfeinerungen auf die Ebene einzelner Anweisungen.

Generell ist es aber nicht wünschenswert, den Entwurf bis auf die Ebene einzelner Anweisungen voranzutreiben, da dann identische Informationen in zweierlei Notation (Flussdiagramm, Programmcode) vorliegen würden. Änderungen an einer einzelnen Anweisung würden dann jedes Mal Änderungen in der Spezifikation nach sich ziehen.



Das zweiten Unterprogramm „Verarbeitung“ des Hauptprogrammes wird nun ebenfalls verfeinert. Hier werden nun gleich alle Verfeinerungsstufen gezeigt:



Für die Eingabe wurde hier die Parameterübergabe verwendet. Für die Ausgabe wurde die Rückgabe des Resultates eingesetzt. Nach diesem Prinzip können sämtliche Funktionen in C aufgebaut werden.

## 4.5 Funktionen in C

Im Programm „Hello world“ bestand das C-Programm nur aus einer einzigen Funktion, der `main()`-Funktion. Dies ist für kleine Programme auch ausreichend. Um ein Programm aber besser strukturieren zu können, erlaubt C die Definition eigener **Funktionen**.

Im folgenden Beispiel wird die Summe der ganzen Zahlen von 1 bis  $n$  gebildet – in mathematischer Schreibweise lautet die Formel dafür:

$$\text{summe} = \sum_{i=1}^n i$$

### 4.5.1 Definition von Funktionen

Bei der Definition einer Funktion unterscheidet man zwei Teile: den Funktionskopf und den Funktionsrumpf.

Der **Kopf einer Funktion** sieht generell folgendermaßen aus:

```
Rueckgabetyf Funktionsname (Parameterliste)
```

Der Funktionskopf enthält die Schnittstelle einer Funktion, wie sie sich nach außen, also gegenüber ihrem Aufrufer, verhält.

Wenn ein Programm aus mehreren Funktionen besteht, dann können diese Funktionen beim Aufruf Daten über Parameter sowie das Funktionsergebnis austauschen.



Der Funktionskopf enthält also den Namen der Funktion, die Liste der **Übergabeparameter** und den **Rückgabetyf**. Hier als Beispiel:

```
int summe(int n)
```

Die Funktion `summe()` erhält als Parameter die Obergrenze, bis zu der die Summe gebildet werden soll. Dieser Parameter  $n$  wird bei Aufruf des Unterprogramms vom Hauptprogramm passend gesetzt. In umgekehrter Richtung erwartet das Hauptprogramm nach Beendigung der Funktion ein Funktionsergebnis mit dem Typ `int`.

Dies stellt den Kopf der Funktion `summe()` dar. Der **Funktionsrumpf** ist der Teil mit den geschweiften Klammern. Er enthält die lokalen Definitionen der Variablen und die Anweisungen der Funktion, beispielsweise:

```
int summe(int n) {  
    int sum = 0;  
    for (int i = 1; i <= n; i = i + 1)  
        sum = sum + i;  
    return sum;  
}
```

Im Rumpf dieser Funktion wird zuerst die lokale Variable `sum` definiert. Dann folgen die Anweisungen, hier zuerst eine `for`-Schleife, in welcher in der Variablen `sum` die Summe berechnet wird. Wenn die Schleife beendet ist, enthält sie das gewünschte Ergebnis, das mittels der `return`-Anweisung an den Aufrufer zurückgegeben wird.

Vergleicht man den Aufbau der Funktion `main()` aus den vorherigen Kapiteln mit diesem allgemeinen Aufbau einer Funktion, so sieht man, dass die Funktion `main()` ganz genau diesem allgemeinen Aufbau einer Funktion folgt. Sie ist also auch einfach eine Funktion.

### 4.5.2 Aufruf von Funktionen

Eine Funktion kann ganz einfach nach folgendem Schema aufgerufen werden:

```
Funktionsname(Argumente)
```

So kann beispielsweise geschrieben werden:

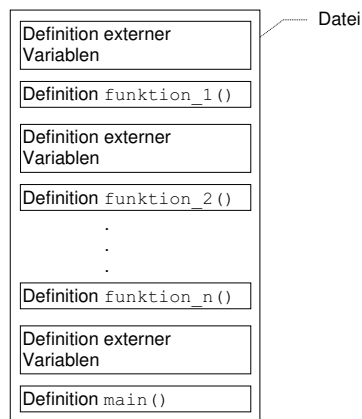
```
resultat = summe(10);
```

Dabei ist `10` das **Argument**, sprich der aktuelle Parameter der Funktion `summe()`. Das bedeutet, dass der formale Parameter `n` der Funktion `summe()` mit dem Wert `10` belegt wird und dass dann die Anweisungen des Funktionsrumpfes durchgeführt werden. Am Ende des **Funktionsaufrufes** kehrt das Programm an die Aufrufstelle zurück und das in der `return`-Anweisung zurückgegebene Funktionsergebnis kann wie jeder andere berechnete Wert verwendet werden – in diesem Falle wird das Funktionsergebnis in der Variablen `resultat` gespeichert.

## 4.6 Struktur einer Quelldatei in C

Bislang wurden nur kleine Programme betrachtet. Mit der Zeit wächst ein Programm jedoch in Funktionalität und Umfang, weswegen der Code dann auf mehrere Dateien verteilt wird. Dies erhöht die Übersichtlichkeit und erlaubt eine arbeitsteilige Entwicklung.

Eine Datei wird als „**Quelldatei**“ bezeichnet, auf Englisch „**source file**“. Eine einzelne Quelldatei in C besteht dabei hauptsächlich aus sogenannten „externen Variablen“ und Funktionsdefinitionen.



**Externe Variablen** werden auch „**globale Variablen**“ genannt. Extern bedeutet, dass sie außerhalb von Funktionen, also extern angelegt sind. Wird eine Variable außerhalb einer Funktion vereinbart, ist diese Variable innerhalb derselben Datei global sichtbar für all diejenigen Funktionen, die nach dieser Variablen definiert werden. Der Begriff „globale Variablen“ hat sich eingebürgert.

Da globale Variablen nach ihrer Definition überall sichtbar sind, können sie zur Kommunikation zwischen Funktionen eingesetzt werden. Nach allgemeinem Bewusstsein ist der Einsatz von globalen Variablen jedoch heutzutage verpönt, da sie die Prinzipien der geordneten Verkapselung von Daten durchbrechen. Sie bringen im Fehlerfall sehr viel Mühe mit sich, um die verursachende Funktion zu ermitteln, und daher ist die Kommunikation über Parameter und Funktionsergebnis vorzuziehen.



Globale (externe) Variablen werden ausführlich in Kapitel [→ 7.4.3](#) behandelt. Bis auf weiteres werden sie nicht weiter betrachtet.

Auch Funktionen sind extern, sprich, sie sind immer extern zu anderen Funktionen. Gewisse Compiler oder Erweiterungen erlauben zwar, Funktionen innerhalb von Funktionen zu definieren, was dann als eine sogenannte „nested function“ bezeichnet wird, der Standard definiert jedoch keine verschachtelten Funktionen.

Funktionsdefinitionen sind also stets extern. Eine Quelldatei besitzt üblicherweise mehrere davon.

Eine Funktion darf sich nicht über mehrere Dateien erstrecken, sondern muss komplett in einer einzigen Datei enthalten sein.



#### 4.6.1 Prototypen und Reihenfolge der Namen

Da ein Compiler eine Quelldatei stets von oben nach unten durcharbeitet, ist die Reihenfolge der externen Variablen und Funktionen nicht unerheblich!

Alle Namen, die an einer Stelle im Programm benutzt werden, müssen zuvor dem Compiler bekanntgegeben worden sein, sonst kann der Compiler die entsprechenden Prüfungen nicht durchführen. Der Grund dafür ist, dass die Sprache C nur ein einziges Mal durch den zu compilierenden Quellcode durchläuft. Dies nennt sich „Single Pass Compiler“.



Bei einfachen Programmen ist dies nicht so ein Problem. Die Funktionsdefinitionen werden einfach in der Reihenfolge definiert, wie sie gebraucht werden.

Sehr schnell jedoch wird ein Programm komplexer, sodass die Einhaltung der Reihenfolge mühsam oder gar unmöglich wird. Wenn sich zwei Funktionen gegenseitig aufrufen, dann muss eine davon zuerst definiert werden. Das führt zu einem Problem.

Der Compiler muss die Schnittstelle einer Funktion kennen, damit er prüfen kann, ob sie korrekt aufgerufen wird.



In einem solchen Falle können sogenannte „**Funktions-Prototypen**“ eingesetzt werden. Genauer erläutert werden diese in Kapitel [→ 11.6](#), hier wird kurz das Prinzip erläutert:

Ein Funktions-Prototyp besteht aus dem Funktionskopf gefolgt von einem Strichpunkt. Mit einem solchen Prototyp wird dem Compiler die Aufrufschnittstelle einer Funktion bekannt gemacht, ohne sie explizit auszuprogrammieren.



Beispielsweise lautet der Prototyp der Funktion `summe()`:

```
int summe(int n);
```

Wenn diese Zeile irgendwo vor einem Aufruf steht, dann nimmt der Compiler an, dass diese Funktion irgendwo definiert werden wird und kennt gleichzeitig die Signatur derselben, sodass ein Aufruf der Funktion übersetzt werden kann.

Wenn eine Funktion keine Parameter erwartet, sollte die Parameterliste beim Prototyp mit dem Schlüsselwort `void` gekennzeichnet werden. Dies ist ein Überbleibsel aus alten Standards. Ab dem C23-Standard ist dieses zusätzliche `void` nicht mehr nötig. Der Compiler nahm früher an, dass bei leerer Parameterliste die Anzahl und der Typ der Parameter erst bei der tatsächlichen Definition der Funktion festgelegt wird.



### 4.6.2 Weitere Bestandteile eines Programms

Zu den Funktionen und externen Variablen eines Programms können noch Präprozessor-Anweisungen, globale Typ-Definitionen und Prototypen hinzukommen.



Prototypen werden üblicherweise im obersten Teil einer Datei hingeschrieben, sodass sie für sämtliche darauffolgenden Definitionen sichtbar sind. Da in einem umfangreicheren Programm die Anzahl an Prototypen sehr schnell anwächst, werden sie in sogenannten „Header-Dateien“ gesammelt und mittels der Präprozessor-Direktive `#include` eingebunden.

**Header-Dateien** bieten Schnittstellen für Code aus anderen Dateien an. Sie werden Header-Dateien genannt, da deren Inhalt normalerweise im Kopf einer Quelldatei, sprich vor den eigentlichen Definitionen, eingebunden wird. Header-Dateien können vom Standard gegeben (wie beispielsweise `<stdio.h>`), von Drittanbietern geliefert (wie beispielsweise bei der Grafikbibliothek OpenGL) oder selbst ausprogrammiert sein. Durch das Einbinden einer Header-Datei werden alle dort beschriebenen Prototypen, Typ-Deklarationen und mehr in das eigene Programm eingebunden.

Zu den Präprozessor-Anweisungen zählt auch die in Kapitel [→ 2.2.2](#) eingeführte `#define`-Anweisung, mit deren Hilfe symbolische Konstanten definiert werden können.

Das Typkonzept von C erlaubt die Vereinbarung eigener Typen wie etwa von Aufzählungs-Typen (`enum`) oder von Struktur-Typen (`struct`). Diese werden meist in mehreren Funktionen benutzt und müssen dann außerhalb dieser Funktionen – also extern – definiert werden. Oftmals werden auch sie in Header-Dateien verpackt.

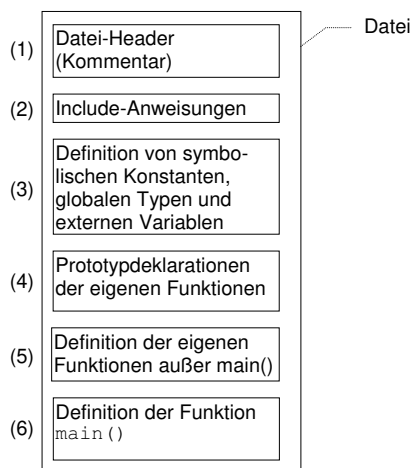


### 4.6.3 Grober Aufbau einer Quelldatei

Im Laufe der Jahre hat sich folgendes, generelle Schema für eine **Quelldatei** in C rauskristallisiert:

- Oft beginnt eine Quelldatei mit einem aussagekräftigen **Kommentar**, um deutlich zu machen, was genau innerhalb der Datei ausprogrammiert wird. Auch werden häufig Anmerkungen zur rechtlichen Benutzung (Copyright) gegeben. Mittlerweile werden solche rechtlichen Angaben jedoch auch häufig erst am Ende der Datei hingeschrieben.
- Dann folgen die `#include`-Anweisungen, die vom Präprozessor bearbeitet werden. Es werden also gleich zu Beginn sämtliche benötigten Bibliotheken und anderweitige Deklarationen eingebunden.
- Es folgen symbolische Konstanten mit `#define`, globale Typvereinbarungen und (gegebenenfalls) globale Variablen sowie vereinzelte Prototypen von Funktionen. All das wird vor den eigenen Funktionen geschrieben.
- Nun folgen alle eigenen Funktionen. Durch die vorher bereits bekannt gemachten Prototypen können sich diese beliebig gegenseitig aufrufen.
- Die Funktion `main()` wird traditionell ans Ende der Datei geschrieben. Dadurch erspart man sich das Schreiben von Prototypen zu Beginn einer neuen Implementation.

Zusammenfassend ist eine C-Quelldatei dann wie im folgenden Bild aufgebaut:



#### 4.6.4 Beispielprogramm für den Aufbau einer Quelldatei

Folgt man dieser vorgeschlagenen Struktur für das Beispiel der Summenberechnung, so sieht die Datei wie folgt aus:

summe.c

```
// Summenberechnung. Berechnet die Summe aller Zahlen
// von 1 bis MAX.

#include <stdio.h>

#define MAX 100

int addition(int a, int b);

int summe(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i = i + 1)
        sum = addition(sum, i);
    return sum;
}

int addition(int a, int b) {
    return a + b;
}

int main(void) {
    int resultat = summe(MAX);
    printf("Die Summe von 1 bis %d ist %d\n", MAX, resultat);
    return 0;
}
```

Hier die Ausgabe des Programms:

```
Die Summe von 1 bis 100 ist 5050
```

Hier wurde zur Veranschaulichung des Prototyps eine Funktion `addition()` hinzugefügt.