

## Die Programmiersprache C

In gewisser Weise ist es ein wenig gewagt, einen Programmierkurs mit der Sprache **C** zu beginnen: Da diese Sprache sehr große Freiheiten bezüglich der Strukturierung von Programmen erlaubt, besteht die Gefahr, sich von Anfang an einen »schlampigen« Programmierstil anzugewöhnen. Andererseits ist C die älteste Programmiersprache, die noch heute von vielen Entwicklern genutzt wird. Außerdem hat die Syntax von C eine Vielzahl neuerer Sprachen stark beeinflusst – die Mehrheit aller in diesem Buch erwähnten Sprachen benutzt die grundlegenden Konstrukte von C.

Die Programmiersprache C wurde ab 1971 von Dennis Ritchie und Brian Kernighan entwickelt, um das Betriebssystem Unix neu zu implementieren. Aus diesem Grund sind Unix und C untrennbar miteinander verbunden; dennoch sind C-Compiler für fast jedes Betriebssystem verfügbar. Seit 1983 wurde eine Neufassung von C als ANSI- und später auch ISO-Standard entwickelt, die nach ihrem endgültigen Veröffentlichungsjahr C90 heißt. 1999 wurde eine weitere Version namens C99 eingeführt, die ein paar weitere Freiheiten erlaubte. 2011 schließlich wurde die Spezifikation der Sprache in ihrer bis heute gültigen Fassung C/11 verabschiedet – die wenigen Unterschiede zu C99 beziehen sich vor allem auf einige Neuerungen, die durch moderne Compiler bereits zuvor zu De-facto-Standards geworden waren.

Wie bereits erwähnt wurde, ist C eine Compilersprache. Ein C-Programm wird also zuerst vollständig in die Maschinensprache des jeweiligen Rechners (mit ein paar Betriebssystem-Bibliotheksaufrufen) übersetzt und dann ausgeführt. Bevor Sie mit dem Programmieren in C beginnen können, müssen Sie sich deshalb einen C-Compiler besorgen. Wenn Sie Linux oder eine andere Unix-Variante einsetzen, ist in der Regel bereits der GNU-C-Compiler GCC auf Ihrem System installiert oder zumindest auf dem Installationsdatenträger oder im Web verfügbar. Bei macOS wird GCC als Teil der Entwicklungsumgebung Xcode mitinstalliert.

Wenn Sie dagegen Windows verwenden, stehen im Internet verschiedene Compiler zum kostenlosen Download bereit. Daneben existieren zahlreiche kommerzielle Angebote, in der Regel im Rahmen komplexer Entwicklungsumgebungen. Um die Beispiele in diesem Abschnitt ohne Änderungen nachvollziehen zu können, sollten Sie sich eine Windows-Version des GCC beschaffen.

Besonders empfehlenswert ist in diesem Zusammenhang der **CygWin-Compiler**, da er auch gleich eine vollständige Unix-Arbeitsumgebung für Windows mitbringt, inklusive *bash* und der wichtigsten Unix-Systemprogramme. Herunterladen können Sie diese Software unter [www.cygwin.com](http://www.cygwin.com). Falls Sie unter Windows einen anderen Compiler einsetzen, funktionieren zwar alle Beispiele in diesem Abschnitt, aber die Compileraufrufe selbst können sich unterscheiden.

### Das erste Beispiel

Am einfachsten erlernen Sie eine Programmiersprache, indem Sie möglichst viele Beispielprogramme ausprobieren, nachvollziehen und anschließend modifizieren. Daher beginnt dieser Abschnitt sofort mit dem ersten Beispiel, das anschließend genau erläutert wird. Öffnen Sie Ihren bevorzugten Texteditor, geben Sie den folgenden Code ein und speichern Sie ihn unter dem Dateinamen **hallo.c**:

```

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

int main() {
    char name[20];
    puts("Hallo Welt!");
    printf("Ihr Name, bitte: ");
    fgets(name, 20, stdin);
    strtok(name, "\n");
    printf("Hallo %s!\n", name);
    return EXIT_SUCCESS;
}

```

Wechseln Sie aus dem Editor in die Konsole, gehen Sie in das Verzeichnis, in dem Sie die Datei **hallo.c** gespeichert haben, und geben Sie Folgendes ein:

```
$ gcc hallo.c
```

Wenn Sie nicht den GCC verwenden, müssen Sie in der Bedienungsanleitung Ihres Compilers nachschlagen, wie der Befehl für die Kompilierung lautet.

Falls Sie das Listing korrekt abgetippt haben, wird der Prompt einfach kommentarlos wieder angezeigt. Andernfalls liefert der Compiler eine oder mehrere Fehlermeldungen, bequemerweise mit Angabe der jeweiligen Zeilennummer. Falls Sie eine Unix-Version verwenden, sollten Sie besser die folgende Variante des Befehls eingeben:

```
$ gcc -o hallo hallo.c
```

Die Option `-o Dateiname` legt einen verbindlichen Dateinamen für das fertig kompilierte Programm fest; ohne diese Angabe trägt das Programm auf Unix-Rechnern je nach konkretem Binärformat einen Namen wie **a.out**. Unter Windows heißt das Resultat automatisch **hallo.exe**.

Unter Unix besteht der nächste Schritt darin, das Programm ausführbar zu machen:

```
$ chmod +x hallo
```

Geben Sie nun unter Unix `./hallo` ein; in Windows genügt die Eingabe `hallo`. Der Grund für diesen Unterschied: unter Windows ist das aktuelle Verzeichnis `.` standardmäßig im Suchpfad enthalten, unter Unix nicht. Das Programm wird ausgeführt und erzeugt folgende Ausgabe:

```

Hallo Welt!
Ihr Name, bitte: Sascha
Hallo Sascha!

```

Es handelt sich bei diesem Programm um eine erweiterte Fassung des klassischen »Hello World«-Beispiels. Es ist Tradition, das Erlernen einer Programmiersprache mit einem Programm zu

beginnen, das diese Begrüßung ausgibt. Unter <http://helloworldcollection.de> finden Sie eine Website mit »Hello World«-Programmen in knapp 600 Programmiersprachen.<sup>1</sup>

Im Folgenden wird das erste Programmierbeispiel Zeile für Zeile erläutert:

- `#include <stdio.h>`  
Diese Zeile ist keine richtige C-Anweisung, sondern eine Präprozessordirektive. Der Präprozessor ist ein Bestandteil des Compilers, der vor der eigentlichen Kompilierung verschiedene organisatorische Aufgaben erledigt. An dieser Stelle lädt er die Header-Datei `stdio.h`, die die Deklarationen der wichtigsten Funktionen für die Ein- und Ausgabe bereitstellt (Standard Input/Output).
- `#include <stdlib.h>`  
Diese zweite `#include`-Direktive importiert die Header-Datei `stdlib.h`. Sie enthält wichtige Funktionen zur Laufzeit- und Speicherkontrolle.
- `#include <strings.h>`  
Ein letztes `#include` bindet die Header-Datei `strings.h` ein. Wie der Name vermuten lässt, stellt sie Funktionen zur Verarbeitung von Strings (Zeichenketten) bereit.
- `int main()`  
In dieser Zeile wird eine Funktion definiert. **Funktionen** sind benannte Codeblöcke, die über ihre Namen aufgerufen werden können. Die spezielle Funktion `main()` übernimmt in einem C-Programm die Aufgabe eines Hauptprogramms: Sie wird beim Start des Programms automatisch vom Betriebssystem aufgerufen.  
  
Der Datentyp beziehungsweise Rückgabewert der Funktion `main()` sollte `int` (ganzzahlig) sein, um dem System einen Wert zurückgeben zu können, der Erfolg oder Fehler anzeigt. Die beiden Klammern hinter dem Funktionsnamen sind Platzhalter für mögliche Parametervariablen. Der Rumpf der Funktion, also die eigentlichen Anweisungen, steht in geschweiften Klammern.
- `char name[20];`  
Diese Zeile deklariert eine Variable mit der Bezeichnung `name`. Eine Variable ist ein benannter Speicherplatz. Wenn Sie den Namen der Variablen in einem Ausdruck (zum Beispiel in einer Berechnung) verwenden, wird automatisch ihr aktueller Wert eingefügt.  
  
Die Variable `name` hat den Datentyp `char[]`. Es handelt sich dabei um einen Verbund einzelner Zeichen, der in C als Ersatz für einen String-Datentyp (eine Zeichenkette) verwendet wird. Der Wert `[20]` in den eckigen Klammern bedeutet, dass die Textlänge maximal 20 Zeichen betragen darf.  
  
Diese Anweisung wird durch ein Semikolon (;) abgeschlossen. In C muss jede Anweisung mit einem Semikolon enden.

---

<sup>1</sup> Noch beeindruckender ist die Website <http://99-bottles-of-beer.net>, die zurzeit 1.500 verschiedene Implementierungen zur Ausgabe des Saufliedes »99 Bottles of Beer« enthält.

- `puts("Hallo Welt!");`  
Die Funktion `puts()` hat die Aufgabe, den angegebenen Text, gefolgt von einem Zeilenumbruch, auszugeben. Text in Anführungszeichen ist ein sogenanntes **Zeichenketten-** oder **String-Literal**, das heißt Text, der »wörtlich gemeint« ist: Er wird unverändert wiedergegeben.
- `printf("Ihr Name, bitte: ");`  
Die Anweisung `printf()` dient der Ausgabe von Text oder einer Formatierung für verschiedene Ausdrücke. Im vorliegenden Fall wird auch wieder nur einfacher Text ausgegeben, allerdings ohne abschließenden Zeilenumbruch, damit die folgende Eingabe in derselben Zeile stattfindet.
- `fgets(name, 20, stdin);`  
Mithilfe von `fgets()` wird maximal die als zweites Argument angegebene Anzahl von Zeichen (hier 20) einer Zeichenkette aus einem Datenstrom gelesen, z. B. aus einer zuvor geöffneten Datei oder wie hier aus der Standardeingabe (`stdin`), und in der als Argument angegebenen Variablen `name` gespeichert. Die Standardeingabe ist für gewöhnlich die Tastatur, es sei denn, Sie leiten die Eingabe um.  
  
In früheren Auflagen wurde an dieser Stelle der kürzere Funktionsaufruf `gets(name)` verwendet, der automatisch aus der Standardeingabe liest. Modernere C-Compiler warnen jedoch zu Recht davor, diese Variante einzusetzen: Da keine maximale Zeichenanzahl angegeben wird, kann `gets()` einen größeren Speicherbereich beanspruchen, als ihm die angegebene Variable bietet, und so andere Elemente überschreiben, was die Lauffähigkeit eines Programms und eventuell sogar die Systemsicherheit gefährdet.
- `strtok(name, "\n");`  
Mit `strtok()` – kurz für *string tokenize* – wird der als erstes Argument angegebene String an Stellen, an denen der im zweiten Argument übergebene String vorkommt, zerteilt. Im vorliegenden Fall geht es darum, den bei der Eingabe entstandenen Zeilenumbruch zu entfernen.
- `printf("Hallo %s!\n", name);`  
In dieser Anweisung wird der Befehl `printf()` zum ersten Mal für seinen eigentlichen Verwendungszweck eingesetzt: Die Zeichenfolge `%s` ist ein Platzhalter für einen String-Ausdruck. Das `\n` steht für einen Zeilenumbruch. Es gibt eine Reihe solcher speziellen Zeichenfolgen, die als **Escape-Sequenzen** bezeichnet werden. Der durch `%s` ersetzte Ausdruck wird durch ein Komma von der Formatangabe getrennt. In diesem Fall ist der Ausdruck die Variable `name` – der Benutzer wird also mit seinem zuvor eingegebenen Namen begrüßt.
- `return EXIT_SUCCESS;`  
Die Anweisung `return` beendet die Ausführung einer Funktion und gibt gegebenenfalls den Wert des angegebenen Ausdrucks an die aufrufende Stelle zurück. Wenn die Funktion `main()` den Wert `EXIT_SUCCESS` zurückliefert (auf den meisten Plattformen besitzt diese `stdlib.h`-Konstante den Wert 0), signalisiert sie dem Betriebssystem damit, dass alles in Ordnung ist. Um ein Programm mit einem Fehlerzustand zu beenden, wird dagegen die Konstante `EXIT_FAILURE` verwendet, die in der Regel den Wert 1 hat.

## Syntax- und Laufzeitfehler

Bei Fehlern in Computerprogrammen unterscheidet man zwischen *Syntaxfehlern*, die bereits bei der Übersetzung abgefangen werden, und *Laufzeitfehlern*, die erst bei der Ausführung auftreten. Beispielsweise ist eine Anweisung wie

```
str = "Dies ist ein Text;
```

syntaktisch falsch (das Anführungszeichen, das den String abschließen müsste, fehlt), und der Compiler fängt sie ab. Dividieren Sie dagegen zum Beispiel im Verlauf Ihres Programms durch eine Variable, deren Wert zufälligerweise 0 ist, bricht die Programmausführung mit einer Fehlermeldung ab. Um Laufzeitfehler zu verhindern, müssen Sie die Werte, mit denen Sie arbeiten, stets gründlich überprüfen – insbesondere Benutzereingaben, denn diese können sogar Auswirkungen auf die Sicherheit Ihrer Softwareumgebung haben.

## Elemente der Sprache C

Im letzten Abschnitt wurden bereits einige Merkmale der Programmiersprache C angesprochen. In diesem Abschnitt werden nun die wichtigsten Elemente von C systematisch behandelt.

### Die grundlegende Syntax

Ein C-Programm besteht grundsätzlich aus einer Abfolge von *Anweisungen*. Eine Anweisung entspricht einem einzelnen Verarbeitungsschritt, den Ihr Programm durchführen soll. Jede Anweisung steht in einer eigenen Zeile und endet mit einem Semikolon. Falls Ihnen eine Zeile zu lang erscheint, dürfen Sie an einer sinnvollen Stelle einen Backslash (\) einfügen und in der nächsten Zeile weiterschreiben. Dies darf allerdings nicht innerhalb der Anführungszeichen eines String-Literals geschehen.

Es gibt verschiedene Typen von Anweisungen. Die wichtigsten sind Funktionsaufrufe, Deklarationen, Wertzuweisungen und Kontrollstrukturen. Diese Anweisungsarten weisen folgende Eigenschaften auf:

- *Funktionsaufrufe* bestehen aus dem Namen der aufgerufenen Funktion und den zugehörigen Argumenten. Es kann sich sowohl um eingebaute als auch um selbst definierte Funktionen handeln. Beispiel:

```
printf("hallo");
```

- *Deklarationen* sind Variablen- oder Funktionsvereinbarungen. Beide Arten der Deklaration bestehen aus einem Datentyp und einem selbst gewählten Bezeichner (dem Namen des Elements). Variablen können auf Wunsch schon bei der Deklaration einen Wert erhalten. Funktionen besitzen optional beliebig viele Parameter, die als Variablen mit Datentypangabe in die Klammern hinter den Funktionsnamen geschrieben werden. Der Funktionsrumpf steht in geschweiften Klammern und besteht aus beliebig vielen Anweisungen. Beispiele:

```
int wert;           /* Variablendeklaration */  
float zahl = 2.75; /* Deklaration mit Wertzuweisung */
```

```
int summe (int a, int b)
{...}          /* Funktionsdefinition */
```

- **Wertzuweisungen** dienen dazu, einer Variablen einen Wert zuzuordnen. Eine Zuweisung hat die Form `variable = ausdrück`. Der Ausdruck wird zunächst ausgewertet, anschließend wird sein Wert in der Variablen gespeichert. Beispiele:

```
wert = 7;
zahl = 5 / 2;
```

- **Kontrollstrukturen** sind eine Sammelbezeichnung für Anweisungen, die der Flusskontrolle des Programms dienen, dazu gehören beispielsweise Fallunterscheidungen und Schleifen. Beispiel:

```
if (a < 0) {
    printf("a ist negativ");
}          /* Fallunterscheidung */
```

Neben den Anweisungen kann ein C-Programm **Kommentare** enthalten. Ein Kommentar steht zwischen den Zeichenfolgen `/*` und `*/` und kann beliebig viele Zeilen umfassen. Der Compiler ignoriert Kommentare; sie dienen dazu, Ihnen die Orientierung im Programmcode zu erleichtern. Kommentare dürfen nicht ineinander verschachtelt werden, da das erste Auftreten von `*/` den Kommentar bereits aufhebt.

Seit dem 1999 veröffentlichten Standard C99 dürfen auch einzeilige Kommentare verwendet werden, die ursprünglich in C++ eingeführt wurden und auch in Java und anderen Programmiersprachen bekannt sind. Diese beginnen mit den beiden Zeichen `//` und reichen bis zum Ende der aktuellen Zeile.

Leere Zeilen im Programmcode werden ignoriert, auch vor Anweisungen und zwischen den einzelnen Elementen einer Programmzeile dürfen beliebig viele Leerzeichen stehen. Der Ausdruck `a + b` ist äquivalent zu `a+b`. Allerdings dürfen Sie innerhalb von Bezeichnern keine Leerzeichen einfügen; auch einige Operatoren bestehen aus mehreren Zeichen, die nicht voneinander getrennt werden dürfen (beispielsweise `<=` oder `&&`).

### Gängige Formatierungsregeln

Bei der Verwendung von Leerzeichen im Code sollten Sie vor allem konsistent sein. Weitverbreitete Praxis sind folgende Regeln:

- Es wird je ein Leerzeichen vor und hinter einen Operanden gesetzt, zum Beispiel `a = b * c + d`.
- Zwischen Funktionsnamen und der öffnenden Klammer dahinter steht kein Leerzeichen, also etwa `printf(...)`.
- Bei Kontrollstrukturanweisungen wie `if` steht dagegen ein Leerzeichen zwischen dem Schlüsselwort und der geklammerten Bedingung, beispielsweise `if (...)`.

Bezeichner für Variablen und Funktionen dürfen aus Buchstaben, Ziffern und `_` (Unterstrich) bestehen. Sie dürfen allerdings nicht mit einer Ziffer beginnen. Der ANSI-C-Standard, an den sich im Prinzip alle aktuellen C-Versionen halten, schreibt vor, dass mindestens die ersten 31 Zeichen der

Bezeichner ausgewertet werden müssen. Sollten zwei Bezeichner erst beim 32. Zeichen voneinander abweichen, halten manche Compiler sie für ein und denselben. Bei Bezeichnern wird zwischen Groß- und Kleinschreibung unterschieden.

### Variablen

Wie bereits erwähnt, ist eine **Variable** ein benannter Speicherplatz. Anders als in der Mathematik besitzt eine Variable in einer Programmiersprache jederzeit einen definierten Wert. Es handelt sich also zur Laufzeit des Programms nicht um einen Platzhalter.

Variablen müssen zu Beginn jeder Funktion deklariert werden. Dies geschieht durch die Angabe des **Datentyps** und des **Bezeichners**; optional kann ein Anfangswert zugewiesen werden. Das folgende Beispiel deklariert die beiden Variablen `a` und `b`, wobei nur `b` ein Wert zugewiesen wird:

```
int a;  
double b = 2.5;
```

`a` wird als `int` (Integer oder Ganzzahl) deklariert, dient also der zukünftigen Speicherung einer ganzen Zahl. `b` erhält den Datentyp `double`, der zur Ablage doppelt genauer Fließkommazahlen verwendet wird.

Tabelle **Fehler! Kein Text mit angegebener Formatvorlage im Dokument..1** zeigt eine Übersicht über die wichtigsten einfachen Datentypen und ihre Bedeutung.

Datentyp	Bedeutung	Erläuterung
<code>int</code>	Integer (Ganzzahl)	eine ganze Zahl mit der Wortbreite des Prozessors: auf den meisten Rechnern 32 oder 64 Bit
<code>short</code>	kurzer Integer	Integer mit geringerer Speichergröße (oft 16 Bit)
<code>long</code>	langer Integer	Integer mit der größten Bit-Zahl (mindestens 32 Bit)
<code>char</code>	einzelnes Byte	8-Bit-Integer; wird oft zur Darstellung von ASCII-Zeichen verwendet – daher der Name <code>char</code> (für <i>character</i> )
<code>float</code>	Fließkommawert	speichert Fließkommazahlen

		mit einfacher Genauigkeit (meist 32 Bit)
<code>double</code>	Fließkommawert	doppelt genauer Fließkommawert (in der Regel 64 Bit)

Tabelle **Fehler! Kein Text mit angegebener Formatvorlage im Dokument..1**: Die elementaren C-Datentypen

Die ganzzahligen Datentypen `int`, `short`, `long` und `char` speichern einen Integer-Wert je nach Bedarf mit oder ohne Vorzeichen. Sie können der Deklaration `signed` voranstellen, um Werte mit Vorzeichen zu speichern, oder `unsigned` für Werte ohne Vorzeichen. Dies bedeutet beispielsweise für einen 32-Bit-Integer, dass bei `signed` Werte zwischen  $-2.147.483.648$  und  $+2.147.483.647$  möglich sind, während `unsigned` die Werte 0 bis  $4.294.967.295$  zulässt.

Standardmäßig sind alle Integer-Werte `signed`, ein Sonderfall ist lediglich `char`: Da dieser Typ in der Regel zur Darstellung einzelner ASCII-Zeichen eingesetzt wird, ist er zumindest in diesem Zusammenhang `unsigned`. Für die Zeichen modernerer Zeichensätze wie Unicode, die mehr Speicher benötigen als 8 Bit, könnten Sie einfach `unsigned int` benutzen, empfehlenswerter ist jedoch die Verwendung des speziellen Typs `wchar_t`, wofür Sie allerdings mithilfe von `#include` die Header-Datei `stddef.h` einbinden müssen.

Auffällig ist, dass es in C keinen separaten Datentyp für boolesche Wahrheitswerte gibt. Folgerichtig gelten Ausdrücke mit dem Wert 0 als falsch und alle anderen als wahr.

Ein weiteres Merkmal von Variablen ist ihr **Gültigkeitsbereich (Scope)**, der festlegt, in welchen Programmteilen eine Variable definiert bleibt. Grundsätzlich werden zwei verschiedene Arten von Variablen unterschieden:

- **Lokale Variablen**, auch **automatische Variablen** genannt, gelten nur innerhalb der Funktion, in der sie deklariert wurden. Dies gilt sowohl für Variablen, die zu Beginn des Funktionsrumpfes definiert werden, als auch für Parametervariablen, die in den Klammern hinter dem Funktionsnamen aufgeführt werden.
- **Globale Variablen** werden zu Beginn des Programmcodes (hinter eventuellen Präprozessordirektiven) deklariert und gelten im gesamten Programm. Falls Sie allerdings innerhalb einer Funktion eine gleichnamige Variable neu deklarieren, wird dort nur diese lokale Variable verwendet.

Eine besondere Form der lokalen Variablen sind die **statischen Variablen**: Wenn Sie in einer Funktion eine Variablendeklaration mit dem Schlüsselwort `static` einleiten, gilt die Variable zwar nur innerhalb dieser Funktion, behält aber ihren Wert bis zum nächsten Aufruf der Funktion bei.



## Ausdrücke und Operationen

Zu den wichtigsten Fähigkeiten von Programmiersprachen gehört die Auswertung beziehungsweise Berechnung von **Ausdrücken**. An jeder Stelle, an der ein bestimmter Wert erwartet wird, kann stattdessen ein komplexer Ausdruck stehen, der zunächst ausgewertet und anschließend als Wert eingesetzt wird. Voraussetzung ist allerdings, dass der Ausdruck einen passenden Datentyp besitzt.

Die einfachsten Bestandteile von Ausdrücken sind die **Literale**. Es handelt sich dabei um Werte, die nicht weiter berechnet oder ersetzt werden müssen. C unterscheidet die folgenden vier Arten von Literalen:

- **Integer-Literale** dienen der Darstellung ganzzahliger Werte. Normalerweise werden sie dezimal notiert; dazu wird einfach die entsprechende Zahl mit optionalem Vorzeichen geschrieben. Wenn Sie einem Integer-Wert eine 0 voranstellen, wird er als Oktalzahl interpretiert: 033 bedeutet demzufolge nicht 33, sondern 27. Ein vorangestelltes 0x kennzeichnet dagegen eine Hexadezimalzahl: 0x33 steht also für den (dezimalen) Wert 51.
- **Fließkomma-Literale** repräsentieren Fließkommawerte. Beachten Sie, dass in C und anderen Programmiersprachen nicht das Komma als Dezimaltrennzeichen verwendet wird, sondern der Punkt. Alternativ können Sie Fließkomma-Literale in wissenschaftlicher Schreibweise (Exponentialschreibweise) angeben: 3.5E+5 steht beispielsweise für  $3,5 \times 10^5$  (350.000), 4.78E-4 hat dagegen den Wert  $4,78 \times 10^{-4}$  (0,000478).
- **Zeichen-Literale** enthalten ein einzelnes Zeichen aus einem Zeichensatz, mit dem der Compiler umgehen kann. Ein Zeichen-Literal muss in einfachen Anführungszeichen stehen, beispielsweise 'a'.
- **String-Literale** enthalten Zeichenketten, das heißt beliebig lange Textblöcke. Sie müssen in doppelten Anführungszeichen stehen, etwa "hallo". Ein Sonderfall ist die leere Zeichenkette, die durch zwei unmittelbar aufeinanderfolgende Anführungszeichen dargestellt wird: "". Sie wird wie der Wert 0 bei Fallunterscheidungen als falsch betrachtet.

Ein weiterer Bestandteil von Ausdrücken sind Variablen, die bei der Auswertung jeweils durch ihren aktuellen Wert ersetzt werden:

```
a = 5;
b = a + 7;          /* b hat nun den Wert 12 */
```

Mitunter besitzt eine Variable, die Sie in einem Ausdruck verwenden möchten, den falschen Datentyp. C konvertiert den Datentyp immer dann automatisch, wenn keine Gefahr besteht, dass dabei Werte verloren gehen oder verfälscht werden. Beispielsweise wird int ohne Weiteres dort akzeptiert, wo eigentlich ein Fließkommatyp erwartet wird. In Fällen, in denen diese Gefahr besteht, müssen Sie die Typumwandlung dagegen explizit anordnen. Dies geschieht durch das sogenannte **Typecasting**. Dabei wird der gewünschte Datentyp in Klammern vor die umzuwandelnde Variable oder auch vor ein Literal geschrieben:

```
double a = 4.7;
printf("%d", (int)a);
/* Ausgabe: 4 */
```

Sie können innerhalb eines Ausdrucks sogar eine Funktion aufrufen, vorausgesetzt, sie liefert einen Wert mit dem passenden Datentyp zurück:

```
a = sin(b); /* a enthält den Sinus von b */
```

Neben all diesen Elementen, die jeweils einen einzelnen Wert ergeben, können Ausdrücke auch **Operatoren** enthalten. Diese dienen dazu, verschiedene Werte arithmetisch oder logisch miteinander zu verknüpfen. Beachten Sie, dass nicht jeder Operator zu jedem Datentyp passt.

Die **arithmetischen Operatoren** sind **+** (Addition), **-** (Subtraktion), **\*** (Multiplikation), **/** (Division) und **%** (Modulo; der Rest einer ganzzahligen Division).

Die nächste Gruppe bilden die **logischen Operatoren**. Sie dienen dazu, Werte nach logischen Kriterien miteinander zu verknüpfen:

- Das **logische Und** (geschrieben **&&**) erzeugt den Wert 0, wenn mindestens einer der verknüpften Ausdrücke 0 (logisch falsch) ist, andernfalls erhält der Gesamtausdruck einen von 0 verschiedenen Wert und gilt damit als wahr.
- Das **logische Oder** wird als **||** notiert und erhält einen von 0 verschiedenen Wert, sobald mindestens einer der verknüpften Ausdrücke von 0 verschieden ist.
- Das **logische Nicht** wird durch ein **!** dargestellt, das dem zu negierenden Ausdruck vorangestellt wird. Der Ausdruck erhält dadurch den Wert 0, wenn er zuvor einen anderen Wert hatte, und 1, wenn sein ursprünglicher Wert 0 war.

Ähnlich wie die logischen Operatoren, aber auf einer anderen Ebene, arbeiten die **Bit-Operatoren**: Sie manipulieren den Wert ihrer Operanden bitweise, betrachten also jedes einzelne Bit. Im Einzelnen sind die folgenden Bit-Operatoren definiert:

- Das **bitweise Und** (geschrieben **&**) setzt im Ergebnis diejenigen Bits auf den Wert 1, die in beiden Operanden 1 sind, alle anderen auf 0. Beispiel: **94 & 73** führt zu dem Ergebnis 72. Erläutern lässt sich dieses Ergebnis nur anhand der binären Darstellung:

```
0101 1110
& 0100 1001
-----
0100 1000
```

- Das **bitweise Oder** (**|**) setzt alle Bits auf 1, die in mindestens einem der Operanden den Wert 1 haben. **94 | 73** ergibt demzufolge 95:

```
0101 1110
| 0100 1001
-----
0101 1111
```

- Das **bitweise exklusive Oder** (^) setzt nur diejenigen Bits auf 1, die in genau einem Operanden den Wert 1 haben, alle anderen dagegen auf 0. `94 ^ 73` liefert das Ergebnis 23:

```

0101 1110
^ 0100 1001
-----
0001 0111

```

- Die **Bit-Verschiebung nach links** (<<) verschiebt die Bits des ersten Operanden um die Anzahl von Stellen nach links, die der zweite Operand angibt. Beispiel: `73 << 2` ergibt 292, entspricht also einer Multiplikation mit  $2^2$  (4).
- Die **Bit-Verschiebung nach rechts** (>>) verschiebt die Bits des ersten Operanden dagegen um die angegebene Anzahl von Stellen nach rechts. Beispiel: `94 >> 3` führt zu dem Ergebnis 11, da die letzten drei Binärstellen wegfallen.
- Die **bitweise Negation** oder das **Bit-Komplement** (eine vorangestellte Tilde ~) setzt alle Bits mit dem Wert 1 auf 0 und umgekehrt. Das Ergebnis ist abhängig von der Bit-Breite des entsprechenden Werts. Beispiel: `~73` ergibt als **unsigned** 8-Bit-Integer den Wert 182.

Für die Ablaufsteuerung von Programmen sind die **Vergleichsoperatoren** von besonderer Bedeutung: Sie vergleichen Ausdrücke miteinander und liefern je nach Ergebnis 0 oder einen wahren Wert. Es sind folgende Vergleichsoperatoren definiert:

- `==` ist der Gleichheitsoperator; das Ergebnis ist wahr, wenn die beiden verglichenen Ausdrücke gleich sind.
- `!=` überprüft die Ungleichheit von Ausdrücken, ist also wahr, wenn sie verschieden sind.
- `<` ist wahr, wenn der linke Operand kleiner ist als der rechte.
- `>` ist wahr, wenn der linke Operand größer ist als der rechte.
- `<=` ist wahr, wenn der linke Operand kleiner oder gleich dem rechten ist. Diese Operation ist die Negierung von `>`.
- `>=` ist wahr, wenn der linke Operand größer oder gleich dem rechten ist. Dies ist die Negierung von `<`.

Zu guter Letzt gibt es noch den **Wertzuweisungsoperator** =, der dem Operanden auf der linken Seite den Wert des Ausdrucks auf der rechten Seite zuweist. Bei dem linken Operanden handelt es sich in der Regel um eine Variable. Allgemein werden Elemente, die auf der linken Seite einer Wertzuweisung stehen können, als **LVALUE** bezeichnet.

Sehr häufig kommt es vor, dass eine Wertzuweisung den ursprünglichen Wert des LVALUE ändert, sodass der LVALUE selbst in dem Ausdruck auf der rechten Seite auftaucht. Für diese speziellen Fälle wurden einige Abkürzungen definiert; die wichtigsten sind in Tabelle **Fehler! Kein Text mit angegebener Formatvorlage im Dokument..2** aufgeführt.

Langfassung	Kurzfassung	Erläuterung
<code>a = a + 5;</code>	<code>a += 5;</code>	LVALUE um den angegebenen Wert erhöhen
<code>a = a + 1;</code>	<code>a += 1;</code> <code>a++;</code> <code>++a;</code>	LVALUE um 1 erhöhen (beachten Sie dabei die Erläuterung im Anschluss an die Tabelle)
<code>a = a - 5;</code>	<code>a -= 5;</code>	LVALUE um den angegebenen Wert vermindern
<code>a = a - 1;</code>	<code>a -= 1;</code> <code>a--;</code> <code>--a;</code>	LVALUE um 1 vermindern
<code>a = a * 5;</code>	<code>a *= 5;</code>	LVALUE mit dem angegebenen Wert multiplizieren
<code>a = a / 5;</code>	<code>a /= 5;</code>	LVALUE durch den angegebenen Wert dividieren

Tabelle **Fehler! Kein Text mit angegebener Formatvorlage im Dokument.**2: Die wichtigsten Abkürzungen für kombinierte Wertzuweisungen

Neben den in der Tabelle angegebenen Abkürzungen gibt es auch für die logischen Operatoren und die Bit-Operatoren entsprechende Schreibweisen.

Eine Sonderstellung nehmen die Operatoren ein, die einen LVALUE um 1 erhöhen oder vermindern: Sie können `++` oder `--` entweder vor oder hinter den LVALUE schreiben. Wenn Sie dies als einzelne Anweisung hinschreiben, besteht zwischen diesen Varianten kein Unterschied. Werden sie dagegen im Rahmen eines komplexen Ausdrucks verwendet, dann ist der Unterschied folgender:

- Das vorangestellte `++` wird **Prü-Inkrement** genannt. Es erhöht den LVALUE um 1 und verwendet den neuen Wert innerhalb des Ausdrucks:

```
a = 1;
b = ++a;           /* a hat den Wert 2, b auch. */
```

Entsprechend heißt ein vorangestelltes `--` **Prü-Dekrement**. Der LVALUE wird um 1 vermindert, bevor er in einem Ausdruck verwendet wird.

- Ein nachgestelltes `++` wird als **Post-Inkrement** bezeichnet. Ein LVALUE mit Post-Inkrement wird in einem Ausdruck mit seinem alten Wert verwendet und erst danach um 1 erhöht:

```
a = 2;
b = a++;          /* a hat den Wert 3, b bleibt 2 */
```

Das nachgestellte `--` heißt **Post-Dekrement**. Der alte Wert des LVALUE wird im Ausdruck verwendet, bevor es um 1 vermindert wird.

Ein besonderer Operator ist der **Fallunterscheidungsoperator**: Die Schreibweise `Ausdruck1 ? Ausdruck2 : Ausdruck3` hat `Ausdruck2` als Ergebnis, wenn `Ausdruck1` wahr ist, ansonsten ist der Wert `Ausdruck3`. Da er als einziger Operator drei Operanden hat, wird er auch als **ternärer** (dreigliedriger) Operator bezeichnet; entsprechend heißen die meisten Operatoren **binär** (zwei Operanden, zum Beispiel alle arithmetischen Operationen) beziehungsweise **unär** (ein Operand, etwa Vorzeichen). Hier zwei Beispiele:

```
a = 2;
b = (a == 1 ? 3 : 5);
/* a ist nicht 1, also erhält b den Wert 5 */
```

```
a = 1;
b = (a == 1 ? 3 : 5);
/* a ist 1, also erhält b den Wert 3 */
```

Bei der Arbeit mit Operatoren ist zu beachten, dass sie mit unterschiedlicher Priorität ausgewertet werden. Die folgende Liste stellt die Rangfolge der Operatoren in absteigender Reihenfolge dar. Die weiter oben stehenden Operatoren binden also stärker und werden zuerst aufgelöst:

- `!, ~, ++, --, +` (Vorzeichen), `-` (Vorzeichen)
- `*, /, %`
- `+` und `-` (arithmetische Operatoren)
- `<<` und `>>`
- `<, <=, >, >=`
- `==` und `!=`
- `&` (bitweises Und)
- `^` (bitweises Exklusiv-Oder)
- `|` (bitweises Oder)
- `&&` (logisches Und)
- `||` (logisches Oder)
- `?:` (Operator für Fallunterscheidungen)
- `=, +=, -=` etc.

Sie können die Rangfolge der Operatoren durch die Verwendung von Klammern verändern. Beispielsweise besitzt der Ausdruck `3 * 4 + 7` den Wert 19, während `3 * (4 + 7)` das Ergebnis 33

liefert. Beachten Sie, dass zu diesem Zweck – anders als in der Mathematik – immer nur runde Klammern verwendet werden dürfen, egal wie tief sie verschachtelt werden!

### Kontrollstrukturen

Eine der wesentlichen Aufgaben von Computerprogrammen besteht darin, den Programmablauf in Abhängigkeit von bestimmten Bedingungen zu steuern. Dazu definiert C eine Reihe sogenannter **Kontrollstrukturen**, die man grob in Fallunterscheidungen und Schleifen unterteilen kann. Eine **Fallunterscheidung** überprüft die Gültigkeit einer Bedingung und führt abhängig davon bestimmte Anweisungen aus; eine **Schleife** sorgt dagegen dafür, dass bestimmte Anweisungsfolgen mehrmals hintereinander ausgeführt werden.

Die einfachste und wichtigste Kontrollstruktur ist die Fallunterscheidung mit `if()`. Der Ausdruck, der hinter dem Schlüsselwort `if` in Klammern steht, wird ausgewertet. Wenn er wahr (nicht 0) ist, wird die auf das `if` folgende Anweisung ausgeführt. Das nächste Beispiel überprüft, ob die Variable `a` größer als 100 ist, und gibt in diesem Fall »Herzlichen Glückwunsch!« aus:

```
if (a > 100)
    printf("Herzlichen Glückwunsch!\n");
```

Mitunter hängen mehrere Anweisungen von einem einzelnen `if()` ab. In diesem Fall müssen Sie hinter die Bedingungsprüfung einen **Anweisungsblock** schreiben, also eine Sequenz von Anweisungen in geschweiften Klammern. Das folgende Beispiel überprüft, ob die Variable `b` kleiner als 0 ist. In diesem Fall setzt sie `b` auf 100 und gibt eine entsprechende Meldung aus:

```
if (b < 0) {
    b = 100;
    printf("b auf 100 gesetzt.\n");
}
```

Die öffnende geschweifte Klammer schreiben manche Programmierer lieber in die nächste Zeile. Beide Varianten sind üblich und zulässig, Sie sollten sich allerdings konsequent an eine davon halten. In diesem Buch wird die Klammer stets in dieselbe Zeile gesetzt, allein schon aus Platzgründen.

### Verwenden Sie stets geschweifte Klammern!

Letzten Endes lohnt es sich übrigens, auch bei `if`-Abfragen, von denen nur eine einzige Anweisung abhängt, geschweifte Klammern zu verwenden. Erstens kann Ihnen so nicht der Fehler passieren, dass Sie die Klammern vergessen, wenn später weitere Anweisungen dazukommen. Und zweitens gibt es andere Programmiersprachen wie etwa Perl, bei denen die Klammern zwingend vorgeschrieben sind.

Für die Formulierung des Bedingungsausdrucks bieten sich einige Abkürzungen an, die mit der logischen Interpretation von 0 und anderen Werten zu tun haben. Wollen Sie beispielsweise Anweisungen ausführen, wenn die Variable `a` den Wert 0 hat, können Sie entweder die ausführliche Bedingung `a == 0` schreiben oder die Abkürzung `!a` verwenden: Die Negation von `a` ist genau dann wahr, wenn `a` gleich 0 ist. Sollen dagegen Anweisungen ausgeführt werden, wenn `a` nicht 0 ist, genügt

sogar ein einfaches `a` als Bedingung. Diese Nachlässigkeit bei der Überprüfung von Datentypen macht C zu einer sogenannten *schwach typisierten Sprache*: Variablen besitzen festgelegte Datentypen, diese werden aber bei Bedarf sehr großzügig ineinander konvertiert.

Es kommt sehr häufig vor, dass auch bei Nichtzutreffen einer Bedingung spezielle Anweisungen ausgeführt werden sollen. Zu diesem Zweck besteht die Möglichkeit, hinter einer `if`-Abfrage einen `else`-Teil zu platzieren. Die Anweisung oder der Block hinter dem `else` wird genau dann ausgeführt, wenn die Bedingung der `if`-Abfrage nicht zutrifft. Das folgende Beispiel gibt »a ist positiv.« aus, wenn `a` größer als 0 ist, ansonsten wird »a ist nicht positiv.« zurückgegeben:

```
if (a > 0)
    printf("a ist positiv.\n");
else
    printf("a ist nicht positiv.\n");
```

Auch hinter dem `else` kann alternativ ein Block von Anweisungen in geschweiften Klammern folgen:

```
if (a > 0) {
    /* Ausgabe: a ist positiv. */
    printf("a ist positiv.\n");
} else {
    /* Ausgabe: a ist nicht positiv. */
    printf("a ist nicht positiv.\n");
}
```

Sie können hinter dem `else` sogar wieder erneut ein `if` unterbringen, falls eine weitere Bedingung überprüft werden soll, wenn die ursprüngliche Bedingung nicht erfüllt ist. Die folgende Abfrage erweitert das vorangegangene Beispiel so, dass auch die Fälle 0 und negativer Wert unterschieden werden:

```
if (a > 0)
    printf("a ist positiv.\n");
else if (a < 0)
    printf("a ist negativ.\n");
else
    printf("a ist 0.\n");
```

Alternativschreibweise mit geschweiften Klammern:

```
if (a > 0) {
    printf("a ist positiv.\n");
} else if (a < 0) {
    printf("a ist negativ.\n");
} else {
```

```
    printf("a ist 0.\n");
}
```

Das folgende kleine Beispielprogramm verwendet verschachtelte `if-else`-Abfragen, um aus einer eingegebenen Punktzahl in einer Prüfung die zugehörige Note nach dem IHK-Notenschlüssel zu berechnen:

```
#include <stdio.h>

int main() {
    int punkte;
    int note;
    printf("Ihre Punktzahl, bitte: ");
    scanf("%d", &punkte);
    if (punkte < 30)
        note = 6;
    else if (punkte < 50)
        note = 5;
    else if (punkte < 67)
        note = 4;
    else if (punkte < 81)
        note = 3;
    else if (punkte < 92)
        note = 2;
    else
        note = 1;

    printf("Sie haben die Note %d erreicht.\n", note);
    return 0;
}
```

Die Funktion `scanf()` dient übrigens dazu, Daten verschiedener Formate einzulesen – im Gegensatz zu der zuvor verwendeten Funktion `fgets()`, die nur zum Einlesen von Strings verwendet wird. Die Formatangabe `%d` steht für einen Integer (die Abkürzung `d` bedeutet *decimal*). Wichtig ist die Angabe des `&`-Zeichens vor dem Variablennamen – damit wird die Adresse und nicht der Wert angegeben, denn `scanf()` muss wissen, wo der eingelesene Wert gespeichert werden soll. Näheres zu diesem Thema finden Sie im übernächsten Unterabschnitt »Zeiger und Arrays«.

In anderen Fällen kommt es vor, dass Sie eine Variable nacheinander mit verschiedenen festen Werten vergleichen müssen, nicht mit Wertebereichen wie im vorangegangenen Beispiel. Für diesen Verwendungszweck bietet C die spezielle Struktur `switch/case` an. Das Argument von `switch` muss ein LVALUE sein, das nacheinander mit einer Liste von Werten verglichen wird, die hinter dem Schlüsselwort `case` stehen. Die einzelnen `case`-Unterscheidungen stellen dabei Einstiegspunkte in



den `switch`-Codeblock dar. Wenn das LVALUE einem der Werte in der Liste entspricht, wird von dieser Stelle an der gesamte Block ausgeführt. Da dieses Verhalten oft unerwünscht ist, wird der Block vor jedem neuen `case` meist mithilfe von `break` verlassen.

Das folgende Beispiel ermittelt aus einer numerischen Note die entsprechende Zensur in Textform:

```
switch (note) {
    case 6:
        printf("ungenügend\n");
        break;
    case 5:
        printf("mangelhaft\n");
        break;
    case 4:
        printf("ausreichend\n");
        break;
    case 3:
        printf("befriedigend\n");
        break;
    case 2:
        printf("gut\n");
        break;
    case 1:
        printf("sehr gut\n");
        break;
    default:
        printf("unzulässige Eingabe\n");
}
```

Hinter der optionalen Markierung `default` können Anweisungen stehen, die ausgeführt werden, wenn der geprüfte LVALUE keinen der konkreten Werte hat. Dies eignet sich insbesondere, um Fehleingaben abzufangen.

Eine grundlegend andere Art von Kontrollstrukturen sind **Schleifen**. Sie sorgen dafür, dass ein bestimmter Codeblock mehrmals ausgeführt wird, entweder abhängig von einer Bedingung oder mit einer definierten Anzahl von Durchläufen.

Die einfachste Form der Schleife ist die `while()`-Schleife. In den Klammern hinter dem Schlüsselwort `while` wird genau wie bei `if()` eine Bedingung geprüft. Trifft sie zu, wird der **Schleifenrumpf** (die Anweisung oder der Block hinter dem `while`) ausgeführt. Nach der Ausführung wird die Bedingung erneut überprüft. Solange sie zutrifft, wird der Schleifenrumpf immer wieder ausgeführt. Das folgende Beispiel überprüft vor jedem Durchlauf, ob die Variable `i` noch kleiner als 10 ist, und erhöht sie innerhalb des Schleifenrumpfes jeweils um 1:

```
i = 0;
while (i < 10) {
    printf("%d\t", i);
    i++;
}
```

`i` ist der bevorzugte Name für Schleifenzählervariablen. Diese Tradition stammt aus der Mathematik, wo `i` oft als Zähler bei Summenformeln oder Ähnlichem eingesetzt wird (Abkürzung für *index*). Wenn mehrere Schleifen ineinander verschachtelt werden, heißen deren Zählervariablen `j`, `k`, `l` und so fort.

Die Ausgabe dieses kurzen Beispiels (`\t` steht für einen Tabulator) sieht folgendermaßen aus:

```
0  1  2  3  4  5  6  7  8  9
```

Da die Überprüfung der Bedingung vor dem jeweiligen Durchlauf erfolgt, findet der Abbruch statt, sobald `i` nicht mehr kleiner als 10 ist. Eine solche Schleifenkonstruktion wird als **kopfgesteuerte** Schleife bezeichnet.

Eine andere Art der Schleife überprüft die Bedingung erst nach dem jeweiligen Durchlauf und heißt deshalb **fußgesteuert**. In C wird sie durch die Schreibweise `do ... while()` realisiert. Diese Art der Schleife ist nützlich, wenn die zu überprüfende Bedingung sich erst aus dem Durchlauf selbst ergibt, beispielsweise bei der Prüfung von Benutzereingaben. Das vorangegangene Beispiel sieht als fußgesteuerte `do-while`-Schleife so aus:

```
i = 0;
do {
    printf("%d\t", i);
    i++;
} while (i < 10);
```

Interessanterweise stellt sich die Ausgabe dieser Schleife etwas anders dar:

```
0  1  2  3  4  5  6  7  8  9  10
```

Da die Bedingung erst nach der Ausgabe geprüft wird, erfolgt der Abbruch erst einen Durchlauf später. Anders als die kopfgesteuerte Schleife wird die fußgesteuerte immer *mindestens einmal* ausgeführt. Beachten Sie, dass hinter dem `while()` in diesem Fall ein Semikolon stehen muss.

Eine alternative Schreibweise für Schleifen ist die `for`-Schleife. Sie wird bevorzugt in Fällen eingesetzt, in denen eine festgelegte Anzahl von Durchläufen erwünscht ist. Die allgemeine Syntax dieser Schleife ist folgende:

```
for (Initialisierung; Wertüberprüfung; Wertänderung)
    Anweisung;
```

Die Initialisierung wird genau einmal vor dem Beginn der Schleife ausgeführt. Die Wertüberprüfung findet vor jedem Durchlauf statt. Wenn sie einen wahren Wert ergibt, wird der Schleifenrumpf ein weiteres Mal ausgeführt. Nach jedem Durchlauf findet die Wertänderung statt. Beispiel:

```
for (i = 0; i < 10; i++) {
    printf("%d\t", i);
}
```

Dies erzeugt exakt dieselbe Ausgabe wie das vorangegangene kopfgesteuerte `while`-Beispiel; der Code ist sogar absolut äquivalent. Jede `for`-Schleife lässt sich auf diese Weise durch eine `while`-Schleife ersetzen. Es handelt sich lediglich um eine spezielle Formulierung, die für determinierte Schleifen (Schleifen mit festgelegter Anzahl von Durchläufen) besser geeignet ist.

### Funktionen

Wie bereits erwähnt, besteht ein C-Programm aus beliebig vielen Funktionen, die Sie innerhalb Ihres Programms von jeder Stelle aus aufrufen können. Die wichtigste Funktion ist `main()`, weil sie die Aufgabe des Hauptprogramms übernimmt.

Eine Funktion kann jeden der eingangs für Variablen genannten Datentypen innehaben. Es wird erwartet, dass eine Funktion mit einem bestimmten Datentyp mithilfe von `return` einen Wert dieses Typs an die aufrufende Stelle zurückgibt. Eine Funktion, die »nur« bestimmte Anweisungen ausführen, aber keinen Wert zurückgeben soll, kann den speziellen Datentyp `void` haben.

Die wichtigste Aufgabe von Funktionen ist die Strukturierung des Programms. Es lohnt sich, häufig benötigte Anweisungsfolgen in separate Funktionen zu schreiben und bei Bedarf aufzurufen. Eine solche **Modularisierung** des Codes macht das Programm übersichtlicher, weil Sie sich in jeder Funktion auf eine einzelne Aufgabe konzentrieren können. Auf diese Weise lassen sich mehrere Abstraktionsebenen in ein Programm einführen: Grundlegende Bausteine können einmal implementiert und zu immer komplexeren Einheiten zusammengesetzt werden.

Eine Funktion kann nicht nur einen Rückgabewert haben, sondern auch einen oder mehrere Eingabewerte, die in Form von Parametervariablen in die Klammern hinter dem Funktionsnamen geschrieben werden. Eine Funktion mit Parametern erwartet die Übergabe entsprechend vieler Werte mit dem korrekten Datentyp. Aus Sicht der aufrufenden Stelle werden diese Werte als **Argumente der Funktion** bezeichnet, innerhalb der Funktion können sie wie normale lokale Variablen verwendet werden.

Das folgende Beispiel zeigt eine Funktion namens `verdoppeln()`, die einen Wert vom Datentyp `int` entgegennimmt und das Doppelte dieses Werts zurückgibt:

```
int verdoppeln(int wert) {
    return 2 * wert;
}
```

Diese Funktion kann von einer beliebigen Programmstelle aus innerhalb eines Ausdrucks aufgerufen werden, wenn an der entsprechenden Stelle ein Integer-Wert zulässig ist. Im folgenden Beispiel wird `verdoppeln()` aus einer `printf()`-Anweisung heraus aufgerufen, um das Doppelte der Variablen `b` auszugeben:

```
printf("%d\n", verdoppeln(b));
```

Eine Funktion vom Datentyp `void` wird dagegen als einzelne Anweisung aufgerufen. Das folgende Beispiel definiert eine Funktion namens `begruessen()`, die einen Gruß für den angegebenen Namen ausgibt:

```
void begruessen(char[] name) {  
    printf("Hallo, %s!\n", name);  
}
```

Ein Aufruf dieser Funktion sieht etwa folgendermaßen aus:

```
begruessen("Klaus");
```

Die Ausgabe lautet natürlich so:

```
Hallo, Klaus!
```

Übrigens kann auch die Funktion `main()` so geschrieben werden, dass sie Argumente entgegennimmt. Dies dient der Annahme von Kommandozeilenparametern. Die standardisierte Syntax für die Parameter von `main()` lautet folgendermaßen:

```
int main(int argc, char *argv[])
```

Die Variable `argc` enthält dabei die Anzahl der übergebenen Argumente, während das Array `argv[]` die einzelnen Argumentwerte als Strings aufnimmt. `argv[0]` enthält dabei kein echtes Argument, sondern den Namen des aufgerufenen Programms. Arrays werden im folgenden Abschnitt näher erläutert.

### **Zeiger und Arrays**

Der wichtigste Grund dafür, dass C als schwierig zu erlernen und zu benutzen gilt, ist die Tatsache, dass in dieser Sprache mit **Zeigern** operiert werden kann. Ein Zeiger ist eine spezielle Variable, deren Wert eine Speicheradresse ist. Im Grunde handelt es sich dabei also um eine Art indirekte Variable: Eine »normale« Variable ist ein benannter Speicherplatz, in dem unmittelbar ein konkreter Wert abgelegt wird, ein Zeiger verweist dagegen auf den Ort, an dem sich der konkrete Wert befindet.

Zeiger sind unter anderem wichtig, damit Funktionen einander den Speicherort bestimmter Werte mitteilen können, um diese Werte gemeinsam zu manipulieren. Ein Zeiger verweist jeweils auf einen Speicherplatz mit einem bestimmten Datentyp. Er unterscheidet sich von einer Variablen dieses Datentyps durch ein vorangestelltes `*`:

```
int a;                /* normale int-Variable */  
int *b;               /* Zeiger auf int */
```

Der Wert, der einer Zeigervariablen zugewiesen wird, ist normalerweise die Adresse einer anderen Variablen. Diese wird durch den Dereferenzierungsoperator, ein vorangestelltes `&`, ermittelt. Im folgenden Beispiel wird der Zeigervariablen `a` die Adresse von `b` als Wert zugewiesen:

```
int b = 9;  
int *a = &b;          /* a zeigt auf b */
```

Würden Sie daraufhin versuchen, den Wert von `a` selbst auszugeben, wäre das Ergebnis die unvorhersagbare und völlig sinnfreie Nummer einer Speicheradresse. Wenn Sie dagegen den Wert von `*a` ausgeben, erhalten Sie den Inhalt von `b`.

Die interessante Frage ist natürlich, wozu man so etwas überhaupt benötigt. Ein gutes Beispiel ist eine Funktion, die den tatsächlichen Wert einer Variablen ändert, die ihr als Argument übergeben wird. Ein solcher Funktionsaufruf wird als **Call by Reference** bezeichnet im Gegensatz zur einfachen Wertübergabe, die auch **Call by Value** heißt. Die folgenden beiden Funktionen demonstrieren diesen Unterschied:

```
/* Call by Reference */
void doppel1(int *a) {
    *a *= 2;
}

/* Call by Value */
void doppel2(int a) {
    a *= 2;
}
```

Wenn die zweite Funktion mit einer Variablen als Argument aufgerufen wird, ändert diese Variable selbst ihren Wert nicht:

```
b = 3;
doppel2(b);          /* Wert von b: 3 */
```

Die erste Funktion wird dagegen mit der Adresse einer Variablen aufgerufen und manipuliert unmittelbar den Inhalt dieser Speicherstelle:

```
b = 3;
doppel1(&b);         /* Wert von b: 6 */
```

Nahe Verwandte der Zeiger sind die **Arrays**. Es handelt sich dabei um Variablen, die mehrere durch einen numerischen Index ansprechbare Werte besitzen. Realisiert werden Arrays durch hintereinanderliegende Speicherstellen, in denen die einzelnen Werte abgelegt werden. Jedes Array lässt sich alternativ durch einen Zeiger auf die Speicherstelle des ersten Elements beschreiben. Die weiteren Elemente können angesprochen werden, indem zu dieser Adresse die Anzahl der Bytes addiert wird, die ein einzelnes Element einnimmt.

Ein Array wird deklariert, indem hinter dem Variablennamen die gewünschte Anzahl von Elementen in eckigen Klammern angegeben wird:

```
int a[10];           /* 10 int-Werte */
```

Die zehn Elemente des Arrays `a[]` werden als `a[0]` bis `a[9]` angesprochen. Alternativ können Sie die Zeiger-Schreibweise wählen: Die Elemente heißen dann `*a` bis `*(a + 9)`.

Sie können einem Array bei der Deklaration auch Anfangswerte zuweisen und dabei die Anzahl der Elemente weglassen, weil sie implizit feststeht:

```
int test[] = {1, 2, 3, 4, 5};
```

Das folgende Beispiel definiert ein Array mit zehn Werten vom Datentyp `int`, die nacheinander vom Benutzer eingegeben werden. Anschließend gibt das Programm das gesamte Array sowie den kleinsten und den größten enthaltenen Wert aus:

```
#include <stdio.h>

int main() {
    int werte[10];
    int ein;
    int i, min, max;
    printf("Bitte zehn Werte zwischen 1 und 100!\n");
    for (i = 0; i < 10; i++) {
        printf("%d. Wert: ", i + 1);
        scanf("%d", &ein);
        werte[i] = ein;
    }
    /* max und min auf das Anfangselement setzen: */
    min = werte[0];
    max = werte[0];
    printf("Ihre Werte: ");
    for (i = 0; i < 10; i++) {
        printf("%d ", werte[i]);
        if (werte[i] > max)
            max = werte[i];
        if (werte[i] < min)
            min = werte[i];
    }
    printf("\n");
    printf("Kleinsten Wert: %d\n", min);
    printf("Größten Wert: %d\n", max);
    return 0;
}
```

Eine der wichtigsten Aufgaben von Arrays besteht darin, den nicht vorhandenen String-Datentyp zu ersetzen. Anstelle eines Strings verwendet C ein Array von `char`-Werten, dessen Ende durch das Zeichen `\0` (ASCII-Code 0) gekennzeichnet wird. Das Byte für diese Endmarkierung müssen Sie bei der Deklaration des `char`-Arrays mit einplanen: Ein `char[10]` ist ein String mit maximal neun nutzbaren Zeichen.

## Strukturen

Mitunter ist es nützlich, mehrere Werte verschiedener Datentypen »unter einem gemeinsamen Dach« zu verwalten. Zu diesem Zweck stellt C einen speziellen komplexen Datentyp namens `struct` bereit. In einer **Struktur** können sich beliebig viele Variablen verschiedener Datentypen befinden, was besonders nützlich ist, um komplexe Datenstrukturen zwischen Funktionen hin- und herzureichen.

Das folgende Beispiel definiert eine Struktur namens `person`, die verschiedene Daten über Personen verwaltet:

```
struct person {  
    char vorname[20];  
    char nachname[30];  
    int alter;  
};
```

Beachten Sie, dass eine `struct`-Definition, anders als andere Blöcke in geschweiften Klammern, mit einem Semikolon enden muss.

Eine Variable dieses Datentyps wird folgendermaßen deklariert:

```
struct person klaus;
```

Wenn Sie die einzelnen Elemente innerhalb einer Strukturvariablen ansprechen möchten, wird dafür die Form `variable.element` verwendet. Hier sehen Sie beispielsweise, wie die soeben definierte Variable `klaus` mit Inhalt versehen wird:

```
klaus.vorname = "Klaus";  
klaus.nachname = "Schmitz";  
klaus.alter = 42;
```

Oftmals werden Zeiger auf Strukturen als Funktionsargumente eingesetzt. Für die relativ unhandliche Schreibweise `(*strukturvariable).element`, die Sie verwenden müssten, um aus der Funktion heraus auf die Elemente einer Strukturvariablen zuzugreifen, wird die Kurzfassung `strukturvariable->element` definiert. Die folgende Funktion kann beispielsweise aufgerufen werden, um die angegebene Person ein Jahr älter zu machen:

```
void geburtstag(struct person *wer) {  
    wer->alter++;  
}
```

Der Aufruf dieser Funktion erfolgt beispielsweise so:

```
geburtstag(&klaus);
```

## Die C-Standardbibliothek

Wie Sie möglicherweise bemerkt haben, stehen viele Funktionen, die man von einer Programmiersprache erwartet, im bisher besprochenen Sprachkern von C nicht zur Verfügung. Vor allem die Ein- und Ausgabefunktionen sind nicht darin enthalten, weil die Ein- und Ausgabe sich je nach verwendeter Rechnerplattform erheblich unterscheidet. Diese Funktionen werden stattdessen

in externen Dateien zur Verfügung gestellt. Es handelt sich dabei um vorkompilierte Bibliotheksdateien, die vom Compiler mit dem eigenen Programmcode verknüpft werden. Die Schnittstellen der Bibliothek sind in sogenannten *Header-Dateien* definiert, die über die Präprozessordirektive `#include` eingebunden werden.

Die Laufzeitbibliothek der Programmiersprache C ist je nach Hardwareplattform und Betriebssystem unterschiedlich aufgebaut. Allerdings definiert der ANSI-Standard der Sprache eine Reihe vorgeschriebener Bibliotheksfunktionen, die von jeder beliebigen C-Implementierung unterstützt werden. Die Gesamtheit dieser standardisierten Funktionen wird als *C-Standardbibliothek* bezeichnet. Da so gut wie alle Betriebssysteme zu großen Teilen in C geschrieben sind, wird ihr Verhalten in erheblichem Maße von dieser Standardbibliothek beeinflusst.

Die Standardbibliothek besteht aus einer Reihe thematisch gegliederter Header-Dateien. Drei der wichtigsten werden im vorliegenden Abschnitt kurz vorgestellt.

#### **Ein- und Ausgabe: »stdio.h«**

In dieser wichtigsten aller Bibliotheksdateien, `stdio.h`, sind sämtliche Ein- und Ausgabefunktionen der Programmiersprache C zusammengefasst. Viele der Funktionen betreffen die Standardeingabe und Standardausgabe, also die Eingabe über die Tastatur und die Ausgabe auf der Konsole – falls sie nicht auf Dateien umgeleitet wurden. Andere Funktionen beschäftigen sich mit dem Öffnen, Lesen oder Schreiben von Dateien.

- `puts(String-Ausdruck)`  
Die Funktion `puts()` schreibt den Wert des angegebenen String-Ausdrucks auf die Standardausgabe, gefolgt von einem Zeilenumbruch.
- `printf(Format, Wert1, Wert2, ...)`  
Auch diese Funktion dient der Ausgabe von Inhalten auf die Konsole. Das erste Argument ist ein String mit Formatplatzhaltern, die für die anschließend aufgelisteten Werte stehen. Die wichtigsten Formatplatzhalter sind `%s` für einen String, `%d` für einen Integer und `%f` für Fließkommawerte.
- `sprintf(String-Variable, Format, Wert1, Wert2, ...)`  
`sprintf()` funktioniert genau so wie `printf()` – allerdings wird der formatierte String nicht ausgegeben, sondern in der als erstes Argument übergebenen String-Variablen gespeichert.
- `scanf(Format, Adresse)`  
`scanf()` dient der Eingabe eines Werts über die Standardeingabe (meist Tastatur); der eingegebene Wert wird unter der angegebenen Speicheradresse abgelegt. Die Adresse wird in der Regel durch Dereferenzierung einer Variablen (vorangestelltes `&`) angegeben, um die Eingabe in der entsprechenden Variablen zu speichern. Die Formatangabe besteht normalerweise nur aus einem einzelnen Formatplatzhalter (siehe `printf()`).
- `gets(Variable)`  
Mithilfe von `gets()` wird ein String von der Standardeingabe gelesen und in der angegebenen Variablen gespeichert. Aus Sicherheitsgründen, die bereits im Rahmen des Einführungsbeispiels



diskutiert wurden, sollte statt `gets()` besser `fgets()` mit dem speziellen Dateihandle `stdin` zum Einsatz kommen.

- `getchar()`  
Diese Funktion liest ein einzelnes Zeichen von der Standardeingabe. Beachten Sie, dass die Eingabe mit den Mitteln der Standardbibliothek dennoch immer zeilenorientiert verläuft: Sie können zwar in einer Schleife einzelne Zeichen einlesen, erhalten aber erst bei einem Zeilenende (wenn der Benutzer **Enter** drückt) ein Ergebnis. Echte zeichenorientierte Eingabe ist eine Angelegenheit plattformabhängiger Bibliotheken.
- `fopen(Dateiname, Modus)`  
Diese Funktion öffnet eine Datei auf einem Datenträger. Damit Sie auf diese Datei zugreifen können, wird das Funktionsergebnis von `fopen()` einer Variablen vom Typ `FILE` zugewiesen – der Wert ist ein eindeutiger Integer, der als **Dateideskriptor** oder **Dateihandle** bezeichnet wird. Der Dateiname kann ein beliebiger Pfad im lokalen Dateisystem sein. Beachten Sie unter Windows lediglich, dass das Pfadtrennzeichen `\` in einem C-String verdoppelt werden muss, weil es normalerweise Escape-Sequenzen wie `\n` einleitet. Der Modus kann unter anderem eines der Zeichen `"r"` (lesen), `"w"` (schreiben) oder `"a"` (anfügen) sein. Beispiel:  

```
fh = fopen ("test.txt", "r");  
/* test.txt zum Lesen öffnen */
```
- `fclose(Dateideskriptor)` schließt die angegebene Datei.
- `fputs(Deskriptor, String-Ausdruck)` schreibt den Wert des übergebenen String-Ausdrucks mit anschließendem Zeilenumbruch in die gewünschte Datei.
- `fprintf(Deskriptor, Format, Werte)` besitzt die gleiche Syntax wie `printf()`, schreibt aber in die angegebene Datei.
- `fscanf(Deskriptor, Format, Variable)` funktioniert wie `scanf()`, liest aber aus der angegebenen Datei.
- `fgets(Variable, Zeichenzahl, Deskriptor)` liest einen String aus der angegebenen Datei mit der entsprechenden maximalen Zeichenzahl oder bis zum ersten Zeilenumbruch.
- `fflush(Deskriptor)` leert den Puffer eines Eingabedatenstroms. Dies ist manchmal vor Eingabeoperationen wie `fscanf()` erforderlich, damit diese nicht »automatisch« aus der vorherigen Eingabe bedient werden. Für die Standardeingabe können Sie übrigens `fflush(stdin)` schreiben.

### **String-Funktionen: »string.h«**

Die Header-Datei `string.h` enthält verschiedene Funktionen zur Manipulation und Analyse von Strings in ihrer einfachsten Form, also `char`-Arrays. Zu den wichtigsten gehören folgende:

- `strcmp(String1, String2)` vergleicht die beiden angegebenen Strings miteinander. Das Ergebnis ist 0, wenn sie gleich sind, negativ, wenn `String1` alphanumerisch vor `String2` kommt,

und positiv, wenn es umgekehrt ist. Die Variante `strcmp()` unterscheidet nicht zwischen Groß- und Kleinschreibung.

- `strncmp(String1, String2, n)` und `strnicmp(String1, String2, n)` vergleichen nur die ersten `n` Zeichen der beiden Strings, wiederum mit beziehungsweise ohne Rücksicht auf Groß- und Kleinschreibung.
- `strcpy(String1, String2)` kopiert den Wert von `String2` an die Adresse von `String1`. Dies ist die einzige korrekte Methode, um einer String-Variablen den Wert einer anderen zuzuweisen!
- `strcat(String1, String2)` hängt den Wert von `String2` an das Ende von `String1` an.

### Überprüfen Sie stets alle String-Längen!

Sie müssen selbst vor jeder String-Operation die Länge der beteiligten Zeichenketten überprüfen, insbesondere bei Eingaben von Benutzern oder aus anderen unsicheren Quellen: Da diese Funktionen keinen integrierten Schutz vor Überläufen bieten, gehören sie zu den bevorzugten Angriffszielen für Cracker.

#### Datum und Uhrzeit: »time.h«

Die Header-Datei `time.h` definiert verschiedene Funktionen für die Arbeit mit Datum und Uhrzeit:

- `time(NULL)` fragt die aktuelle Systemzeit ab und liefert sie als Wert vom Typ `time_t` zurück. Als Argument in den Klammern wird eigentlich ein Zeiger auf `time_t` erwartet; da das Ergebnis aber bereits die Zeit enthält, wird in der Regel der spezielle Wert `NULL` (Zeiger auf gar nichts!) übergeben.
- `localtime(*Zeitangabe)` wandelt die Rückgabe von `time()` – die Sekunden seit EPOCH – in eine vorformatierte Ortszeit um. Das Argument ist ein Zeiger auf `time_t`, der Rückgabewert eine komplexe Struktur namens `struct tm`. Oft wird `localtime()` nur als »Zwischenwert« für `strftime()` verwendet.
- `strftime(String, Zeichenzahl, Format, *localtime-Wert)` formatiert die Zeitangabe nach der Vorschrift des angegebenen Formats und speichert das Ergebnis in der String-Variablen ab, die als erstes Argument vorliegt. Die Formatangaben entsprechen dem Unix-Befehl `date`. Das folgende Beispiel liest das aktuelle Datum und gibt es formatiert aus:

```
time_t jetzt;
char zeit[20];
...
jetzt = time(NULL);
strftime(zeit, 19, "%d.%m.%Y, %H:%M",
        localtime(&jetzt));
printf("Heute ist der %s.\n", zeit);
```

Die Ausgabe lautet beispielsweise folgendermaßen:

```
Heute ist der 01.05.2019, 14:47.
```

- `difftime (Zeitangabe1, Zeitangabe2)` gibt die Differenz zwischen zwei `time_t`-Zeitangaben in Sekunden an.

### Der Präprozessor

Formal hat der **Präprozessor** zwar nichts mit der Standardbibliothek zu tun, wird aber trotzdem hier kurz angeschnitten, weil er unter anderem für das Einbinden der Header-Dateien mithilfe von `#include` zuständig ist. Viele C-Programme bestehen aus mehr Präprozessordirektiven als aus gewöhnlichen Anweisungen, weil der Präprozessor die Definition bestimmter Abkürzungen ermöglicht.

Die wichtigste Präprozessordirektive haben Sie bereits kennengelernt: `#include` bindet eine Header-Datei ein, die Schnittstellendefinition einer Bibliothekskomponente. Sie können auch eigene häufig genutzte Funktionen in selbst geschriebene Header-Dateien auslagern, müssen dabei aber Folgendes beachten: `#include <Datei>` sucht in den standardisierten Include-Verzeichnissen Ihres Compilers oder Betriebssystems nach der angegebenen Header-Datei. Wenn Sie auf eine Datei im Verzeichnis des C-Programms selbst verweisen möchten, müssen Sie stattdessen die Schreibweise `#include "Datei"` verwenden.

### Eigene Header-Dateien schreiben

Wenn Sie eigene Header-Dateien schreiben möchten, um Programmfunktionalität auszulagern, beachten Sie Folgendes: In der Header-Datei mit der Endung `.h` werden Funktionen nur deklariert, es werden also nur ihre Köpfe hineingeschrieben und mit einem Semikolon abgeschlossen. Die Implementierung erfolgt dagegen in einer ansonsten gleichnamigen Datei mit der Endung `.c`.

Angenommen, in einer Datei namens `func.h` stünde folgende Deklaration:

```
int is_prime(int number);
```

Die Funktion soll überprüfen, ob das Argument `number` eine Primzahl ist oder nicht, und entsprechend 1 für wahr beziehungsweise 0 für falsch zurückgeben. Eine mathematisch primitive Implementierung könnte so aussehen und in der Implementierungsdatei `func.c` stehen:

```
int is_prime(int number) {
    if (number == 0 || number == 1) {
        return 0;
    }
    if (number == 2) {
        return 1;
    }
    for (int i = 2; i <= number / 2; i++) {
        if (number % i == 0) {
            return 0;
        }
    }
}
```

```

    return 1;
}

```

In der Programmdatei, beispielsweise `main.c`, können Sie dann Ihre Header-Datei einbinden und die Funktion aufrufen. Hier eine sehr kurze Beispielimplementierung, die überprüft, ob ein vom Benutzer eingegebener Integer eine Primzahl ist oder nicht:

```

#include <stdio.h>
#include "func.h"

int main(int argc, char* argv[]) {
    int input;
    printf("Ganze Zahl: ");
    scanf("%d", &input);
    if (is_prime(input)) {
        printf("%d ist eine Primzahl.\n", input);
    } else {
        printf("%d ist keine Primzahl.\n", input);
    }
    return 0;
}

```

Beim Kompilieren müssen Sie die beiden `.c`-Dateien angeben. Beispiel:

```
$ gcc -o main main.c func.c
```

Eine weitere wichtige Funktion des Präprozessors ist die Definition symbolischer Konstanten mithilfe der Direktive `#define`. Diese werden vor allem verwendet, um konstante Werte tief im Inneren des Programms zu vermeiden, wo sie sich später nur schwer auffinden und ändern lassen. Angenommen, Sie möchten in Ihrem Programm den Umrechnungsfaktor von DM nach € verwenden<sup>2</sup>, dann können Sie ihn folgendermaßen als symbolische Konstante festlegen:

```
#define DM 1.95583
```

In Ihrem Programm wird nun jedes Vorkommen von DM noch vor der eigentlichen Kompilierung durch 1.95583 ersetzt – außer innerhalb der Anführungszeichen von String-Literalen. Beachten Sie, dass am Ende einer Konstantendefinition kein Semikolon stehen darf.

Diese Fähigkeit des Präprozessors wird auch oft zur bedingten Kompilierung eingesetzt: Die Direktive `#ifdef` fragt ab, ob die angegebene symbolische Konstante existiert, und kompiliert nur in diesem Fall alle Zeilen bis zum Auftreten von `#endif`. Dies wird zum Beispiel zur Unterscheidung verschiedener Rechnerplattformen verwendet. Im folgenden Beispiel wird eine zusätzliche Anweisung mitkompiliert, falls eine symbolische Konstante namens `DEBUG` definiert ist:

---

<sup>2</sup> Sicherlich denken Sie, dieses Beispiel sei nach über 17 Jahren Euro-Bargeld irrelevant. Aber auch im Weihnachtsgeschäft 2018 bekam die Kundschaft einer großen Bekleidungskette wieder die Möglichkeit, mit eventuell vorhandenen DM-Rechtsbeständen zu zahlen.

```
#ifdef DEBUG
printf("Debug-Modus aktiviert.\n");
#endif
```

Die gegenteilige Variante `#ifndef` (nicht definiert) wird häufig verwendet, um zu überprüfen, ob eine bestimmte Header-Datei bereits irgendwo in den Programmdateien eines Projekts eingebunden wurde. Dazu wird der gesamte Deklarationsblock von `#ifndef` und `#endif` umschlossen, und die Konstante wird darin – oft ohne konkreten Wert – definiert. Eine solche Version der eben beschriebenen Header-Datei **func.h** sähe wie folgt aus:

```
#ifndef FUNCTIONS

#define FUNCTIONS

int is_prime(int number);

#endif
```