

# 11 Blöcke und Funktionen



Wer ein Programm in C schreibt, verpackt die auszuführenden Anweisungen in sogenannten „Funktionen“. Diese Funktionen besitzen einen Namen und können an beliebigen Stellen aufgerufen werden.

Eine **Funktion** ist eine Folge von Anweisungen mit einem Namen, die mittels eines Funktionsaufrufes ausgeführt werden.



Um dem Compiler klarzumachen, welche Anweisungen zu einer Funktion gehören, werden die Anweisungen in einen sogenannten „Block“ geschrieben. Eine Funktion besteht somit stets aus einem **Funktionskopf** (der sogenannten „**Signatur**“) und einem **Funktionsrumpf** (auch „Funktionskörper“ genannt, auf Englisch „body“).

```
int meineFunktion()      // Funktionskopf
{
    Anweisungen          // Funktionsrumpf
}
```

Ein **Block** stellt eine beliebige Folge von Anweisungen dar. Diese Folge von Anweisungen wird sequenziell im Programmcode ausgeführt.



Blöcke treten jedoch auch an anderen Stellen auf, insbesondere können sie ineinander verschachtelt werden. Diese Blöcke bezeichnen dann nicht Funktionskörper, sondern stehen einfach so für sich alleine ohne Namen und bilden eine abgrenzende Code-Umhüllung.

Innerhalb von Blöcken können Variablen definiert und Anweisungen ausgeführt werden. In den folgenden Unterkapiteln wird auf die verschiedenen Eigenschaften von Blöcken und Funktionen eingegangen.

**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\\_11](https://doi.org/10.1007/978-3-658-45209-4_11).

## 11.1 Struktur und Schachtelung von Blöcken

Die Anweisungen eines Blockes werden durch geschweifte Klammern als Blockbegrenzer zusammengefasst. Nach der schließenden geschweiften Klammer kommt kein Strichpunkt. Statt Block ist auch die Bezeichnung „zusammengesetzte Anweisung“ (auf Englisch „**compound statement**“) üblich.



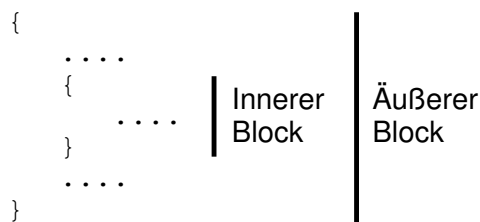
Beispielsweise:

```
{
  Anweisung1;
  Anweisung2;
}
```

Da eine Anweisung eines Blocks selbst wieder ein Block sein kann, können Blöcke geschachtelt werden.



Das folgende Bild symbolisiert die Schachtelung von Blöcken:



Blöcke treten im Programmcode an den folgenden Stellen auf:

- Der Rumpf einer Funktion ist ein Block.
- Da ein Block syntaktisch als eine einzige Anweisung gilt, kann er im Programmtext überall da stehen, wo von der Syntax her nur eine einzige Anweisung zugelassen ist, wie beispielsweise im if- oder else-Zweig einer if-Struktur. Darüber wurde bereits in Kapitel [10.2](#) geschrieben.

- Ein Block kann auch einfach nur zum logischen Gliedern von Anweisungsfolgen dienen. Dies wird als „Sequenz“ bezeichnet. Diese Kontrollstruktur wurde bereits in Kapitel [→ 10.1](#) vorgestellt.
- Ein Block definiert einen Rahmen für die Gültigkeit, Sichtbarkeit und Lebensdauer von Bezeichnern. Er kann an bestimmten Stellen wie beispielsweise bei case-Marken dazu dienen, Variablendefinitionen zu verkapseln. Siehe dazu Kapitel [→ 10.3.4](#).

### 11.1.1 Vereinbarungen und Anweisungen innerhalb von Blöcken

Vereinbarungen umfassen Deklarationen und Definitionen von Variablen und Funktionen. Sie beschreiben die Bereitstellung von Variablen und Funktionen mittels eines Namens und unterscheiden sich somit grundsätzlich von tatsächlich ausführbaren Anweisungen. Siehe auch Kapitel [→ 7.4](#).

Bis zum Standard C90 hatten Blöcke den folgenden Aufbau:

```
{  
    Vereinbarung  
    Anweisungen  
}
```

Somit müssen in C90 die Vereinbarungen immer vor den Anweisungen stehen, sprich zuerst müssen alle Definitionen der Variablen aufgeführt werden, welche danach in Anweisungen benutzt werden. Seit dem C99-Standard existiert diese Einschränkung nicht mehr.

Nach C99 und C11 dürfen Vereinbarungen und Anweisungen gemischt werden. Eine Variable darf aber dennoch erst benutzt werden, nachdem sie vereinbart wurde.



Hierzu wurde von Stroustrup aus C++ das Konzept des „declaration statement“ (der Vereinbarungsanweisung) übernommen. Nach diesem Konzept werden Variablen erst angelegt, wenn man sie benötigt. Ein Vorteil dieses Konzepts ist, dass Variablen mit einer kürzeren Lebensdauer prinzipiell die Zahl der Fehler reduzieren. Es wird vermieden, dass Variablen am Anfang einer Funktion (eines Blockes) definiert werden müssen und anschließend immer wieder – auch für verschiedene Zwecke – gebraucht werden.

## 11.2 Lebensdauer, Gültigkeit und Sichtbarkeit von Variablen

In C können in jedem Block – auch in verschachtelten Blöcken – Vereinbarungen durchgeführt werden.



Generell können Definitionen und Deklarationen von Variablen an zwei Stellen vorgenommen werden:

- Außerhalb von Funktionen (extern)
- Innerhalb von Funktionen (intern) und Blöcken



Variablen, welche außerhalb von Funktionen definiert werden, heißen **externe Variablen**. Sie werden nach allgemeinem Bewusstsein auch „**globale Variablen**“ genannt. Variablen, die innerhalb einer Funktion definiert werden, heißen **interne Variablen** und werden auch als „**lokale Variablen**“ bezeichnet. Siehe dazu auch Kapitel [→ 7.4.2](#) und [→ 15.1.2](#).

Je nach Platzierung definiert eine Variable dabei eine unterschiedliche Lebensdauer, Gültigkeit und Sichtbarkeit:

Die **Lebensdauer** einer Variablen ist die Zeitspanne, in der das Laufzeitsystem des Compilers für die Variable einen Platz im Speicher reserviert hat. Mit anderen Worten, während ihrer Lebensdauer besitzt eine Variable einen Speicherplatz.



Der Speicherplatz für globale Variablen wird bereits vom Lader (siehe dazu Kapitel [→ 5.3](#)) zur Verfügung gestellt in dem sogenannten „Daten-Segment“ (siehe dazu Kapitel [→ 15.2](#)). Der Lader ermittelt, wieviele Bytes er für die Variablen benötigt und reserviert diesen Platz bis zum Ende des Programmes.

Der Speicherplatz für lokale Variablen wird auf dem Stack (siehe dazu auch Kapitel [→ 15.2](#)) angelegt. Der Speicherplatz von lokalen Variablen ist also verfügbar noch bevor sie innerhalb eines Blockes verwendet werden, sprich, bevor sie gültig werden.

Die **Gültigkeit** einer Variablen bedeutet, dass an einer Programmstelle der Name einer Variablen dem Compiler durch eine Vereinbarung bekannt ist.



Globale Variablen werden vor dem Aufruf der `main()`-Funktion initialisiert und sind ab der Stelle ihrer Definition gültig bis zum Ende des Programmes.

Lokale Variablen sind dem Compiler bekannt ab ihrer Definition, beziehungsweise ihrer Initialisierung innerhalb des zugehörigen Blockes. Ab diesem Punkt ist der Name dem entsprechenden Speicherplatz zugeordnet und bleibt bis zum Ende desselben Blockes verfügbar.

Beim Anlegen von Variablen sollte immer ein minimaler Gültigkeitsbereich angestrebt werden. Damit ist gemeint, dass der innerst-mögliche Block benutzt werden sollte.

Variablen sollten immer möglichst nahe an ihrer ersten Verwendung angelegt werden.



Ist eine Variable nur in einem Block von Nutzen, so sollte sie auch erst in diesem angelegt werden. So kann man sich beim Lesen des Programmtextes sicher sein, dass die Variable danach keine Rolle mehr spielt. Dies erhöht die Wart- und Lesbarkeit des Codes und macht die Programme weniger fehleranfällig.

Die **Sichtbarkeit** einer Variablen bedeutet, dass man von einer Programmstelle aus diese Variable sieht, das heißt, dass man auf sie über ihren Namen zugreifen kann.



Die in einem Block definierten Namen sind grundsätzlich innerhalb dieses Blockes und aller darin verschachtelter Blöcke sichtbar. In dem umfassenden Block sowie in allen Blöcken, welche nicht Teil der Verschachtelung sind, sind sie unsichtbar. Globale Variablen sind überall sichtbar.



Im folgenden Beispiel ist zu sehen, dass in den inneren Blöcken des „wahr“- und „falsch“-Zweiges der if-Anweisung lokale Variablen eingeführt werden:

sichtbar.c

```
#include <stdio.h>

int main(void) {
    int x;

    for (x = 0; x < 10; ++x) {
        if (x < 5) {
            int y = 2;
            printf("x * y hat den Wert %d\n", x * y);
        } else {
            int x = 1234;
            printf("Das innere x hat den Wert %d\n", x);
        }
    }

    printf("x hat den Wert %d\n", x);
    return 0;
}
```

Das Programm erzeugt folgenden Output:

```
x * y hat den Wert 0
x * y hat den Wert 2
x * y hat den Wert 4
x * y hat den Wert 6
x * y hat den Wert 8
Das innere x hat den Wert 1234
Das innere x hat den Wert 1234
Das innere x hat den Wert 1234
Das innere x hat den Wert 1234
Das innere x hat den Wert 1234
x hat den Wert 10
```

In diesem Beispiel sieht man zum einen, wie die Variable `y` lokal definiert wird. Man sieht aber auch, dass eine „neue“ Variable `x` in einem inneren Block definiert werden kann, obschon bereits eine Variable mit demselben Namen außerhalb definiert wird. In diesem Falle wird die äußere Variable von der inneren „verdeckt“. Diese Verdeckung des Namens wird im Englischen auch als „**shadowing**“ bezeichnet.

Eine Variable kann gültig sein und von einer Variablen des-  
selben Namens verdeckt werden und deshalb nicht sichtbar  
sein.



Auf eine verdeckte Variable kann nur über ihre Adresse zugegriffen werden,  
nicht aber über ihren Namen.

Lokale Variablen können sogar nicht nur die Namen von Vari-  
ablen, sondern auch die Namen von Funktionen verdecken.



Dies ist im folgenden Beispiel zu sehen:

quadrat.c

```
#include <stdio.h>

double quadrat(double n) {
    return n * n;
}

int main(void) {
    int quadrat;
    printf("%d", quadrat(5));
    return 0;
}
```

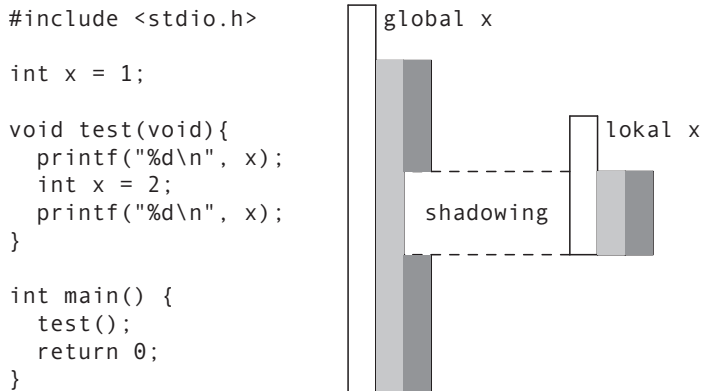
Hier wird durch die lokale Variable `quadrat` die Funktion `quadrat()` verborgen.  
Folglich meldet der Compiler einen Fehler, dass `quadrat()` keine Funktion ist  
und daher ein Funktionsaufruf im Hauptprogramm nicht zulässig ist.

Die folgende Tabelle fasst die Lebensdauer, die Sichtbarkeit und die Gültigkeits-  
bereiche für lokale und globale Variablen zusammen:

	lokal (intern)	global (extern)
Lebensdauer	Block	Programm
Gültigkeitsbereich	ab Vereinbarung	ab Vereinbarung
Sichtbarkeit	Im Block, einschließlich der inneren Blöcke, ab Vereinbarung, solange nicht verdeckt.	Im Programm ab Verein- barung einschließlich der inneren Blöcke, solange nicht verdeckt.

Im folgenden Beispiel kann die Lebensdauer und Sichtbarkeit von Variablen durch die gezeichneten Balken beobachtet werden. Der weiße Balken zeigt die Lebensdauer, der hellgraue Balken die Gültigkeit und der dunkelgraue Balken die Sichtbarkeit.

Die linken Balken zeigen die Eigenschaften der globalen Variablen `x`. Die Variable lebt solange wie das Programm, ist gültig ab der Stelle ihrer Definition und grundsätzlich auch dann sichtbar, wenn sie nicht zwischenzeitlich durch die lokale Variable `x` verdeckt werden würde. Die Balken rechts zeigen die Eigenschaften der lokalen Variablen `x`. Sie lebt nur solange, wie die Funktion `test()` lebt, das heißt, während ihrer Abarbeitung. Gültig und sichtbar ist sie nur innerhalb der Funktion `test()` ab der Stelle ihrer Definition. Während der Gültigkeitsdauer der lokalen Variablen wird die globale Variable verdeckt.



Die Ausgabe dieses Programmes lautet:

```
1
2
```



## 11.3 Definition von Funktionen

Die Definition einer Funktion besteht in C aus dem Funktionskopf und dem Funktionsrumpf. Der Funktionskopf legt die Aufruf-Schnittstelle der Funktion fest. Der Funktionsrumpf enthält lokale Vereinbarungen und die Anweisungen der Funktion.



Die Hauptaufgabe einer Funktion ist es grundsätzlich, aus Eingabedaten Ausgabedaten zu erzeugen. Das bedeutet, dass eine Funktion Daten empfangen und am Ende einen Wert mittels **return** zurückgeben kann.

Während der Abarbeitung einer Funktion können jedoch auch beliebig andere Nebeneffekte auftreten wie beispielsweise:

- Änderungen an Variablen, deren Adressen an die Funktion über die Parameterliste übergeben wurden (siehe Kapitel [→ 11.5.1](#) ).
- Änderungen an Werten globaler Variablen (siehe Kapitel [→ 7.4.3](#) ).
- Systemaufrufe wie beispielsweise die Ausgabe auf dem Bildschirm oder das Öffnen und Schreiben einer Datei. Diese Nebeneffekte werden hier nicht weiter erläutert.

Wie eine Funktion genau definiert wird, wird in den folgenden Unterkapiteln erklärt.

### 11.3.1 Funktionskopf und Funktionsrumpf

Folgendes ist die allgemeine Syntax einer Funktionsdefinition:

```
Rueckgabetyt Funktionsname(Parameterliste) {  
    Anweisungen  
}
```

Eine Funktion hat einen Rückgabetyt, einen Namen und eine (möglicherweise leere) Parameterliste. Dies wird als der „**Funktionskopf**“ bezeichnet. Man spricht auch von der „**Signatur**“ oder der „**Aufrufsstelle**“ einer Funktion.

Innerhalb der geschweiften Klammern stehen die Anweisungen, welche ausgeführt werden sollen. Dies wird als der „**Funktionsrumpf**“ bezeichnet.

Der Funktionsname ist der Name, mit welchem die Funktion an anderer Stelle im Code aufgerufen werden kann. Für Funktionsnamen gelten dieselben Regeln für die Sichtbarkeit wie für Variablennamen: Sie sind innerhalb einer Datei verfügbar ab dem Punkt ihrer Definition, siehe dazu Kapitel [→ 11.2](#).

Es ist möglich, Funktionen eines Programms auf verschiedene Dateien zu verteilen. Eine Funktion muss dabei jedoch stets am Stück in einer einzigen Datei enthalten sein.



Vorerst werden nur Programme betrachtet, die aus einer einzigen Datei bestehen. In Kapitel [→ 22](#) werden Programme, die aus mehreren Dateien bestehen, behandelt.

Der **Rückgabetyt** ist der Typ des Wertes, welcher von der Funktion schlussendlich mittels der return-Anweisung zurückgegeben wird. Siehe dazu auch Kapitel [→ 11.4](#).

Wird der Typ **void** als Rückgabetyt angegeben, so kann zwar mit return die Funktion verlassen werden, die Rückgabe eines Wertes ist dabei aber nicht möglich. Für jeden anderen Typ muss immer ein Wert mit return zurückgegeben werden.



Funktionen mit Rückgabewert `void` werden auch als „**Prozedur**“ bezeichnet.

Diese `void`-Funktionen (Prozeduren also) werden eingesetzt, wenn eine Funktion nicht explizit einen Wert berechnet, sondern beispielsweise lediglich auf dem Bildschirm Ausgaben durchführt, oder aber Daten verändert, welche über Parameter übergeben wurden.

Der Umgang mit dem Rückgabotyp ist je nach Anwendung unterschiedlich. Während in manchen Funktionssammlungen sozusagen alle Funktionen den Rückgabotyp `void` besitzen, definieren andere Funktionssammlungen grundsätzlich zu jeder Funktion einen Rückgabewert, welcher oftmals zum Weiterreichen von Fehlercodes verwendet wird. Die Entscheidung, wie mit dem Rückgabewert umgegangen wird, ist jedem freigestellt.

Wird der Rückgabotyp weggelassen, so wird als Default-Wert vom Compiler der Rückgabotyp `int` angenommen. Dies war insbesondere im C90-Standard gebräuchlich, heutige Compiler geben eine Warnung aus.



Die **Parameterliste** schlussendlich beschreibt die Eingabedaten für eine Funktion. Die Funktion wird an der aufrufenden Stelle mit aktuellen Parametern gefüllt, welche sodann innerhalb der Funktion als formale Parameter verfügbar sind. Im Folgenden wird unterschieden zwischen parameterlosen Funktionen und Funktionen mit Parametern:

### 11.3.2 Parameterlose Funktionen

Hat eine Funktion keine Übergabeparameter, so wird an den Funktionsnamen bei der Definition ein Paar runder Klammern angehängt, beispielsweise:

```
void printHallo() {  
    printf("Hallo");  
}
```

Bei der Definition einer Funktion wird dem Compiler damit mitgeteilt, dass diese Funktion keine Parameter erwartet.

Der Aufruf der Funktion erfolgt durch Anschreiben des Funktionsnamens, gefolgt von einem Paar runder Klammern:

```
printHallo();
```

Da die Funktion keine Parameter erwartet, darf man beim Aufruf auch keine hinschreiben. Ansonsten meldet der Compiler einen Fehler.

Im Gegensatz zur Definition muss gemäß dem Standard bei der Deklaration (dem Prototyp, siehe Kapitel [→ 11.6](#)) einer parameterlosen Funktion mit dem Typ `void` beschrieben werden. Ansonsten nimmt der Compiler an, dass die Funktionsdeklaration unvollständig ist und die Anzahl an zu erwarteten Parametern noch unbestimmt ist. Heutige Compiler warnen vor diesem Versäumnis.



```
void printHallo(void);
```

### 11.3.3 Funktionen mit Parametern

Eine Funktion mit Parametern wird definiert, indem man in die Klammern eine durch Komma getrennte Auflistung von **formalen Parametern** schreibt:

```
Rueckgabe-Typ Funktionsname(Typ1 formalerParameter1,
                             Typ2 formalerParameter2,
                             ...,
                             Typn formalerParameterN)
{
    Anweisungen
}
```

Jede Parameterdefinition besteht aus einem Typ und einem Parameternamen. Innerhalb des Funktionsrumpfes sind diese Parameter als Variablen verfügbar.



Ein formaler Parameter ist eine spezielle lokale Variable. Deshalb darf eine lokale Variable nicht denselben Namen wie ein formaler Parameter tragen.



Als Beispiel für eine Funktion mit Parametern wird die Funktion `printPLZ()` betrachtet, welche eine gegebene Postleitzahl auf dem Bildschirm ausgibt:

```
void printPLZ(int plz) {
    printf("Die Postleitzahl ist: ");
    printf("%05d\n", plz);
}
```

Beim Aufruf der Funktion muss man genau soviele Parameter mit den passenden Typen übergeben, wie die Funktionsdefinition vorschreibt. Der Aufruf für dieses Beispiel kann somit erfolgen mit:

```
printPLZ(meinePLZ);
```

Hier ist `meinePLZ` der **aktuelle Parameter**. Er ist die Postleitzahl, die auf dem Bildschirm ausgegeben werden soll.

Erlaubt ist, dass der Typ eines aktuellen Parameters verschieden ist vom Typ des formalen Parameters, wenn zwischen diesen Typen implizite Typwandlungen möglich sind. Diese impliziten Typumwandlungen finden dann beim Aufruf statt. Da implizite Typwandlungen oft nicht auf Anhieb verständlich sind, ist es immer besser, wenn der Typ des aktuellen mit dem Typ des formalen Parameters übereinstimmt. Das Regelwerk für die implizite Typumwandlung von Parametern (siehe Kapitel [→ 9.10](#)) ist dasselbe wie bei einer Zuweisung, da beim Aufruf tatsächlich eine Zuweisung des Werts des aktuellen Parameters an den formalen Parameter stattfindet.

Ein formaler Parameter wird bei Aufruf der Funktion mit dem Wert des entsprechenden aktuellen Parameters initialisiert. Anders gesagt, der Wert des aktuellen Parameters wird dem formalen Parameter zugewiesen und damit kopiert.



Im Falle des oben aufgeführten **Funktionsaufrufs** `printPLZ(meinePLZ)` wird beim Aufruf der Funktion eine lokale Kopie von `meinePLZ` im formalen Parameter `plz` angelegt. Die Funktion arbeitet nur mit der lokalen Kopie `plz` und kann den aktuellen Parameter `meinePLZ` nicht beeinflussen. Werden Parameter als Kopie ihres Werts übergeben, so nennt man das „call by value“. Mehr dazu wird in Kapitel [→ 11.5](#) beschrieben werden.

Der aktuelle Parameter braucht keine Variable zu sein. Er kann ein beliebiger Ausdruck sein.



```
printPLZ(73730);
```

Anstelle der Begriffe „formaler Parameter“ und „aktueller Parameter“ wird oft auch das Begriffspaar **Parameter** und **Argument** verwendet.



## 11.4 Rücksprung aus einer Funktion – die return-Anweisung

Die **return**-Anweisung beendet einen Funktionsaufruf. Das Programm kehrt zu der Anweisung, in der die Funktion aufgerufen wurde, zurück und beendet diese Anweisung.



Gibt eine Funktion mit `return` einen Wert zurück, so kann dieser Rückgabewert in Ausdrücken weiterverwendet werden. So kann der Rückgabewert in die Berechnung komplexer Ausdrücke einfließen, er kann einer Variablen zugewiesen werden oder an eine andere Funktion übergeben werden. Anschließend wird die nächste Anweisung nach dem Funktionsaufruf abgearbeitet.

Im folgenden Beispiel wird die Funktion `sin()` aufgerufen (Sinus, siehe Anhang → A.2) und das Resultat sofort in einem Ausdruck weiterverwendet:

```
hangAbtrieb = gewicht * sin(alpha);
```

Es ist nicht zwingend notwendig, dass der Rückgabewert einer Funktion abgeholt wird.



So liefert beispielsweise die Funktion `printf()` als Rückgabewert die Anzahl der ausgegebenen Zeichen. Dieser Rückgabewert wird meist nicht abgeholt, wie in folgendem Beispiel:

```
printf("Der Rueckgabewert wird hier nicht abgeholt");
```

Durch das Anhängen eines Strichpunktes wird der Ausdruck hier zu einer Ausdrucksanweisung. Siehe dazu auch Kapitel → 9.1 .

Funktionen, die keinen Rückgabewert haben, können nicht in die Berechnung von Ausdrücken einfließen. Sie können auch nicht auf der rechten Seite einer Zuweisung – eine Zuweisung stellt auch einen Ausdruck dar – verwendet werden. Sie können nur als Ausdrucksanweisung angeschrieben werden.

Eine Funktion, welche keinen Resultatwert liefern soll, wird mit dem Rückgabebetyp `void` definiert. Wann immer aus einer solchen Funktion zurückgekehrt werden soll, kann folgende einfache Anweisung geschrieben werden:

```
return;
```

Enthält ein Funktionsrumpf einer Funktion mit dem Rückgabebetyp `void` keine `return`-Anweisung, so wird die Funktion beim Erreichen der den Funktionsrumpf abschließenden geschweiften Klammer beendet, wobei kein Ergebnis an den Aufrufer zurückgeliefert wird.



Hat eine Funktion einen von `void` verschiedenen Rückgabebetyp, so muss sie mit `return` einen Wert zurückgeben. Nach `return` kann ein beliebiger Ausdruck stehen:

```
return Ausdruck;
```

Mit Hilfe der `return`-Anweisung ist es möglich, den Wert eines Ausdrucks, der in der Funktion berechnet wird (das Funktionsergebnis), an den Aufrufer der Funktion zurückzugeben.



Wenn der Typ von dem Ausdruck nicht mit dem Resultat-Typ der Funktion übereinstimmt, versucht der Compiler, eine implizite Typumwandlung durchzuführen. Das Regelwerk für die implizite Typumwandlung (siehe Kapitel [→ 9.10](#)) ist dasselbe wie bei einer Zuweisung. Ist die implizite Typumwandlung nicht möglich, resultiert eine Fehlermeldung.



## 11.5 Konventionen bei der Parameterübergabe

Funktionen definieren durch ihren Funktionskopf eine Parameterliste. An der aufrufenden Stelle werden passende aktuelle Parameter (Argumente) an die Funktion übergeben.

In C gibt es grundsätzlich nur eine Art, wie Parameter übergeben werden, das sogenannte „call by value“:

Bei einem **call by value** wird der Wert eines aktuellen Parameters an eine Funktion als Kopie übergeben. Dabei kann man den aktuellen Parameter von der aufgerufenen Funktion aus nicht abändern, da die aufgerufene Funktion mit einer Kopie des aktuellen Parameters arbeitet.



Somit hat eine Funktion keine Möglichkeit, Werte außerhalb der Funktion zu verändern. Tatsächlich möchte man jedoch häufig auch Änderungen außerhalb der Funktion bewirken. Hierfür gibt es grundsätzlich drei Möglichkeiten:

- Die Verwendung von globalen Variablen
- call by reference
- call by pointer

Die Möglichkeiten, wie mit globalen Variablen Werte verändert werden können, wurden bereits in Kapitel [→ 7.4.3](#) behandelt.

**call by reference** ist eine Möglichkeit, welche in C nicht existiert, aber beispielsweise in C++. Mit call by reference ist es möglich, über Übergabeparameter nicht nur Werte in eine Funktion hinein, sondern auch aus ihr herauszubringen. Ein solcher Parameter wird dann „Referenzparameter“ genannt.

Ein Aliasname ist einfach ein zweiter Name für dieselbe Variable. Eine Variable kann sowohl über ihren eigentlichen als auch über ihren Aliasnamen angesprochen werden. Somit kann eine Funktion den Aliasnamen in ihrem lokalen Block verwenden und dennoch die tatsächliche Variable außerhalb der Funktion verändern.

Um in C ähnliche Mechanismen zu verwenden, muss auf Pointer zurückgegriffen werden.

### 11.5.1 call by pointer

In C ist eine call by reference-Schnittstelle als Sprachmittel nicht vorgesehen. Man kann das Verhalten dieser Schnittstelle jedoch auch mit der call by value-Schnittstelle erreichen, indem man einen Pointer auf den aktuellen Parameter mit call by value übergibt. Dies wird als **call by pointer** bezeichnet.



Dies zeigt das folgende Beispiel:

ref.c

```
#include <stdio.h>

void init(int* alpha) {
    *alpha = 10;
}

int main(void) {
    int a;
    init(&a);
    printf("Der Wert von a ist %d\n", a);
    return 0;
}
```

Die Ausgabe am Bildschirm ist:

```
Der Wert von a ist 10
```

Also wird der Variablen a der Wert 10 zugewiesen.

Im Detail wird hier die Funktion `init(int* alpha)` aufgerufen durch `init(&a)`. Somit wird die lokale Variable `alpha` beim Aufruf angelegt und mit dem Wert des aktuellen Parameters initialisiert, hier also mit der Adresse von `a`. Man kann sich das als Kopiervorgang vorstellen:

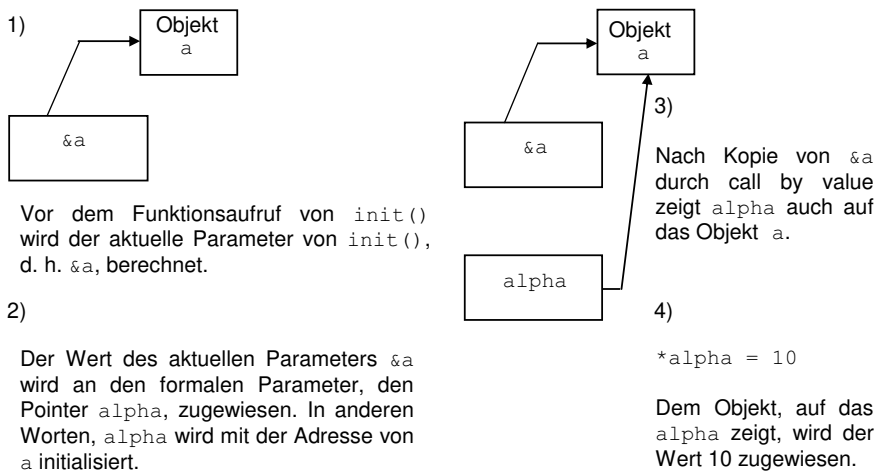
```
int* alpha = &a;
```

Damit steht im formalen Parameter `alpha` die Adresse der Variablen `a`. In der Anweisung innerhalb der Funktion wird sodann dem Objekt, auf das der Pointer `alpha` zeigt (`*alpha` eben), der Wert `10` zugewiesen.

```
*alpha = 10;
```

Da `alpha` schlussendlich jedoch auf dieselbe Speicherzelle zeigt wie `a`, ist somit der eigentliche Speicherinhalt der Variablen `a` verändert worden. Und so wurde innerhalb der Funktion eine Veränderung außerhalb der Funktion erwirkt.

Das folgende Bild symbolisiert die Zwischenschritte bei dem Funktionsaufruf nochmals:



### 11.5.2 const-safe-Programmierung

Wenn Funktionen auf Speicherbereiche außerhalb zugreifen dürfen, können viele Fehler passieren. Entsprechend hat man in der Programmiersprache C eine Schutzmaßnahme eingebaut:

Eine Funktion kann ein per Pointer übergebenes Objekt mittels des `const`-Schlüsselwortes vor Schreibzugriffen schützen. So kann im obigen Beispiel der Funktionskopf auch folgendermaßen definiert werden:

```
void init(const int* alpha)
```

Mit dieser Angabe kann man grundsätzlich davon ausgehen, dass die Variable `alpha` durch den Aufruf dieser Funktion nicht verändert werden wird. Die Anweisung `*alpha = 10;` innerhalb der Funktion wird bei diesem Funktionskopf vom Compiler als Fehler markiert.

Die Markierung eines Pointers mit `const` bewirkt, dass der Compiler den Inhalt des Objekts, auf das der Pointer zeigt, als unveränderbar annimmt. Der Pointer selbst kann verändert werden, der Wert des Objekts hingegen wird vom Compiler vor jeglichem Schreibzugriff geschützt.



Diese Art der Parameterübergabe wird als **const-safe-Programmierung** bezeichnet. Man geht dabei davon aus, dass innerhalb solcher Funktionen keine krummen Dinge mit den Pointern gedreht werden und somit die Daten vor Schreibzugriffen wirksam durch den Compiler geschützt sind.

const-safe-Programmierung ist beim erstmaligen Programmieren mit C oftmals sehr hinderlich, da der Compiler tatsächlich jegliche Schreibzugriffe verweigert. Für fortgeschrittene Programmierung bietet dies jedoch eine wertvolle Hilfe und führt zu einem guten Programmierstil. Es wird im Rahmen dieses Buches somit empfohlen, ein Programm erst allmählich const-safe zu gestalten und sich hierbei bei jedem Parameter zu überlegen: Braucht die Funktion auf das entsprechende Objekt einen Schreibzugriff?



Bei Funktionen, welche Pointer ohne `const` annehmen, muss beim Programmieren immer damit gerechnet werden, dass Nebeneffekte auftreten, welche möglicherweise nicht erwartet sind. Leider gibt es jedoch auch bei durchgängiger const-safe-Programmierung das Problem, dass in C mittels eines expliziten Casts (siehe Kapitel [→ 9.9.5](#)) ein Pointer beliebig in einen nicht-const-Pointer gewandelt werden kann. Deswegen gilt diese Art des Castens in der modernen Programmierung als verpönt.



Der const-Qualifikator wurde bereits in Kapitel [→ 7.5.1](#) besprochen als eine Möglichkeit, einen beliebigen Wert als „schreibgeschützt“ zu definieren.

Die übliche Schreibweise des const-Schlüsselwortes ist dabei auf der linken Seite des Typs:

```
const int;
```

Tatsächlich ist das const-Schlüsselwortes jedoch linksassoziativ, sprich, es wirkt auf das Element links davon. Korrekterweise müsste also Folgendes geschrieben werden:

```
int const;
```

Dass der Typ auch rechts von const stehen kann, ist eine Vereinfachung der Schreibweise (sogenannter „syntactic sugar“), welche durch den Compiler ermöglicht wird.

Bei einfachen Typen spielt dies keine Rolle, bei Pointer-Typen jedoch gibt es nun mehrere Möglichkeiten, das Schlüsselwort zu setzen:

```
const int *;  
int const *;  
int * const;
```

Die ersten beiden Zeilen sind äquivalent, denn const ist jeweils der Qualifikator für den int-Typ. Die dritte Zeile jedoch verändert die Bedeutung:

Während die Angabe `const int *` dasselbe bedeutet wie `int const *`, bedeutet `int * const` hingegen, dass der Pointer als const deklariert ist und nicht der `int`-Typ! Der Compiler verbietet somit zwar ein erneutes Zuweisen des Pointers, erlaubt jedoch, den Pointer zu dereferenzieren und den im referenzierten Objekt gespeicherten Wert zu überschreiben.



Auf weitere Eigenheiten der Platzierung des Schlüsselwortes const bei Pointern wird in Kapitel [→ 12.6](#) eingegangen.

## 11.6 Vorwärtsdeklaration von Funktionen

Im Rahmen der Standardisierung von C durch das ANSI-Komitee wurde festgelegt, dass die Konsistenz zwischen Funktionskopf und Funktionsaufrufen vom Compiler überprüft werden soll. Wenn der Compiler aber prüfen soll, ob eine Funktion richtig aufgerufen wird, dann muss ihm beim Aufruf der Funktion die Schnittstelle der Funktion, also der Funktionskopf, bereits bekannt sein.

In komplexeren Programmen wird man schnell in die Situation geraten, dass Funktionen sich gegenseitig aufrufen und somit keine Anordnung der Funktionen dem Compiler alle Schnittstellen vor den jeweiligen Aufrufen bereitstellen kann. Hierfür gibt es jedoch eine Lösung:

Steht 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 deklariert wird.

Mit der **Vorwärtsdeklaration** (dem sogenannten **Funktions-Prototypen**) wird dem Compiler der Name der Funktion, der Typ ihres Rückgabewerts und der Aufbau ihrer Parameterliste bekannt gemacht. Stimmen die Vorwärtsdeklaration, der Aufruf der Funktion und die Definition der Funktion nicht überein, so resultiert eine Warnung des Compilers oder ein Compilerfehler.



Im folgenden Beispiel wird die Funktion `printQuadrat()` in der Funktion `main()` aufgerufen. Die Definition von `printQuadrat()` erfolgt jedoch im folgenden Programm erst nach dem Aufruf:

prototype.c

```
#include <stdio.h>

void printQuadrat(int x);           // Funktions-Prototyp

int main(void) {
    printQuadrat(5);               // Aufruf von printQuadrat()
    return 0;
}

void printQuadrat(int x) {         // Funktionsdefinition
    printf("%d\n", x * x);
}
```

Ein Funktions-Prototyp entspricht vom Aufbau her einem Funktionskopf. Dabei sind aber die folgenden Abweichungen zur Struktur des Funktionskopfes zugelassen:

- Der Name eines Parameters im Prototyp muss nicht mit dem Namen des entsprechenden formalen Parameters im Funktionskopf übereinstimmen.
- Der Name eines formalen Parameters kann im Prototyp auch weggelassen werden. Entscheidend aber ist, dass der Typ jedes formalen Parameters angegeben wird.
- Im Gegensatz zu einem Funktionskopf wird ein Funktions-Prototyp mit einem Semikolon abgeschlossen.

So könnte im obigen Beispiel `prototype.c` der Funktions-Prototyp auch wie folgt geschrieben werden:

```
void printQuadrat(int);  
void printQuadrat(int zahl);
```

Der Rückgabetyt sowie die Anzahl, die Datentypen und die Reihenfolge der formalen Parameter müssen identisch sein zwischen Prototyp und Funktionskopf.



Wird bei einem Prototyp oder bei einer Funktionsdefinition kein Rückgabetyt angegeben (also auch nicht `void`), dann setzt der Compiler in C90 automatisch den Rückgabetyt `int` für diese Funktion ein. In C99 und C11 ist diese implizite Typenerweiterung nicht gestattet. Viele Compiler tun es aus Kompatibilitätsgründen jedoch trotzdem, geben aber eine entsprechende Warnung aus.



### 11.6.1 Header-Dateien von Bibliotheken

Die Regeln von Funktions-Prototypen gelten auch für Bibliotheksfunktionen. Will man also Bibliotheksfunktionen aufrufen, so müssen ihre Prototypen bekannt sein. Diese befinden sich – neben Makros und Konstanten – in den Header-Dateien.

Durch das Einbinden der **Header-Datei** einer **Bibliothek** werden die Funktions-Prototypen der Bibliotheksfunktionen eingefügt. Dadurch werden die Parameterliste und der Rückgabebetyp jeder Bibliotheksfunktion dem Compiler bekannt gemacht.



So war bisher in den meisten Beispielen die Zeile `#include <stdio.h>` vorhanden. Der Präprozessor (siehe Kapitel [→ 21](#)), der diese Zeile bearbeitet, sorgt dafür, dass an dieser Stelle der Inhalt der Datei `stdio.h` in den Quellcode eingebunden und übersetzt wird, in der unter anderem die Prototypen der Funktionen für die Ein- und Ausgabe stehen. Dadurch kennt dann der Compiler die Prototypen der Bibliotheksfunktionen, während er den anschließenden Programmtext übersetzt. Anhang [→ A](#) stellt die verschiedenen Klassen von Bibliotheksfunktionen des ISO-Standards vor.

Genau genommen sollte man zwischen den Header-Dateien und den entsprechenden Bibliotheken unterscheiden. Im alltäglichen Sprachgebrauch wird jedoch häufig das Einbinden einer Header-Dateien mit der Verlinkung der tatsächlichen Bibliothek gleichgestellt.



### 11.6.2 Eigene Header-Dateien

Bei einem Programm, das in mehrere Quelldateien aufgeteilt ist, muss man zur Übersetzung in jeder Datei, in der man eine Funktion außerhalb der Datei aufruft, einen Prototypen zur Verfügung stellen. Fügt man in jeder Datei einen eigenen Prototypen ein, dann kann leicht ein Wildwuchs entstehen. Der Übersetzer kann dann nicht gänzlich prüfen, ob alle Prototypen zur Funktionsdefinition passen, ob alle Funktions-Prototypen für die gleiche Funktion untereinander verträglich sind, etc. Aus Gründen der Projektorganisation ist es günstig, nur einen einzigen Prototypen für eine Funktion zu haben, der überall da benutzt wird, wo die Funktion aufgerufen wird.

Die folgenden Regeln haben sich als zweckmäßig herausgestellt und ermöglichen größtmögliche Konsistenz über Dateigrenzen hinweg:

1. Jede Funktion, die außerhalb der Datei benutzt werden soll, in der sie definiert ist, erhält einen Prototypen in einer Header-Datei.
2. Jede Quellcodedatei, die einen Aufruf einer Funktion aus einer anderen Datei enthält, inkludiert die entsprechende Header-Datei.
3. Die Quelldatei, die die Definition einer Funktion enthält, soll die Header-Datei ebenfalls inkludieren.

Regel 1 gewährleistet, dass nur ein einziger Funktions-Prototyp für eine Funktion existiert. Regel 2 erlaubt es dem Übersetzer, die aufrufende Funktion zu überprüfen, ob sie verträglich mit dem Prototypen ist. Regel 3 schließlich stellt sicher, dass der Compiler prüfen kann, ob der Prototyp zur Definition passt. Auf die Verwendung eigener Header-Dateien wird in Kapitel [→ 22](#) noch ausführlich eingegangen.

Dadurch, dass Header-Dateien somit von verschiedensten Dateien eingebunden werden, kann es passieren, dass ein und dieselbe Header-Datei innerhalb einer Übersetzungseinheit mehrfach eingebunden wird. Damit dies nicht passiert, behilft man sich in C mit einer bedingten Compilierung, welche in Kapitel [→ 22.3.4](#) nachgelesen werden kann.

## 11.7 Die Ellipse ... – variable Parameteranzahl

Die Programmiersprache C bietet neben den Funktionen mit fester Parameteranzahl auch eine Möglichkeit, Funktionen so zu definieren, dass eine beliebige Anzahl von Parametern übergeben werden kann. Die Kennzeichnung einer solchen Funktion erfolgt mit drei Punkten ... nach dem letzten formalen Parameter in der Parameterliste. Dies wird als „**Ellipse**“ oder „Auslassung“ bezeichnet. Dabei muss die Funktion mindestens einen explizit angegebenen Parameter enthalten.



Beim Aufruf muss die Anzahl der aktuellen Parameter mindestens so groß sein wie die Anzahl der explizit angeschriebenen formalen Parameter. Folgendes Beispiel zeigt die Definition einer Funktion mit **variabler Parameterliste**:

```
int varFunc(int zahl1, double zahl2, ...);
```

Beispiele zum Aufruf der Funktion varFunc() sind:

```
int z1 = 3;
double z2 = 5.4;

varFunc(z1, z2, "String"); // 1 zusätzlicher String
varFunc(z1, z2, 19, 27);  // 2 zusätzliche Integer-Werte
varFunc(z1, z2);          // keine zusätzlichen Parameter
varFunc(z1);              // !! Fehler: nur 1 Parameter !!
```

Da die Parameter des variablen Anteils nicht als feste formale Parameter definiert werden können, kann der Compiler für den variablen Anteil natürlich keine Typüberprüfung der aktuellen Übergabeparameter gegen die formalen Parameter durchführen.



Eine Funktion mit einer variabel langen Parameterliste muss irgendwie Zugriff auf die aktuellen Parameterwerte bekommen und zudem erfahren, wie viele Werte und von welchem Typ an sie übergeben wurden. Bei einer Ellipse fehlen diese Angaben.

C stellt daher ein Hilfsmittel zur Verfügung, nämlich Typen und Makros, die in der Datei `<stdarg.h>` definiert sind und mit denen dieser Zugriff auf die einzelnen Parameter möglich wird. Das folgende Beispiel berechnet den prozentualen Ausschuss einer Menge von Prüflingen mit Hilfe eines Schwellwertes:

ellipse.c

```
#include <stdio.h>
#include <stdarg.h>

const double SCHWELLE = 3.0;
const double ENDE = -1;

double qualitaet(double, ...);

int main(void) {
    printf("Der Ausschuss betraegt %5.2f %%\n",
        100 * qualitaet(SCHWELLE, 2.5, 3.1, 2.9, 3.2, ENDE));
    printf("Der Ausschuss betraegt %5.2f %%\n",
        100 * qualitaet(SCHWELLE, 4.2, 3.8, 3.4, 2.9, 2.7, ENDE));
    return 0;
}

double qualitaet(double schwellwert, ...) {
    int anzahlSchlechterTeile = 0;
    double wert;
    int i = 0;
    va_list listenposition;
    va_start(listenposition, schwellwert);
    wert = va_arg(listenposition, double);

    while (wert != ENDE) {
        if (wert > schwellwert) ++anzahlSchlechterTeile;
        ++i;
        wert = va_arg(listenposition, double);
    }

    va_end(listenposition);
    return (double)anzahlSchlechterTeile / i;
}
```

Die Ausgabe des Programmes ist:

```
Der Ausschuss betraegt 50.00 %
Der Ausschuss betraegt 60.00 %
```

Die Funktion `qualitaet()` in diesem Beispiel erhält als ersten Parameter den gewünschten Schwellwert. Die nächsten Parameter stellen die aktuellen Messwerte der Prüflinge dar, die mit dem Schwellwert zu vergleichen sind. Der Wert `ENDE`, der mit keinem gültigen Messwert übereinstimmen darf, zeigt das Ende der Messreihe an.

Der Zugriff auf die aktuellen Parameter erfolgt folgendermaßen: Als erstes wird an der Stelle (1) eine Variable `listenposition` definiert mit dem Typ `va_list`. Dann wird mit Hilfe von `va_start()` an der Stelle (2) diese Variable so initialisiert, dass sie auf den ersten variablen Parameter zeigt. Dazu wird als zweiten Parameter der Funktion `va_start()` der letzte feste Parameter der Parameterliste übergeben (hier die Variable `schwellwert`). Ab dem C23-Standard wird die Angabe des letzten festen Parameters überflüssig.

Die Variable `listenposition` wird anschließend von `va_arg()` benutzt an den Stellen (3) und (4). Der Aufruf von `va_arg()` liefert als Ergebniswert den Wert des aktuellen Parameters, auf den `listenposition` aktuell zeigt. Der Typ dieses Parameters wird als zweiter Parameter an `va_arg()` übergeben. Mit jedem weiteren Aufruf von `va_arg()` zeigt `listenposition` auf den nächsten Parameter. Der Abschluss ist erreicht, wenn `listenposition` auf `ENDE` zeigt.

Es ist zu beachten, dass es sich bei `va_start()` und `va_arg()` nicht um Funktionen handelt, sondern um Makros mit Parametern (siehe Kapitel [→ 21.2](#)). Dementsprechend ist es `va_start()` möglich, die Variable zu initialisieren und `va_arg()` kann einen Typen als Parameter entgegennehmen. So etwas ist mit Funktionen nicht möglich.

Schlussendlich wird an der Stelle (5) noch `va_end()` aufgerufen. Dies dient zur Freigabe des Speichers, auf den `listenposition` zeigt.

Wie genau Parameter und deren Typinformation mit der Ellipse übergeben werden können, ist nicht in der Sprache definiert. Obiges Beispiel geht davon aus, dass alle Parameter vom selben Typ sind. So kann man als letzten aktuellen Parameter einen Wert übergeben, der als Endekennung dient. Diese Endekennung wird auch „Wächter“, oder auf Englisch „sentinel“ genannt. Auch möglich ist das Übergeben der Anzahl an Parameter als einer der fixen Parameter. Dann wird kein Wächter mehr benötigt.

Ein anderes Beispiel, wie Typen ermittelt werden können, zeigt die Funktion `printf()`. Hier wird als erster (fixer) Parameter ein String übergeben, in welchem codiert ist, welche Typen für die darauffolgenden Parameter erwartet werden. Beispielsweise `%d` für einen `int`-Typ oder `%f` für einen `double`-Typ.

## 11.8 Iteration und Rekursion

Ein Algorithmus heißt **iterativ**, wenn bestimmte Abschnitte des Algorithmus innerhalb einer einzigen Ausführung des Algorithmus mehrfach durchlaufen werden.



Diese Art eines Algorithmus läuft grundsätzlich auf die Benutzung einer Schleife hinaus. Innerhalb einer `while`- oder `for`-Schleife wird ein Wert immer mehr verfeinert, bis er am Ende der Schleife das finale Resultat darstellt. Insofern ist eine Iteration also nichts neues.

Ein Algorithmus heißt **rekursiv**, wenn er Bereiche enthält, die sich selbst wiederum aufrufen.



Dies läuft in der Programmiersprache C grundsätzlich auf die Verwendung einer Funktion hinaus, welche sich selbst im Verlauf ihrer Abarbeitung wiederum aufruft. Wenn eine Funktion sich selbst aufruft, wird dies als „direkte Rekursion“ bezeichnet.

Es gibt aber auch eine „indirekte Rekursion“. Eine indirekte Rekursion liegt beispielsweise vor, wenn zwei oder mehr Funktionen sich wechselseitig, beziehungsweise im Kreis aufrufen. Da indirekte Rekursionen schnell unübersichtlich werden können, wird in der Praxis versucht, eine indirekte Rekursion zu vermeiden beziehungsweise durch eine direkte Rekursion zu ersetzen. In der Praxis tritt die indirekte Rekursion eher als unbeabsichtigter Programmierfehler auf.



Die Rekursion eignet sich, wie man noch sehen wird, zunächst einmal gut zum Umkehren von Reihenfolgen wie beispielsweise von 1, 2, 3, 4 in 4, 3, 2, 1. Zum anderen wird die Rekursion jedoch meist angewandt, um Wachstumsvorgänge (beispielsweise Zellwachstum in der Biologie) einfach zu modellieren oder um Lösungsalgorithmen von „Rätseln“, zum Beispiel dem Finden eines Weges im Labyrinth mit Backtracking, zu programmieren. Auch beim Implementieren von „teile und herrsche“-Algorithmen (auf Englisch „divide and conquer“) leistet die Rekursion wertvolle Dienste. Beispiele für die beiden Ansätze werden beim Suchen und beim Backtracking in Kapitel [→ 20](#) erklärt.

Bei rekursiven Algorithmen, die „nach einer Problemlösung suchen“, ist es bei einer schrittweisen „Suche“ nach solch einer Lösung erforderlich, nicht erfolgreiche Ansätze zu verwerfen und an einer vorherigen Stelle der Lösungssuche erneut fortzufahren. Diese Vorgehensweise führt zum Begriff „Backtracking“.

**Backtracking** ist bei einer baumartigen Lösungssuche die Rückkehr aus einer Sackgasse zu einer vorhergehenden Stelle im Lösungsbaum, von der aus ein erneuter Lösungsversuch gestartet werden soll.



Iteration und Rekursion sind Prinzipien, die oft als Alternativen für die Programmkonstruktion erscheinen. Theoretisch sind Iteration und Rekursion äquivalent, weil man jede Iteration in eine Rekursion umformen kann und umgekehrt. In der Praxis gibt es allerdings oftmals den Fall, dass die iterative oder rekursive Lösung auf der Hand liegt, dass man aber auf die dazu alternative rekursive beziehungsweise iterative Lösung nicht so leicht kommt.

Im Folgenden wird auf die Rekursion und Iteration eingegangen. Es werden zwei Algorithmen erklärt und beispielhaft Code für eine iterative und eine rekursive Lösung gezeigt und erklärt. Um jedoch Rekursion besser zu begreifen, muss zunächst der Begriff des „Stacks“ erläutert werden.

### 11.8.1 Die LIFO-Struktur eines Stacks

Als **Stack** wird ein Speicherbereich bezeichnet, auf dem Informationen temporär abgelegt werden können. Ein Stack wird auch als „Stapel“ bezeichnet. Ganz allgemein ist das Typische an einem Stack, dass auf die Information, die zuletzt abgelegt worden ist, als erstes wieder zugegriffen werden kann. Denken Sie beispielsweise an einen Bücherstapel. Sie beginnen mit dem ersten Buch, legen darauf das zweite, dann das dritte und so fort. In diesem Beispiel soll beim fünften Buch Schluss sein. Beim Abräumen nehmen Sie zuerst das fünfte Buch weg, dann das vierte, dann das dritte und so weiter, bis kein Buch mehr da ist. Bei einem Stack ist es nicht erlaubt, Elemente von unten oder aus der Mitte des Stacks wegzunehmen. Die Datenstruktur eines Stacks wird als LIFO-Datenstruktur bezeichnet. LIFO ist die Abkürzung für „Last-In-First-Out“, sprich das, was als Letztes abgelegt wird, wird wieder als Erstes entnommen. Das Ablegen eines Elementes auf dem Stack wird als `push()`-Operation, das Wegnehmen eines Elementes als `pop()`-Operation bezeichnet.

In der Sprache C werden bei Funktionsaufrufen die übergebenen Parameter, die lokalen Variablen und der Pointer auf die aktuelle Anweisung einer unterbrochenen Funktion (Rücksprungadresse) auf einem Stack abgelegt. Aus diesem Grund wird der Stack auch „**Call-Stack**“ genannt. Dieser wird von der Speicherverwaltung in einem eigenen Bereich, dem **Stack-Segment**, abgelegt, siehe Kapitel [→ 15.2.4](#).

Man muss sich nicht darum kümmern, sondern das Laufzeitsystem des Compilers erledigt diese Arbeit: Wenn eine Funktion aufgerufen wird, werden die zuvor genannten Objekte einfach auf den Stack gelegt. Wenn aus einer Funktion zurückgesprungen wird, werden die auf dem Stack abgelegten Objekte wieder entfernt.

Mehr Informationen über die verschiedenen Speicherbereiche können in Kapitel [→ 15](#) nachgelesen werden.

### 11.8.2 Iterative und rekursive Berechnung der Fakultätsfunktion

Das Prinzip der Iteration und der Rekursion von Funktionen soll an dem folgenden Beispiel der Berechnung der Fakultätsfunktion veranschaulicht werden.

**Iterativ** ist die Fakultätsfunktion definiert durch:

$$n! = 1 \cdot 2 \cdot \dots \cdot n \quad \text{Oder umgekehrt:} \quad n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$$

Dieser Algorithmus lässt sich leicht programmieren:

fakultaetIterativ.c

```
#include <stdio.h>

int main(void) {
    int n = 5;
    printf("Fakultaet(%d): ", n);

    int faku = 1;
    while (n > 1) {
        faku = faku * n;
        --n;
    }

    printf("%d\n", faku);
    return 0;
}
```

Folgendes gibt das Programm aus:

Fakultaet(5): 120

Der Algorithmus arbeitet wie folgt: Zuerst wird die Resultats-Variable faku mit 1 initialisiert. Man beachte: Eine Zahl multipliziert mit 1 ergibt die Zahl selbst. Sodann wird die while-Schleife solange durchlaufen, wie n größer ist als 1. Innerhalb der Schleife wird die Resultats-Variable mit dem aktuellen Wert von n multipliziert und der Wert von n danach um 1 verringert.

Um den Algorithmus **rekursiv** anzugehen, benötigt man eine alternative Beschreibung des Problems, bestehend aus einer Schritt-Anweisung und einer Stopp-Anweisung:

$$n! = n \cdot (n-1)!$$

$$1! = 1$$

Damit gilt:

$$4! = 4 \cdot 3!$$

$$3! = 3 \cdot 2!$$

$$2! = 2 \cdot 1!$$

$$1! = 1$$

Das bedeutet, man schaut jetzt auf eine Funktion  $f(n)$  und versucht, diese Funktion durch sich selbst – aber mit anderen Aufrufparametern – darzustellen. Die mathematische Analyse ist im Beispiel der Fakultätsfunktion ziemlich leicht, denn man sieht sofort:

$$f(n) = n \cdot f(n-1)$$

Damit hat man das fundamentale Rekursionsprinzip für dieses Problem bereits gefunden. Dies ist die Schritt-Anweisung. Damit die Rekursion jedoch nicht ewig geht, braucht es eine Stopp-Anweisung. Dieses sogenannte „Abbruchkriterium“ wurde bereits oben erwähnt. Es heißt:

$$1! = 1$$



Der Algorithmus lässt sich nun leicht programmieren:

fakultaetRekursiv.c

```
#include <stdio.h>

int faku(int n) {
    printf("n: %d\n", n);
    if (n > 1) {
        return n * faku(n - 1);    // (1)
    } else {
        return 1;
    }
}

int main(void) {
    int n = 5;

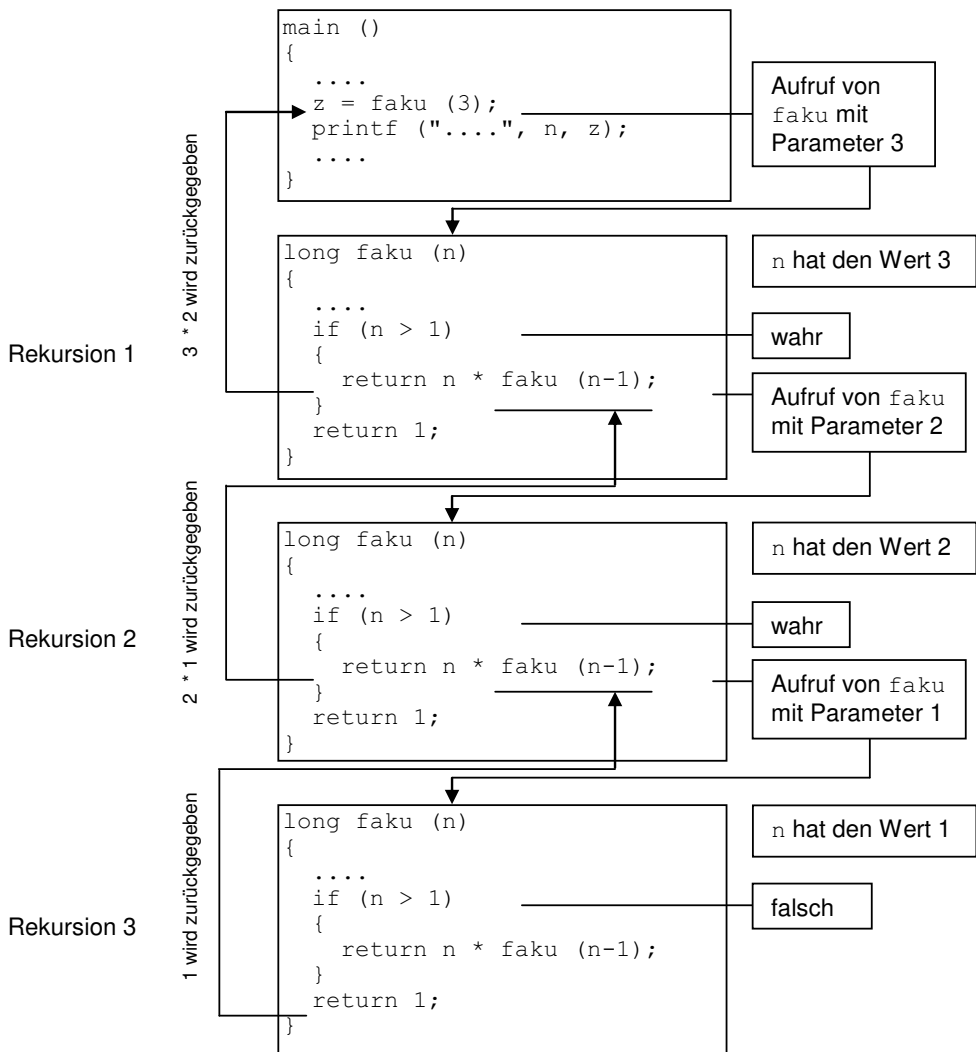
    printf("Fakultaet(%d) = %d\n", n, faku(n));
    return 0;
}
```

Das Programm gibt aus:

```
n: 5
n: 4
n: 3
n: 2
n: 1
Fakultaet(5) = 120
```

Die Funktion `faku()` enthält eine Abfrage, in welcher zwischen der Schritt- und der Stopp-Anweisung unterschieden wird. Ist der gegebene Parameter `n` größer als 1, so wird die Rekursion aufgerufen und das Resultat der Funktion zurückgegeben. Ist sie jedoch gleich 1, so wird einfach nur 1 zurückgegeben.

Die folgende Skizze veranschaulicht am Beispiel der Berechnung von `faku(3)` den rekursiven Aufruf von `faku()` bis zum Erreichen des Abbruchkriteriums und die Beendigung aller wartenden `faku()`-Funktionen nach Erreichen des Abbruchkriteriums:



Wird `fakul()` von der `main()`-Funktion mit dem Wert 3 aufgerufen, so wird auf dem Stack die lokale Variable `n` mit dem Wert 3 angelegt und `fakul(3)` wird abgearbeitet, bis zur Zeile (1). An dieser Stelle wird `fakul()` mit dem neuen Wert `n - 1`, also 2, erneut aufgerufen. `fakul(3)` wird aber noch nicht beendet, sondern wartet auf den Rückgabewert von `fakul(2)`. Um bei dem erneuten Aufruf von `fakul()` den lokalen Parameter `n` nicht zu überschreiben, wird somit eine neue lokale Variable `n` mit dem Wert 2 auf dem Stack abgelegt. Damit das Programm zudem nach Beendigung von `fakul(2)` wieder an die richtige Stelle zurückspringen kann, wird die Rücksprungadresse ebenfalls auf dem Stack abgelegt.

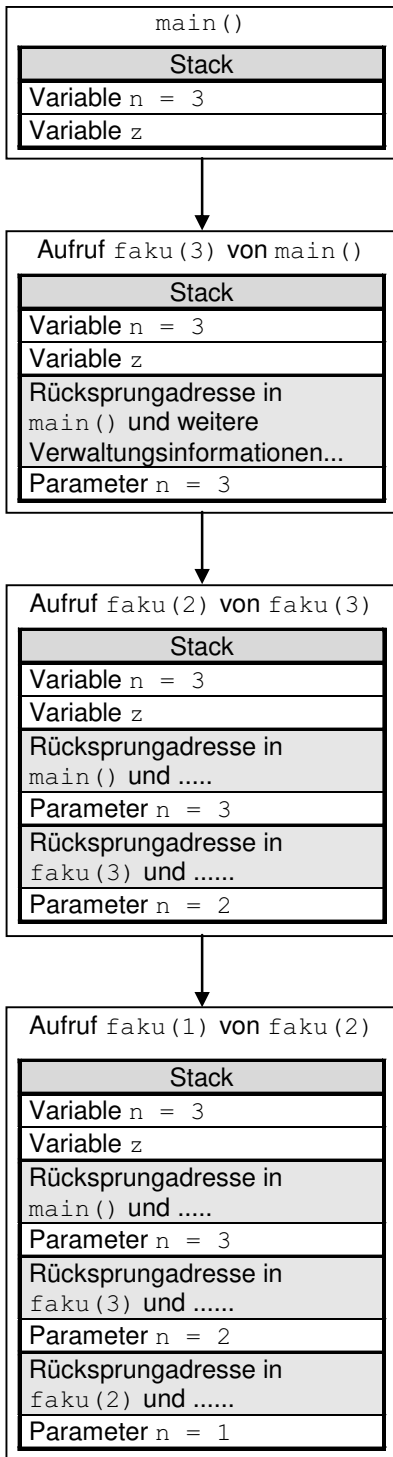
faku(2) wird sodann wieder bis zur Stelle (1) abgearbeitet, an der nun der Funktionsaufruf faku(1) stattfindet. Die lokale Variable  $n$  von faku(1) und die Rücksprungadresse werden erneut auf dem Stack abgelegt und faku(1) aufgerufen. faku(1) beendet nun die Rekursion, da hier die Bedingung falsch ist, und gibt somit direkt den Wert 1 an faku(2) zurück. Bei diesem Rücksprung an die auf dem Stack befindliche Rücksprungadresse wird der Stack abgebaut und somit auch die lokale Variable  $n$  von faku(1) entfernt. Damit bezeichnet  $n$  nun wieder die lokale Variable von faku(2). Nun kann faku(2) die Berechnung von  $n * \text{faku}(1)$  durchführen und somit den Wert  $2 * 1$  an faku(3) zurückgeben. Wieder werden die lokale Variable  $n$  von faku(2) und die Rücksprungadresse abgebaut. faku(3) kann jetzt die Berechnung  $n * \text{faku}(2)$  durchführen und den Wert  $3 * 2$  an  $z$  in der `main()`-Funktion zurückgeben. Damit sind alle von der Funktion faku() auf den Stack abgelegten Daten wieder abgeholt und alle Aufrufe beendet.

Das Ablegen auf dem Stack und das Aufräumen desselben wird auf den folgenden beiden Seiten in Diagrammen verdeutlicht.

Eine zu hohe Zahl von rekursiven Aufrufen führt zum Überlauf des Stacks.



Auch wenn es nicht zum Stacküberlauf kommen sollte, so ist dennoch zu berücksichtigen, dass die Rekursion mehr Speicherplatz und Rechenzeit erfordert als die entsprechende iterative Formulierung. Wenn man den zu einem rekursiven Algorithmus entsprechenden iterativen Algorithmus kennt, so ist dem iterativen Algorithmus üblicherweise der Vorzug zu geben.

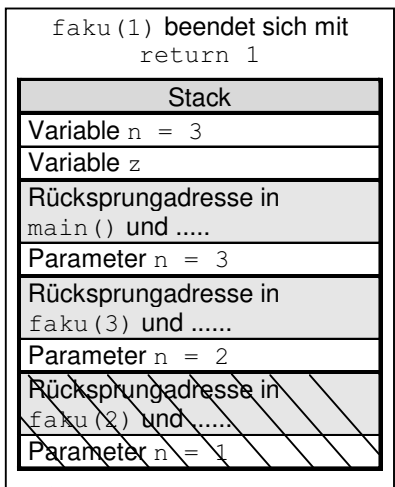


### Aufbau des Stacks für `faku(3)`:

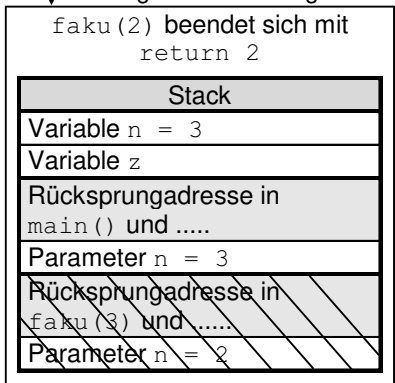
Bei jedem Aufruf von `faku()` werden die Rücksprungadresse und weitere Verwaltungsinformationen auf einem Stack abgelegt, der durch das Laufzeitsystem verwaltet wird. Auch die übergebenen Parameter (hier nur einer) werden auf diesem Stack abgelegt. Dabei wächst der Stack mit der Rekursionstiefe der Funktion.

Der letzte Aufruf von `faku()` mit dem Parameter `n` = 1 bewirkt keine weitere Rekursion, da ja die Abbruchbedingung erfüllt ist.

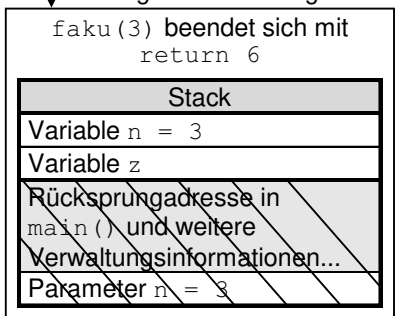
Der Abbau des Stacks geschieht in umgekehrter Reihenfolge, wie aus dem folgenden Bild ersichtlich wird.



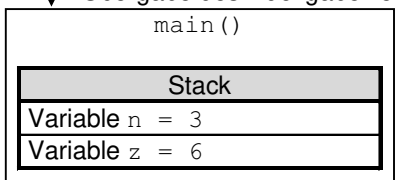
↓ Übergabe des Rückgabewertes über Register



↓ Übergabe des Rückgabewertes über Register



↓ Übergabe des Rückgabewertes über Register



### Abbau des Stacks für faku(3):

Beim Beenden der aufgerufenen Funktion werden auf dem Stack die lokalen Variablen (inklusive der formalen Parameter) freigegeben und die Rücksprungadresse und sonstigen Verwaltungsinformationen abgeholt. Der Rückgabewert wird in diesem Beispiel über ein Register an die aufrufenden Funktionen zurückgegeben. Der Rückgabewert kann auf verschiedene Weise an die aufrufende Funktion zurückgegeben werden, beispielsweise auch über den Stack. Dies ist vom Compiler abhängig.

### 11.8.3 Beispiel für iterative und rekursive Berechnung der Binärdarstellung

Es soll die Binärdarstellung einer Zahl berechnet werden. Bei diesem Beispiel wird ebenfalls deutlich, welche Unterschiede zwischen einer iterativen und einer rekursiven Lösung bestehen und welche Vorteile die Rekursion bieten kann.

binaerIterativ.c

```
#include <stdio.h>

void binaerZahlIter(int zahl1) {
    int array[sizeof(int) * 8];
    int stelle;

    for (int i = 0; i < (sizeof(int) * 8); ++i) {
        array[i] = 0;
    }
    for (stelle = 0; zahl1 != 0; ++stelle) {
        array[stelle] = zahl1 % 2;
        zahl1 /= 2;
    }
    for (; stelle > 0; --stelle) {
        printf("%d ", array[stelle - 1]);
    }
}

int main(void) {
    int zahl = 35;
    binaerZahlIter(zahl);
    return 0;
}
```

Folgendes wird beim Programmablauf ausgegeben:

```
1 0 0 0 1 1
```

Das Programm `binaerit.c` berechnet aus einer gegebenen Dezimalzahl die zugehörige Binärzahl (siehe auch Anhang [→ C.1.2](#)). Dies geschieht durch Iteration. Wird eine Dezimalzahl durch 2 geteilt, so ist der Rest (also 0 oder 1) die letzte Stelle der zugehörigen Binärzahl. Teilt man nun das Ergebnis des Teilvorganges wieder durch 2, so ist der Rest die vorletzte Stelle der Binärzahl usw.

Die folgende Tabelle visualisiert das Umwandeln der Zahl 35 dezimal in eine Binärzahl mit Hilfe des Modulo-Operators:

Rechen- schritt	array						Ergebnis	Rest
	[5]	[4]	[3]	[2]	[1]	[0]		
35 / 2	0	0	0	0	0	1	17	1
17 / 2	0	0	0	0	1	1	8	1
8 / 2	0	0	0	0	1	1	4	0
4 / 2	0	0	0	0	1	1	2	0
2 / 2	0	0	0	0	1	1	1	0
1 / 2	1	0	0	0	1	1	0	1
Dualzahl 1 0 0 0 1 1								

Da nach Ablauf der ersten Schleife die letzte Stelle der Binärzahl an erster Stelle im Array array steht, muss dieses Array mit einer for-Schleife rückwärts ausgegeben werden. Deshalb wird der Schleifenzähler stelle erniedrigt, damit die erste Stelle der Binärzahl als letzte ausgegeben wird.

Da die Umkehr der Reihenfolge mit einer Rekursion einfacher auszuführen ist, jetzt zum Vergleich noch die rekursive Lösung:

binaerRekursiv.c

```
#include <stdio.h>

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

int main(void) {
    int zahl = 35;
    binaerZahlReku(zahl);
    return 0;
}
```

Und hier der Programmablauf:

```
1 0 0 0 1 1
```

Da die Rekursion hier vor der Ausgabe der Binärziffern durchgeführt wird, beginnt das Programm mit der Ausgabe einer Binärziffer erst, wenn die letzte Modulo-Operation durchgeführt wurde. Dies ist aber gerade die erste Stelle der Binärzahl. Dann werden rückwärts alle weiteren Stellen mit `zahl % 2` ausgegeben. Wie man sieht, ist die Realisierung einer „Reihenfolgeumkehr“ rekursiv mit weniger Aufwand zu realisieren als iterativ. Das liegt daran, dass der Programm-Stack als Zwischenspeicher zum Umkehren der Reihenfolge verwendet werden kann. Dies funktioniert im Gegensatz zur iterativen Lösung sogar ohne zu wissen, wie viele Binärziffern die Zahl letztendlich hat.

## 11.9 inline-Funktionen

Inline-Funktionen wurden mit C99 in C eingeführt. Einer Inline-Funktion muss das Schlüsselwort **inline** vorangestellt werden. Hier ein Beispiel:

```
inline double dot3(double* vec1, double* vec2) {  
    return vec1[0] * vec2[0]  
        + vec1[1] * vec2[1]  
        + vec1[2] * vec2[2];  
}
```

Inline-Funktionen definieren ganz normale Funktionen, können jedoch ähnlich wie Makros vom Compiler an der aufrufenden Stelle direkt in den Code hineinkopiert werden.



Inline-Funktionen haben gegenüber Makros den Vorteil, dass sie wie normale Funktionen mit aktuellen Parametern aufgerufen werden können. Der Compiler kann die Parameter wie auch den Rückgabewert prüfen.





Eine Inline-Funktion hat das Ziel, eine solche Funktion so schnell wie möglich zu machen. Es obliegt dem Compiler, ob er das Schlüsselwort `inline` akzeptiert. Ist die Inline-Funktion zu lang, so kann er einen normalen Unterprogrammaufruf daraus machen. In anderen Worten, Performance-Gewinn ist implementierungsabhängig.

Es ist sinnvoll, Inline-Funktionen anstelle von `define`-Makros einzusetzen. Der Programmcode einer Inline-Funktion wird vom Compiler an der Stelle des Aufrufs eingesetzt. Damit ist das Programm so schnell, als ob der Code direkt eingefügt worden wäre. Der aufwendige Aufruf eines Unterprogramms entfällt.



Da der Compiler den Code für das Kopieren und Einsetzen kennen muss, muss die Implementation einer Inline-Funktion mit interner Bindung definiert, sprich während der Compilation als Quellcode verfügbar sein. Siehe dazu auch Kapitel

→ 15.1.2

Es ist jedoch erlaubt, gleichzeitig zur internen `inline`-Funktion eine Definition mit externer Bindung bereitzustellen. Ist dies der Fall, so gilt die Implementation mit `inline` als Alternative zur externen Funktion. Falls die Implementation mit `inline` aus irgendwelchen Gründen nicht geeignet ist, wird der Compiler einen normalen Funktionsaufruf der externen Funktion ausführen. Wann genau welche Funktion aufgerufen wird, ist compilerspezifisch.

Da `inline`-Funktionen während der Compilation als Quellcode verfügbar sein müssen, werden sie häufig in Header-Dateien geschrieben. Da jedoch Header-Dateien oftmals in mehreren Übersetzungseinheiten gleichzeitig eingebunden werden, kann es passieren, dass am Ende in mehreren Übersetzungseinheiten ein und dieselbe `inline`-Funktion mehrfach definiert wird. Der Linker wird somit einen Fehler melden. Damit dies nicht passiert, behilft man sich bei `inline`-Funktionen in Header-Dateien mit dem Trick, die `inline`-Funktion zusätzlich als **`static`** zu vereinbaren (siehe Kapitel → 15.4). Dadurch wird gewährleistet, dass die Implementation der `inline`-Funktion nur innerhalb der aktuellen Übersetzungseinheit sichtbar sein wird. Der Nachteil hierbei ist, dass `inline`-Funktionen, welche aufgrund der Entscheidung des Compilers nicht in den Programmcode eingesetzt, sondern als Funktionsaufruf realisiert werden, in mehreren Objektdateien als exakte Kopie vorliegen. Da `inline`-Funktionen jedoch tendenziell eher klein sind, ist auch der dadurch entstehende Platzverbrauch vernachlässigbar und wird meistens in Kauf genommen.

## 11.10 Funktionen ohne Wiederkehr in C11



Es gibt einige wenige Funktionen, die nie zu ihrem Aufrufer zurückkehren, beispielsweise weil sie das Programm beenden. Dies lässt sich nun dem Compiler mit dem Schlüsselwort `_Noreturn` mitteilen, das mit C11 eingeführt wurde. Mit diesem Wissen ist es dem Compiler möglich, bessere Optimierungen beim Übersetzen durchzuführen.



Analysewerkzeuge können besser beim Debugging helfen, indem sie beispielsweise prüfen können, ob noch Code nach einer Funktion ohne Wiederkehr steht, der niemals erreicht werden kann.

Beispielsweise wird die `exit()`-Funktion (siehe Kapitel [→ 17.2.3](#)) folgendermaßen deklariert:

```
_Noreturn void exit(int status);
```

Alternativ lässt sich dafür auch die neue Bibliothek `<stdnoreturn.h>` verwenden. Die Deklaration sieht dann wie folgt aus:

```
#import <stdnoreturn.h>
noreturn void exit(int status);
```

Welche der beiden Notationen verwendet wird, ist Geschmackssache.

## 11.11 Übungsaufgaben

### Aufgabe 1: Blöcke

Analysieren Sie das folgende Programm und sagen Sie voraus, welchen Wert x an den Stellen mit printf() haben wird. Starten Sie das Programm erst nach Ihrer Analyse.

blocks.c

```
#include <stdio.h>

int x = 5;

void function1(int* u) {
    int x = 4;
    *u = 6;
    printf("f1    - der Wert von x ist %d\n", x);
}

void function2(int x) {
    printf("f2    - der Wert von x ist %d\n", x);
}

int main(void) {
    printf("main - der Wert von x ist %d\n", x);
    function1(&x);
    function2(7);
    printf("main - der Wert von x ist %d\n", x);
    return 0;
}
```

**Aufgabe 2: Funktionen**


Schreiben Sie eine Funktion, welche den Ersatzwiderstand  $R$  einer Parallelschaltung aus zwei Widerständen  $R_1$  und  $R_2$  bestimmt. Die Funktion soll in der `main`-Funktion aufgerufen und dort das Resultat ausgegeben werden. Die Formel lautet:

$$1/R = 1/R_1 + 1/R_2$$

oder

$$R = (R_1 * R_2) / (R_1 + R_2)$$

Schreiben Sie die Funktion mehrmals mit unterschiedlicher Signatur:

- a) Die Funktion erwartet die Parameter  $R_1$  und  $R_2$  und gibt das Resultat als Rückgabewert zurück.
- b) Die Funktion erwartet die Parameter  $R_1$  und  $R_2$  und das Resultat mittels `call-by-pointer`.
- c) Die Funktion erwartet die Parameter  $R_1$  und  $R_2$  und nutzt eine globale Variable für das Resultat.
- d) Fortgeschritten:  Ermitteln Sie im Internet die korrekte Formel für den Ersatzwiderstand einer beliebigen Anzahl an Widerständen und schreiben Sie die Funktion mit einer Parameter-Ellipse, sodass bei Aufruf der Funktion die parallel geschalteten Widerstände einfach aufgelistet werden können.

**Aufgabe 3: Rückgabe mit return und über die Parameterliste**

Schreiben sie drei Funktionen, welche je ein Array von 10 Zahlen als Parameter annehmen und jeweils etwas unterschiedliches berechnen und als Rückgabewert zurückgeben: Die Funktion `summe()` berechnet die Summe der gegebenen Zahlen, die Funktion `maximum()` das Maximum der gegebenen Zahlen und die Funktion `durchschnitt()` den Durchschnitt der gegebenen Zahlen.

Schreiben Sie die Funktion für den Durchschnitt der Zahlen so, dass sie das Resultat der Summe nutzt.

Schreiben sie danach eine weitere Funktion `statistik()`, welche drei Pointer-Parameter `sum`, `max` und `avg` erwartet. Nach Aufruf dieser Funktion sollen alle passenden Werte in den entsprechenden Variablen vorzufinden sein.

Nutzen Sie folgendes Programmgerüst. Ergänzen Sie, wo immer nötig.

return.c

```
#include <stdio.h>

#define MAX 10

int summe(???) { ??? }

int maximum(???) { ??? }

double durchschnitt(???) { ??? }

void statistik(int* sum, int* max, double* avg, ???) { ??? }

int main(void) {
    int a[MAX] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int sum;
    int max;
    double avg;

    statistik(???);

    printf("Summe: %d\n", sum);
    printf("Maximum: %d\n", max);
    printf("Durchschnitt: %f\n", avg);

    return 0;
}
```

### Aufgabe 4: Rekursion

Studieren Sie das folgende Programm:

recursion.c

```
#include <stdio.h>

void unbekannteFunktion(void) {
    int c = getchar();
    if (c != '\n') {
        unbekannteFunktion();
    }
    putchar(c);
}

int main(void) {
    unbekannteFunktion();
    return 0;
}
```

getchar und putchar werden in Kapitel [→ 16.8.1](#) und [→ 16.6.1](#) behandelt.

Welche Ausgabe erwarten Sie von dem Programm, wenn Sie 123 eintippen? Schreiben Sie das Ergebnis auf. Testen Sie anschließend das Programm.

### Aufgabe 5: Potenzen iterativ und rekursiv berechnen

Die Potenz  $a^n$  soll für ein reellwertiges  $a$  und ein ganzzahliges positives  $n$  berechnet werden.

- Schreiben Sie eine Funktion, welche die Potenz mittels der `pow()`-Funktion berechnet. Siehe dazu Anhang [→ A.2](#).
- Berechnen Sie die Potenz mittels einer Schleife.
- Berechnen Sie die Potenz mittels einer Rekursion.

Alle drei Funktionen sollten dasselbe Resultat ergeben. Nehmen Sie die Funktion mit der `pow()`-Funktion als Referenz, sie gibt das korrekte Resultat zurück. Testen Sie Ihr Programm mit folgenden Eingabewerten:

$5^2$   $2.5^8$   $1000^3$   $3^{1000}$   $7^1$   $1^7$   $1^{777777}$   $0^1$   $1^0$   $0^0$

Wenn die Resultate der Funktionen nicht übereinstimmen, überlegen Sie sich wieso. Es könnte sogar sein, dass etwas gänzlich unerwartetes passiert.