Grundkurs C

Release 0.2.0d

Aktualisiert am 02.12.2018

Bernhard Grotz

http://www.grund-wissen.de

Die Quelldateien dieses Buchs wurden unter Linux mittels Vim und Sphinx, die enthaltenen Graphiken mittels Inkscape erstellt. Der Quellcode sowie die Original-Graphiken können über die Projektseite heruntergeladen werden:

http://www.grund-wissen.de

Bei Fragen, Anmerkungen und Verbesserungsvorschlägen bittet der Autor um eine kurze Email an folgende Adresse:

info@grund-wissen.de

Augsburg, den 2. Dezember 2018.

Bernhard Grotz

Inhaltsverzeichnis

Einführung: Editieren und Übersetzen

Um ein lauffähiges C-Programm zu erzeugen, muss zunächst mit einem Texteditor eine Quelltext-Datei angelegt und mit Code gefüllt werden. Anschließend wird ein Compiler gestartet, der den Quellcode in Maschinen-Code übersetzt und ein lauffähiges Programm erstellt.

Als klassisches Beispiel soll hierzu ein minimales Programm dienen, das lediglich `"Hallo, Welt!"` auf dem Bildschirm ausgibt. Hierzu wird mit einem Texteditor folgender Code in eine (neue) Datei `hallo.c` geschrieben:

```
// Datei: hallo.c                    /* 1. */

#include <stdio.h>                    /* 2. */

void main()                           /* 3. */
{
    printf("Hallo, Welt!\n");         /* 4. */
}
```

Das obige Programm enthält folgende Komponenten:

1. Eine mit `//` eingeleitete Zeile am Dateianfang stellt einen Kommentar dar. Sie wird beim Übersetzen durch den Compiler ignoriert und dient lediglich der besseren Lesbarkeit. Ebenso werden Textbereiche, die durch `/*` und `*/` begrenzt sind, als Kommentare für Erklärungen oder Hinweise genutzt.[1]

2. Mit der Anweisung `#include <stdio.h>` wird dem Compiler mitgeteilt, die Standard-Input-Output-Bibliothek `stdio.h` zu laden.[2] Diese von vielen C-Programmen genutzte „Sammlung“ an Quellcode stellt u.a. Funktionen für die Ausgabe von Text auf dem Bildschirm bereit.

3. Die Funktion `main()` startet das Hauptprogramm, das sich innerhalb der folgenden geschweiften Klammern befindet. Jedes C-Programm verfügt über eine derartige `main()`-Funktion.[3]

[1] In vielen Programmen werden ausschließlich Kommentare verwendet, die mit den Zeichenfolgen `/*` und `*/` begrenzt sind. Hierdurch wird eine Kompatibilität mit alten C-Compiler-Versionen sicher gestellt. Im obigen Tutorium wird hingegen – nach persönlichem Geschmack – die `//`-Variante für (einzeilige) Kommentare verwendet.

Zusätzliche Kommentare der Form `/* 1. */` dienen in diesem Tutorium als Marker, um im Text auf die jeweiligen Stellen im Quellcode eingehen zu können.

[2] Genauer gesagt gilt die Anweisumg dem Präprozessor, einem Teil des Compilers.

[3] Die Bezeichung `void` besagt lediglich, dass die Funktion keinen Rückgabe-Wert liefert, der anderweitig im Programm zu verwenden wäre.

4. Durch den Aufruf der Funktion `printf()` wird auf dem Bildschirm der in doppelten Hochkommata stehende Text ausgegeben. Die Zeichenfolge `\n` steht dabei als Zeichen für eine neue Zeile. Der Aufruf der Funktion muss, wie jede C-Anweisung, mit einem Strichpunkt `;` beendet werden.

Um die Datei in lauffähigen Maschinen-Code zu übersetzen, wechselt man in einer Shell in den Ordner der Quellcode-Datei und ruft den Compiler `gcc` auf:

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

Durch die Option `-o hallo` wird dabei die Output-Datei, d.h. das fertige Programm, mit `hallo` benannt. Ist der Compilier-Vorgang abgeschlossen, kann das neu geschriebene Programm im gleichen Ordner aufgerufen werden:

```
./hallo

# Ergebnis: Hallo, Welt!
```

Damit ist das erste C-Programm fertig gestellt. In den folgenden Abschnitten werden weitere Eigenschaften und Funktionen der Programmiersprache C erläutert sowie einige nützliche Werkzeuge und Programmiertechniken vorgestellt.

Definition von Variablen

Ein wesentlicher Vorteil eines Computer-Programms gegenüber einem Taschenrechner besteht darin, dass es (nahezu beliebig viele) Werte und Zeichen in entsprechenden Platzhaltern („Variablen“) speichern und verarbeiten kann.

Da ein Computer-Prozessor nur mit Maschinencode arbeiten kann, müssen intern sowohl Zahlen wie auch Text- und Sonderzeichen als Folgen von Nullen und Einsen dargestellt werden. Dies ist aus der Sichtweise eines Programmierers zunächst nur soweit von Bedeutung, als dass er wissen muss, dass ein und dieselbe Folge von Nullen und Einsen vom Computer wahlweise als Zeichen oder als Zahl interpretiert werden kann. Der Programmierer muss dem Computer somit mitteilen, wie der Inhalt einer Variable zu interpretieren ist.

Deklaration, Definition, Initialisierung

Um Variablen benutzen zu können, muss der Datentyp der Variablen (z.B. `int` für ganze Zahlen) dem Compiler mitgeteilt werden („Deklaration“). Muss dabei auch Speicherplatz reserviert werden (was meist der Fall ist, wenn sich die Deklaration nicht auf Variablen externer Code-Bibliotheken bezieht), so spricht man von einer Definition einer Variablen.

In C werden Variablen stets zu Beginn einer Datei oder zu Beginn eines neuen, durch geschweifte Klammern begrenzten Code-Blocks definiert. Sie sind im Programm gültig, bis die Datei beziehungsweise der jeweilige Code-Block abgearbeitet ist.[1]

Eine Definition von Variablen erfolgt nach folgendem Schema:

```
int n;
```

Es dürfen auch mehrere gleichartige Variablen auf einmal definiert werden; hierzu werden die einzelnen Namen der Variablen durch Kommata getrennt und die Definition mit einem abschließenden Strichpunkt beendet.

```
int x,y,z;
```

Wird einer Variablen bei der Definition auch gleich ein anfänglicher Inhalt („Initialwert“) zugewiesen, so spricht man auch von einer Initiation einer Variablen.[2]

[1] Die einzige Ausnahme bewirkt hierbei das Schlüsselwort *static*.

[2] Die Initialisierung, d.h. die erstmalige Zuweisung eines Werts an eine Variable, kann auch erst zu einem späteren Zeitpunkt erfolgen.

```
int c = 256;
```

In C wird das Ist-Gleich-Zeichen = als Zuweisungsoperator genutzt, der den Ausdruck auf der rechten Seite in die Variablen auf der linken Seite abspeichert.[3] Eine erneute Angabe des Datentyps einer Variablen würde beim Übersetzen sogar eine Fehlermeldung des Compilers zur Folge haben, da in diesem Fall von einer (versehentlichen) doppelten Vergabe eines Variablennamens ausgegangen wird.

Variablennamen dürfen in C maximal 31 Stellen lang sein. Sie können aus den Buchstaben `A-Z` und `a-z`, den Ziffern `0-9` und dem Unterstrich bestehen. Die einzige Einschränkung besteht darin, dass am Anfang von Variablennamen keine Ziffern stehen dürfen; Unterstriche am Anfang von Variablennamen sind zwar erlaubt, sollten aber vermieden werden, da diese üblicherweise für Bibliotheksfunktionen reserviert sind.

In C wird allgemein zwischen Groß- und Kleinschreibung unterschieden, beispielsweise bezeichnen `a` und `A` zwei unterschiedliche Variablen. Im Allgemeinen werden Variablen und Funktionen in C-Programmen fast immer klein geschrieben.

Ist einmal festgelegt, um welchen Datentyp es sich bei einer Variablen handelt, wird die Variable im Folgenden ohne Angabe des Datentyps verwendet.

Elementare Datentypen

Als grundlegende Datentypen wird in C zwischen folgenden Arten unterschieden:

Typ	Bedeutung	Speicherbedarf
char	Ein einzelnes Zeichen	1 Byte (= 8 Bit)
int	Eine ganzzahlige Zahl	4 Byte (= 32 Bit)
short	Eine ganzzahlige Zahl	2 Byte (= 16 Bit)
long	Eine ganzzahlige Zahl	8 Byte (= 64 Bit)
float	Eine Fließkomma-Zahl	4 Byte (= 32 Bit)
double	Eine Fließkomma-Zahl	8 Byte (= 64 Bit)

Der Speicherbedarf der einzelnen Datentypen hängt von der konkreten Rechnerarchitektur ab; in der obigen Tabelle sind die Werte für 32-Bit-Systeme angegeben, die für Monocore-Prozessoren üblich sind. Auf anderen Systemen können sich andere Werte für die einzelnen Datentypen ergeben. Die Größe der Datentypen auf dem gerade verwendeten Rechner kann mittels des *sizeof*-Operators geprüft werden:

```
// Datei: sizeof.c

#include <stdio.h>

void main()
```

(continues on next page)

[3] Der Wertevergleich, wie er in der Mathematik durch das Ist-Gleich-Zeichen ausgedrückt wird, erfolgt in C durch den Operator ==.

(Fortsetzung der vorherigen Seite)

```
{
    printf("Size of char:   %lu\n", sizeof (char)  );
    printf("Size of int:    %lu\n", sizeof (int)   );
    printf("Size of short:  %lu\n", sizeof (short) );
    printf("Size of long:   %lu\n", sizeof (long)  );
    printf("Size of float:  %lu\n", sizeof (float) );
    printf("Size of double: %lu\n", sizeof (double));
}
```

In diesem Beispiel-Programm werden nach dem Compilieren mittels `gcc -o sizeof sizeof.c` und einem Aufruf von `./sizeof` die Größen der einzelnen Datentypen in Bytes ausgegeben. Hierzu wird bei der Funktion `printf()` das Umwandlungszeichen `%lu` verwendet, das durch den Rückgabewert von `sizeof` (entspricht `long integer`) ersetzt wird.

Einen „Booleschen“ Datentyp, der die Wahrheitswerte `True` oder `False` repräsentiert, existiert in C nicht. Stattdessen wird der Wert Null für `False` und jeder von Null verschiedene Wert als `True` interpretiert.

Komplexere Datentypen lassen sich aus diesen elementaren Datentypen durch Aneinanderreihungen (*Felder*) oder Definitionen von Strukturen (struct) erzeugen. Zusätzlich existiert in C ein Datentyp namens `void`, der null Bytes groß ist und beispielsweise dann genutzt wird, wenn eine Funktion *keinen* Wert als Rückgabe liefert.

Modifier

Alle grundlegenden Datentypen (außer `void`) können zusätzlich mit einem der folgenden „Modifier“ versehen werden:

- `signed` bzw. `unsigned`:

  Ohne explizite Angabe dieses Modifiers werden Variablen üblicherweise als `signed`, d.h. mit einem Vorzeichen versehen, interpretiert. Beispielsweise lassen sich durch eine 1 Byte (8 Bit) große Variable vom Typ `signed char` Werte von `-128` bis `+128` abbilden, durch eine Variable vom Typ `unsigned char` Werte von `0` bis `255`. Diese Werte werden dann üblicherweise als ASCII-Codes interpretiert.

- `extern`:

  Dieser Modifier ist bei der Deklaration einer Variablen nötig, wenn diese bereits in einer anderen Quellcode-Datei definiert wurde. Für externe Variablen wird kein neuer Speicherplatz reserviert. Gleichzeitig wird durch den `extern`-Modifier dem Compiler mitgeteilt, in den zu Beginn eingebundenen Header-Dateien nach einer Variablen dieses Namens zu suchen und den dort reservierten Speicherplatz gemeinsam zu nutzen.

- `static`:

Eine Variable ist üblicherweise nur innerhalb des jeweiligen durch geschweifte Klammern begrenzten Codeblocks gültig, innerhalb dessen sie definiert wurde.

Wird eine Variable, beispielsweise als Zählvariable, innerhalb einer *Funktion* definiert, so wird ihre Speicherstelle gelöscht, sobald der Aufruf der Funktion beendet ist. Wird bei der Definition einer solchen „lokalen“ Variable jedoch der Modifier `static` verwendet, so liegt ihr Wert auch beim nächsten Aufruf der gleichen Funktion unverändert vor.

Auch Variablen, die gleich zu Beginn einer Datei definiert werden, können mit dem Modifier `static` versehen werden. Auf eine solche Variable können dann alle Funktionen dieser Datei zugreifen, für Funktionen anderer Dateien ist sie hingegen nicht sichtbar.

Umgekehrt ist jede Funktion und jede außerhalb einer Funktion definierte Variable „global“, wenn sie nicht mit `static` versehen wurde. Globale Variablen sollten, sofern möglich, vermieden werden, da sie von vielen Stellen aus manipuliert werden können und im Zweifelsfall die Fehler verursachende Stelle im Code nur schwer gefunden wird.

- `auto` bzw. `register`:

  Ohne explizite Angabe dieses Modifiers werden Variablen üblicherweise als `auto` interpretiert; diese Angabe wird automatisch vom Compiler ergänzt und daher grundsätzlich weggelassen. Wird eine Variable hingegen mit dem Modifier `register` versehen, so ist dies eine Empfehlung des Programmierers an den Compiler, diese Variable nicht im (externen) Arbeitsspeicher, sondern im Prozessorspeicher abzulegen. Dadurch kann in seltenen Fällen ein schnellerer Zugriff auf die Variable erreicht werden. Da der Prozessorspeicher jedoch meist sehr begrenzt ist, wird der `register`-Modifier nur selten (und ausschließlich fuer numerische Variablen) eingesetzt und vom Compiler gegebenenfalls als `auto` umgeschrieben.

- `const`:

  Mit `const` können Variablen bezeichnet werden, auf die nur lesend zugegriffen werden sollte. Schreibzugriffe auf solche Konstanten sind zwar möglich, sollten jedoch vermieden werden, da das Ergebnis undefiniert ist. Das Schlüsselwort `const` wird somit zur besseren Lesbarkeit verwendet und erlaubt es dem Compiler, gewisse Optimierungen vorzunehmen.

  Neben dem Schlüsselwort `const` können Konstanten ebenfalls mittels der Präprozessor-Direktive `define` festgelegt werden.

  Bei einzelnen ASCII-Zeichen, also beispielsweise Buchstaben von `'a'` bis `z` beziehungsweise `'A'` bis `'Z'` sowie Sonderzeichen und Umlauten handelt es sich in C ebenfalls um Konstanten.

- `volatile`

  Es gibt Variablen, die sich ändern können, ohne dass der Compiler dies vermuten würde. Üblicherweise werden solche Variablen vom Compiler aus Optimierungsgründen durch eine Konstante ersetzt und nicht stets

erneut eingelesen. Mit dem Schlüsselwort `volatile` hingegen zwingt man den Compiler, den Wert dieser Variablen bei jeder Benutzung erneut aus dem Speicher zu lesen und mehrfaches Lesen nicht weg zu optimieren. Das ist beispielsweise wichtig bei Variablen, die Zustände von Hardwarekomponenten anzeigen, oder bei Variablen, die durch Interrupt-Routinen verändert werden.

Beispiel:

```
volatile int Tastenzustand;

Tastenzustand = 0;
while (Tastenzustand == 0)
{
    // Warten auf Tastendruck
}
```

Ohne das Schlüsselwort `volatile` könnte der Compiler im obigen Beispiel eine Endlosschleife erzeugen, da er nicht wissen kann, dass sich der Zustand Tastenzustand während der Schleife ändern kann.

Zeiger und Felder

In vielen Fällen ist es nützlich, Variablen nicht direkt anzusprechen, sondern anstatt dessen so genannte Zeiger („Pointer“) zu nützen. Bei einem solchen Zeiger handelt es sich um eine eigenständige Variable, deren Inhalt die Speicheradresse einer anderen Variablen ist.

Zeiger

Bei der Definition eines Zeigers wird festgelegt, für welchen Datentyp der Zeiger vorgesehen ist. Die Definition eines Zeigers ähnelt dabei weitgehend der einer normalen Variablen, mit dem Unterschied, dass zur eindeutigen Kennzeichnung vor den Namen der Zeigervariablen ein `*` geschrieben wird:

```
int *n;
```

Es dürfen wiederum mehrere Zeiger auf einmal definiert werden; hierzu werden die einzelnen Namen der Zeigervariablen durch Kommata getrennt und die Definition mit einem abschließenden Strichpunkt beendet.

```
int *x, *y, *z;
```

Der Adress-Operator `&`

Um einer Zeigervariablen einen Inhalt, d.h. die eine gültige Speicheradresse zuzuweisen, wird der so genannte Adress-Operator `&` verwendet. Wird dieser Operator vor eine beliebige Variable geschrieben, so gibt er die zugehörige Speicheradresse aus. Diese kann wie gewöhnlich in der Variablen auf der linken Seite des =-Zeichens gespeichert werden:

```
int num = 256;
int *p_num;

p_num = &num;
```

In diesem Beispiel ist `p_num` ein Zeiger auf eine Integer-Variable, hat also selbst den Datentyp `int *`. Entsprechend gibt es auch Zeiger auf die anderen Datentypen, beispielsweise `float *`, `char *` usw.[1]

Ein Zeiger, dem noch keine Speicheradresse zugewiesen würde oder der auf eine ungültige Speicheradresse zeigt, bekommt in C automatisch den Wert `NULL` zugewiesen.[2]

Der Inhalts-Operator *

Möchte man den Zeiger wiederum dazu nutzen, um auf den Inhalt der Speicheradresse zuzugreifen, kann der sogenannte Inhalts-Operator * verwendet werden. Angewendet auf eine bereits deklarierte Variable gibt dieser den zur Speicheradresse gehörigen Inhalt aus.

Erzeugt man beispielsweise einen Zeiger `b`, der auf eine Variable `a` zeigt, so ist `*b` identisch mit dem Wert von `a`:

```
int a;
int *b;

a = 15;
b = &a;

printf("Die Adresse von a ist %u!\n" ,  b);
printf("Der Wert    von a ist %i!\n" , *b);
```

Das Symbol * hat in C somit zwei grundlegend verschiedene Verwendungsarten. Einerseits ist es nötig um bei der Deklaration Zeigervariablen von normalen Variablen zu unterscheiden. Im eigentlichen Programm bezeichnet * andererseits einen Operator, der es ermöglicht den Inhalt der in der Zeigervariablen abgelegten Speicherstelle abzufragen.

Der *-Operator kann auch für Wertzuweisungen, also auf der linken Seite des Istgleich-Zeichens benutzt werden. Hierbei muss der Programmierer allerdings unbedingt darauf achten, dass der jeweilige Zeiger bereits initiiert (nicht `NULL`) ist, sondern auf eine gültige Speicherstelle zeigt:

```
int a;
int *b;

// Zeiger NIEMALS ohne Initialisierung
// auf die linke Seite schreiben:
```

(continues on next page)

[1] Es gibt auch `void *`-Zeiger, die auf keinen bestimmten Datentyp zeigen. Solche Zeiger werden beispielsweise von der Funktion `malloc()` bei einer *dynamischen Reservierung von Speicherplatz* als Ergebnis zurückgegeben. Der Programmierer muss in diesem Fall dem Zeiger selbst den gewünschten Datentyp zuweisen.

[2] Der Grund für die Verwendung eines `NULL`-Zeigers (einer in der Datei `stddef.h` definierten Konstanten mit dem Wert 0) liegt darin, dass eine binär dargestellte Null in C niemals als Speicheradresse verwendet wird.

Manchmal wird der `NULL`-Pointer von *Funktionen*, die gewöhnlich einen bestimmten Zeiger als Ergebnis liefern, zur Anzeige einer erfolglosen Aktion verwendet. Liegt kein Fehler vor, so ist der Rückgabewert die Adresse eines Speicherobjektes und somit von `0` verschieden.

(Fortsetzung der vorherigen Seite)

```
// *b = 15;              // Fataler Fehler, Speicheradresse nicht bekannt!
// !!!

// Zeiger IMMER erst initialisieren:
b  = &a;                 // Der Zeiger zeigt jetzt auf die Adresse von a
*b = 15;                 // Zuweisung in Ordnung!
```

Wäre der Zeiger auf der linken Seite gleich `NULL`, so würde die Wertzuweisung an eine undefinierte Stelle erfolgen; im schlimmsten Fall würde eine andere für das Programm wichtige Speicheradresse überschrieben werden. Ein solcher Fehler kann vom Compiler nicht erkannt werden, kann aber mit großer Wahrscheinlichkeit ein abnormales Verhalten des Programms oder einen Absturz zur Folge haben.

Felder

Als Feld („Array“) bezeichnet man eine Zusammenfassung von mehreren Variablen gleichen Datentyps zu einem gemeinsamen Speicherbereich.

Bei der Definition eines Arrays muss einerseits der im Array zu speichernde Datentyp angegeben werden, andererseits wird zusätzlich in eckigen Klammern die Größe des Arrays angegeben. Damit ist festgelegt, wie viele Elemente in dem Array maximal gespeichert werden können.[3] Die Syntax lautet somit beispielsweise:

```
int numbers[10];

// Definition und Zuweisung zugleich:
int other_numbers[5] = { 10, 11, 12, 13, 14 };
```

Wird ein Array bei der Definition gleich mit einem konkreten Inhalt initialisiert, so kann die explizite Größenangabe entfallen und anstelle dessen ein leeres Klammerpaar `[]` gesetzt werden.

Der Hauptvorteil bei der Verwendung von Arrays liegt darin, eine Vielzahl gleichartiger Datei über eine einzige Variable (den Namen des Arrays) ansprechen zu können. Auf die einzelnen Elemente eines Feldes kann nach im eigentlichen Programm mittels des so genannten Selektionsoperators `[]` zugegriffen werden. Zwischen die eckigen Klammern wird dabei ein (ganzzahliger) Laufindex `i` geschrieben.

Hat ein Array insgesamt `n` Elemente, so kann der Laufindex `i` alle ganzzahligen Werte zwischen `0` und `n-1` annehmen. Das erste Element hat also den Index `0`, das zweite den Index `1`, das letzte schließlich den Index `n-1`. Somit kann der Inhalt jeder im Array gespeicherten Variablen ausgelesen oder durch einen anderen ersetzt werden:

[3] Die Größe von Feldern kann nach der Deklaration nicht mehr verändert werden. Somit muss das Feld ausreichend groß gewählt werden, um alle zu erwartenden Werte speichern zu können. Andererseits sollte es nicht unnötig groß gewählt werden, da ansonsten auch unnötig viel Arbeitsspeicher reserviert wird.

Soll die Größe eines Feldes erst zur Laufzeit festgelegt werden, so müssen die Funktionen `malloc()` bzw. `calloc()` verwendet werden.

```c
int numbers[5];

numbers[0] =  3;
numbers[1] =  5;
numbers[2] =  8;
numbers[3] = 13;
numbers[4] = 21;

printf("Die vierte Nummer des Feldes 'num' ist %i.\n", numbers[3]);
```

Eine Besonderheit von Arrays in C ist es, dass der Compiler beim Übersetzen nicht prüft, ob bei der Verwendung eines Laufindex die Feldgrenzen eingehalten werden. Im Fall eines Arrays `numbers` mit fünf Elementen könnte beispielsweise mit `numbers[5] = 1` ein Eintrag in einen Speicherbereich geschrieben werden, der außerhalb des Arrays liegt. Auf korrekte Indizes muss somit der Programmierer achten, um Programmfehler zu vermeiden.

Mehrdimensionale Felder

Ein Array kann wiederum Arrays als Elemente beinhalten. Beispielsweise kann man sich eine Tabelle aus einer Vielzahl von Zeilen zusammengesetzt denken, die ihrerseits wiederum eine Vielzahl von Spalten bestehen können. Beispielsweise könnte ein solches Tabellen-Array, das als Einträge jeweils Zahlen erwartet, folgendermaßen deklariert werden:[4]

```c
// Tabelle mit 3 Zeilen und je 4 Spalten deklarieren:
int zahlentabelle[3][4];
```

Auch in diesem Fall laufen die Indexwerte bei n Einträgen nicht von 1 bis n, sondern von 0 bis $n-1$. Der erste Auswahloperator greift ein Zeilenelement heraus, der zweite eine bestimmte Spalte der ausgewählten Zeile. Auch eine weitere Verschachtelung von Arrays nach dem gleichen Prinzip ist möglich, wobei der Zugriff auf die einzelnen Werte meist über *for*-Schleifen erfolgt.

Zeiger auf Felder

In C sind Felder und Zeiger eng miteinander verwandt: Gibt man den Namen einer Array-Variablen ohne eckige Klammern an, so entspricht dies einem Zeiger auf die erste Speicheradresse, die vom Array belegt wird; nach der Deklaration `int numbers[10];` kann also beispielsweise als abkürzende Schreibweise für das erste Element des Feldes anstelle von `&numbers[0]` auch die Kurzform `numbers` benutzt werden.[5]

[4] Eine direkte Initialisierung eines mehrdimensionalen Arrays ist ebenfalls unmittelbar möglich; dabei werden die einzelnen „Zeilen“ für eine bessere Lesbarkeit in geschweifte Klammern gesetzt. Beispielsweise kann gleich bei der Definition `int zahlentabelle[3][4] = { {3,4,1,5}, {8,5,6,9}, {4,7,0,3} };` geschrieben werden.

[5] Legt man bei der Deklaration eines Feldes seine Groesse nicht fest, um diese erst zur Laufzeit mittels *malloc()* zu reservieren, so kann bei der Deklaration anstelle von `int numbers[];` ebenso `int *numbers;` geschrieben werden.

Da alle Elemente eines Arrays den gleichen Datentyp haben und somit gleich viel Speicherplatz belegen, unterscheiden sich die einzelnen Speicheradressen der Elemente um die Länge des Datentyps, beispielsweise um `sizeof (int)` für ein Array mit `int`-Werten oder `sizeof (float)` für ein Array mit `float`-Werten. Ausgehend vom ersten Element eines Arrays erhält man somit die weiteren Elemente des Feldes, indem man den Wert des Zeigers um das $1, 2, \ldots, n-1$-fache der Länge des Datentyps erhöht:

```
int numbers[10];
int *numpointer;

// Pointer auf erstes Element des Arrays:
numpointer = &numbers;                          // oder: &numbers[0]

// Pointer auf zweites Element des Arrays:
numpointer = &numbers + sizeof (int);           // oder: &numbers[1]

// Pointer auf drittes Element des Arrays:
numpointer = &numbers + 2 * sizeof (int);       // oder: &numbers[2]
```

Beim Durchlaufen eines Arrays ist eine Erhöhung des Zeigers in obiger Form auch mit dem *Inkrement-Operator* möglich: Es kann also auch `numpointer++` statt `numpointer = numpointer + sizeof (int)` geschrieben werden, um den Zeiger auf das jeweils nächste Element des Feldes zu bewegen; dies wird beispielsweise in *for*-Schleifen genutzt. Ebenso kann das Feld mittels `numpointer--` schrittweise rückwärts durchlaufen werden; auf das Einhalten der Feldgrenzen muss der Programmierer wiederum selbst achten.

Da es sich bei Speicheradressen um `unsigned int`-Werte handelt, können zwei Zeiger auch ihrer Größe nach verglichen werden. Hat man beispielsweise zwei Pointer `numpointer_1` und `numpointer_2`, die beide auf ein Elemente eines Arrays zeigen, so würde `numpointer_1 < numpointer_2` bedeuten, dass der erste Pointer auf ein Element zeigt, das sich weiter vorne im Array befindet. Ebenso kann in diesem Fall mittels `numpointer_2 - numpointer_1` die Anzahl der Elemente bestimmt werden, die zwischen den beiden Pointern liegen.

Andere mathematische Operationen sollten auf Zeiger nicht angewendet werden; ebenso sollten Array-Variablen, obwohl sie letztlich einen Zeiger auf das erste Element des Feldes darstellen, niemals direkt inkrementiert oder dekrementiert werden, da das Array eine feste Stelle im Speicher einnimmt. Stattdessen definiert man stets einen Zeiger auf das erste Element des Feldes und inkrementiert diesen, um beispielsweise in einer Schleife auf die einzelnen Elemente eines Feldes zuzugreifen.

Zeichenketten

Zeichenketten („Strings“), beispielsweise Worte und Sätze, stellen die wohl häufigste Form von Arrays dar. Eine Zeichenkette besteht aus einer Aneinanderreihung einzelner Zeichen (Datentyp `char`) und wird stets mit einer binären Null (`'\0'`) abgeschlossen. Beispielsweise entspricht die Zeichenkette `"Hallo!"` einem Array, das aus `'H'`, `'a'`, `'l'`, `'l'`, `'o'`, `'!'` und dem Zeichen `'\0'` besteht. Dieser Unterschied besteht allgemein zwischen Zei-

chenketten, die mit doppelten Hochkommatas geschrieben werden, und einzelnen Zeichen, die in einfachen Hochkommatas dargestellt werden.

Die Deklaration einer Zeichenkette entspricht der Deklaration eines gewöhnlichen Feldes:

```
// Deklaration ohne Initialisierung:
char string_one[15];

// Deklaration mit Initialisierung:
char string_two[] = "Hallo Welt!"
```

Bei der Festlegung der maximalen Länge der Zeichenkette muss beachtet werden, dass neben den zu speichernden Zeichen auch Platz für das String-Ende-Zeichen `'\0'` bleiben muss. Als Programmierer muss man hierbei selbst darauf achten, dass die Feldgröße ausreichend groß gewählt wird.

Wird einer String-Variablen nicht bereits bei der Deklaration eine Zeichenkette zugewiesen, so ist dies anschliessend zeichenweise (beispielsweise mittels einer *Schleife*) möglich:

```
string_one[0] = 'H';
string_one[1] = 'a';
string_one[2] = 'l';
string_one[3] = 'l';
string_one[4] = 'o';
string_one[5] = '!';
string_one[6] = '\0';
```

Eine Zuweisung eines ganzen Strings an eine String-Variable in Form von `string_one = "Hallo!"` ist nicht direkt möglich, sondern muss über die Funktion *strcpy()* aus der Standard-Bibliothek *string.h* erfolgen:

```
// Am Dateianfang:
#include <string.h>

// ...

// String-Variable deklarieren:
char string_one[15];

// Zeichenkette in String-Variable kopieren:
strcpy(string_one, "Hallo Welt!");

// Zeichenkette ausgeben:
printf("%s\n", string_one);
```

Anstelle der Funktion `strcpy()` kann auch die Funktion `strncpy()` verwendet werden, die nach der zu kopierenden Zeichenkette noch einen `int`-Wert n erwartet; diese Funktion kopiert maximal n Zeichen in die Zielvariable, womit ein Überschreiten der Feldgrenzen ausgeschlossen werden kann.

ASCII-Codes und Sonderzeichen

Die einzelnen Zeichen (Datentyp `char`) werden vom Computer intern ebenfalls als ganzzahlige Werte ohne Vorzeichen behandelt. Am weitesten verbreitet ist die so genannte ASCII-Codierung („American Standard Code for Information Interchange“), deren Zuweisungen in der folgenden *ASCII-Tabelle* abgebildet sind. Wird beispielsweise nach der Deklarierung `char c;` der Variablen `c` mittels `c = 120` ein numerischer Wert zugewiesen, so liefert die Ausgabe von `printf("%c\n", c);` den zur Zahl `120` gehörenden ACII-Code, also `x`.

Dez	ASCII	Dez	ASCII	Dez	ASCII	Dez	ASCII	Dez	ASCII	Dez	ASCII	Dez	ASCII	Dez	ASCII
0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	‘	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Die zu den Zahlen `0` bis `127` gehörenden Zeichen sind bei fast allen Zeichensätzen identisch. Da der ASCII-Zeichensatz allerdings auf die englische Sprache ausgerichtet ist und damit keine Unterstützung für Zeichen anderer Sprachen beinhaltet, gibt es Erweiterungen des ASCII-Zeichensatzes für die jeweiligen Länder.

Neben den Obigen ASCII-Zeichen können Zeichenketten auch so genannte „Escape-Sequenzen“ als Sonderzeichen beinhalten. Der Name kommt daher, dass zur Darstellung dieser Zeichen ein Backslash-Zeichen \ erforderlich ist, das die eigentliche Bedeutung des darauf folgenden Zeichens aufhebt. Einige wichtige dieser Sonderzeichen sind in der folgenden Tabelle aufgelistet.

Zeichen	Bedeutung
`\n`	Zeilenwechsel („new line“)
`\t`	Tabulator (entspricht üblicherweise 4 Leerzeichen)
`\b`	Backspace
`\\`	Backslash-Zeichen
`\"`	Doppeltes Anführungszeichen
`\'`	Einfaches Anführungszeichen

Eine weitere Escape-Sequenz ist das Zeichen `'\0'` als Endmarkierung einer Zeichenkette, das verständlicherweise jedoch nicht innerhalb einer Zeichenketten stehen darf.

Ausgabe und Eingabe

Das Ausgeben und Einlesen von Daten über den Bildschirm erfolgt häufig mittels der Funktionen `printf()` und `scanf()`.[1] Beide Funktionen sind Teil der *Standard-Bibliothek stdio.h*, so dass diese zu Beginn der Quellcode-Datei mittels `include <stdio.h>` eingebunden werden muss.[2]

`printf()` – Daten formatiert ausgeben

Die Funktion `printf()` dient grundsätzlich zur direkten Ausgabe von Zeichenketten auf dem Bildschirm; beispielsweise gibt `printf("Hallo Welt!")` die angegebene Zeichenkette auf dem Bildschirm aus. Innerhalb der Zeichenketten können allerdings Sonderzeichen sowie Platzhalter für beliebige Variablen und Werte eingefügt werden.

Zeichen	Bedeutung
`\n`	Neue Zeile
`\t`	Tabulator (4 Leerzeichen)
`\\`	Backslash-Zeichen \
`\'`	Einfaches Anführungszeichen
`\"`	Doppeltes Anführungszeichen

Die in der obigen Tabelle angegebenen Sonderzeichen werden auch „Escape-Sequenzen“ genannt, da sie nur mittels des vorangehenden Backslash-Zeichens, das ihre sonstige Bedeutung aufhebt, innerhalb einer Zeichenkette dargestellt werden können.

Ein Platzhalter besteht aus einem `%`-Zeichen, gefolgt von einem oder mehreren Zeichen, welche den Typ der auszugebenden Werte oder Variablen angeben und gleichzeitig festlegen, wie die Ausgabe formatiert werden soll. Damit kann beispielsweise bestimmt werden, wie viele Stellen für einen Wert reserviert werden sollen, ob die Ausgabe links- oder rechtsbündig erfolgen soll, und/oder ob bei der Ausgabe von Zahlen gegebenenfalls führende Nullen angefügt werden sollen.

[1] Um Daten von Dateien anstelle vom Bildschirm einzulesen, gibt es weitere Funktionen, die im Abschnitt *Dateien und Verzeichnisse* näher beschrieben sind.

[2] Genau genommen erfolgt bei der Funktion `printf()` die Ausgabe auf den Standard-Ausgang (`stdout`). Bei diesem handelt es sich als Voreinstellung um den Bildschirm, in speziellen Fällen kann jedoch mittels der Funktion *freopen()* auch eine beliebige Datei oder ein angeschlossenes Gerät als Standard-Ausgang festgelegt werden.

Ebenso liest die Funktion `scanf()` vom Standard-Eingang (`stdin`) ein, der als Voreinstellung wiederum dem Bildschirm entspricht.

```
// Den Wert Pi auf sechs Nachkommastellen genau ausgeben:

printf("Der Wert von Pi ist %.6f...\n", 3.141592653589793)
// Ergebnis: Der Wert von Pi ist 3.141593...

// Maximal dreistellige Zahlen rechtsbündig ausgeben:

printf("%3i:\n%3i:\n%3i:\n", 1, 10, 100);
// Ergebnis:
//   1:
//  10:
// 100:

// Maximal dreistellige Zahlen linksbündig ausgeben:

printf("%3i:\n%3i:\n%3i:\n", 1, 10, 100);
// Ergebnis:
// 1  :
// 10 :
// 100:

// Einstelligen Zahlen eine Null voranstellen:

printf("%02i.:\n%02i.:\n%02i.:\n", 8, 9, 10);
// Ergebnis:
// 08.:
// 09.:
// 10.:
```

In den obigen Beispielen wurden der Funktion `printf()` zwei oder mehr Argumente übergeben. Beim ersten Argument handelt es sich um einen so genannten Formatstring, bei den folgenden Argumenten um die auf dem Bildschirm auszugebenden Werte. Falls diese, wie im ersten Beispiel, mehr Nachkommastellen haben als in der Formatierung vorgesehen (Die Angabe `%.6f` steht für einen Wert vom Datentyp `float` sechs Nachkommastellen), so wird der Wert automatisch auf die angegebene Genauigkeit gerundet.

Zur Festlegung des Datentyps einer auszugebenden Variablen gibt es allgemein folgende Umwandlungszeichen:

Zeichen	Argument	Bedeutung
`d, i`	`int`	Dezimal-Zahl mit Vorzeichen.
`o`	`int`	Oktal-Zahl ohne Vorzeichen (und ohne führende Null).
`x, X`	`int`	Hexadezimal-Zahl ohne Vorzeichen (und ohne führendes `0x` oder `0X`), also `abcdef` bei `0x` oder `ABCDEF` bei `0X`.
`u`	`int`	Dezimal-Zahl ohne Vorzeichen.
`c`	`int`	Ein einzelnes Zeichen (`unsigned char`).
`s`	`char *`	Zeichen einer Zeichenkette bis zum Zeichen `\0`, oder bis zur angegebenen Genauigkeit.
`f`	`double`	Dezimal-Zahl als `[-]mmm.ddd`, wobei die angegebene Genauigkeit die Anzahl der `d` festlegt. Die Voreinstellung ist `6`, bei `0` entfällt der Dezimalpunkt.
`e, E`	`double`	Dezimal-Zahl als `[-]m.dddddde±xx` oder `[-]m.ddddddE±xx`, wobei die angegebene Genauigkeit die Anzahl der `d` festlegt. Die Voreinstellung ist `6`, bei `0` entfällt der Dezimalpunkt.
`g, G`	`double`	Dezimal-Zahl wie wie `%e` oder `%E`. Wird verwendet, wenn der Exponent kleiner als die angegebene Genauigkeit ist; unnötige Nullen am Schluss werden nicht ausgegeben.
`p`	`void *`	Zeiger (Darstellung hängt von Implementierung ab).
`n`	`int *`	Anzahl der aktuell von `printf()` ausgegebenen Zeichen.

Die obigen Formatangaben lassen sich durch Steuerzeichen („flags“) zwischen dem `%`- und dem Umwandlungszeichen weiter modifizieren:

- `Zahl`: Minimale Feldbreite festlegen: Das umgewandelte Argument wird in einem Feld ausgegeben, das mindestens so breit ist, bei Bedarf aber auch breiter. Hat das umgewandelte Argument weniger Zeichen als die Feldbreite es verlangt, so werden auf der linken Seite Leerzeichen eingefügt.
- `.Zahl`: Genauigkeit von Gleitkommazahlen festlegen: Gibt die maximale Anzahl von Zeichen an, die nach dem Dezimalpunkt ausgegeben werden
- `-`: Ausrichten des umgewandelten Arguments am linken Rand des Ausgabefeldes (Leerzeichen werden bei Bedarf nicht links, sondern rechts eingefügt)
- `+`: Ausgabe einer Zahl stets mit Vorzeichen
- `Leerzeichen`: Ausgabe eines Leerzeichens vor einer Zahl, falls das erste Zeichen kein Vorzeichen ist
- `0`: Zahlen bei der Umwandlungen bis zur Feldbreite mit führenden Nullen aufüllen

Anstelle einer Zahl kann auch das Zeichen `*` als Feldbreite angegeben werden. In diesem Fall wird die Feldbreite durch eine zusätzlich an dieser Stelle in der Argumentliste angegebenen `int`-Variablen festgelegt:

```
int zahl = 1000;
int breite = 5;

printf("Der Wert von der Variable \"zahl\" ist: %*d", breite, zahl);
```

Die Formatangaben `%e` und `%g` können gleichermaßen zur Ausgabe von Gleitkomma-Zahlen in der Zehnerpotenz-Schreibweise verwendet werden. Sie unterscheiden sich nur bei Zahlen mit wenig Nachkommastellen. Beispielsweise würde die Ausgabe `printf("%g\n", 2.15);` als Ergebnis `2.15` anzeigen, während `printf("%e\n", 2.15);` als Ergebnis `2.150000e+00` liefern würde.

Soll eine `long`-Variante eines Integers ausgegeben werden, so muss vor das jeweilige Umwandlungszeichen ein `l` geschrieben werden, beispielsweise `lu` für `long unsigned int` oder `ld` für `long int`; für `long double` wird `L` geschrieben.

Soll das `%`-Zeichen innerhalb einer Zeichenkette selbst ausgegeben werden, so muss an dieser Stelle `%%` geschrieben werden.

Soll über mehrere Zeilen hinweg Text mittels `printf()` ausgegeben werden, so ist meist es für eine bessere Lesbarkeit empfehlenswert, für jede neue Zeile eine eigene `printf()`-Anweisung zu schreiben.

`puts()` – Einzelne Zeichenketten ausgeben

Sollen nur einfache Zeichenketten (ohne Formatierung und ohne Variablenwerte) ausgegeben werden, so kann anstelle von `printf()` auch die Funktion `puts()` aus der Standard-Bibliothek *stdio.h* verwendet werden. Die in der Tabelle *Escape-Sequenzen* aufgelisteten Sonderzeichen können auch bei `puts()` verwendet werden, es muss jedoch am Ende einer Ausgabezeile kein `\n` angehängt werden; `puts()` gibt automatisch jeden String in einer neuen Zeile aus.

`putchar()` – Einzelne Zeichen ausgeben

Mittels `putchar()` können einzelne Zeichen auf dem Bildschirm ausgegeben werden. Diese Funktion wird nicht nur von den anderen Ausgabefunktionen aufgerufen, sondern kann auch verwendet werden, wenn beispielsweise eine Datei zeichenweise eingelesen und nach Anwendung eines Filters wieder zeichenweise auf dem Bildschirm ausgegeben werden soll.[3]

`scanf()` – Daten formatiert einlesen

Die Funktion `scanf()` kann als flexible Funktion verwendet werden, um Daten direkt vom Bildschirm beziehungsweise von der Tastatur einzulesen. Dabei wird bei `scanf()`, ebenso wie bei `printf()`, ein Formatstring angegeben, der das Format der Eingabe festlegt. Die Funktion weist dann die eingelesen Daten, die dem Format entsprechen, vom Bildschirm ein und weist ihnen eine oder mehrere Programmvariablen zu. Im Formatstring können die gleichen *Umwandlungszeichen* wie bei `printf()` verwendet werden.

Die Eingabe mittels `scanf()` erfolgt „gepuffert“, d.h. die mit der Tastatur eingegebenen Zeichen werden zunächst in einem Zwischenspeicher („Puffer“) des Betriebsystems abgelegt. Erst, wenn der Benutzer die `Enter`-Taste drückt, wird der eingegebene Text von `scanf()` verarbeitet.

[3] Streng genommen handelt es sich bei `putchar()` nicht um eine Funktion, sondern um ein *Makro*: Letztlich wird `putchar(Zeichen)` vom Präprozessor durch einen Funktionsaufruf von `fputc(Zeichen, stdin)` ersetzt. Die Funktion `fputc()` wird im Abschnitt *Dateien und Verzeichnisse* näher beschrieben.

Bei der Zuweisung der eingelesenen Daten wird bei Benutzung der Funktion `scanf()` nicht der jeweilige Variablenname, sondern stets die zugehörige Speicheradresse angegeben, an welcher die Daten abgelegt werden sollen; diese kann leicht mittels des *Adress-Operators* `&` bestimmt werden. Um also beispielsweise einen `int`-Wert vom Bildschirm einzulesen, gibt man folgendes ein:

```
int n;

// Benutzer zur Eingabe auffordern:
printf("Bitte einen ganzzahligen Wert eingeben: ")

// Eingegebenen Wert einlesen:
scanf("%i", &n);
```

Sobald der Benutzer seine Eingabe mit `Enter` bestätigt, wird im obigen Beispiel die eingegebene Zahl eingelesen und am Speicherplatz der Variablen `n` hinterlegt.

Zum Einlesen von Zeichenketten muss dem Variablennamen kein `&` vorangestellt werden, da es sich bei einer Zeichenkette um ein *Array* handelt. Dieses wiederum entspricht einem *Zeiger* auf den ersten Eintrag, und ab eben dieser Stelle soll die eingelesene Zeichenkette abgelegt werden. Beim Einlesen von Daten in Felder muss allerdings beachtet werden, dass der angegebene Zeiger bereits *initialisiert* wurde. Eine simple Methode, um dies sicherzustellen, ist dass eine String-Variable nicht mit `char *mystring;`, sondern beispielsweise mit `char mystring[100];` definiert wird.

Whitespace als Trennzeichen

Mit einer einzelnen `scanf()`-Funktion können auch mehrere Werte gleichzeitig eingelesen werden, wenn mehrere Umwandlungszeichen im Formatstring und entsprechend viele Speicheradressen als weitere Argumente angegeben werden. Beim Einlesen achtet `scanf()` dabei so genannte Whitespace-Zeichen (Leerzeichen, Tabulator-Zeichen oder Neues-Zeile-Zeichen), um die einzelnen Daten voneinander zu trennen. Soll der Benutzer beispielsweise zwei beliebige Zahlen eingeben, so können diese mit einem einfachen Leerzeichen zwischen ihnen, aber ebenso in zwei getrennten Zeilen eingegeben werden.

```
int n1, n2;

// Benutzer zur Eingabe auffordern:
printf("Bitte zwei beliebige Werte eingeben: ")

// Eingegebene Werte einlesen:
scanf("%f %f", &n1, &n2);
```

`fflush()` – Zwischenspeicher löschen

Da die Daten bei Verwendung von `scanf()` zunächst in einen Zwischenspeicher eingelesen werden, können Probleme auftreten, wenn der Benutzer mehr durch Whitespace-Zeichen getrennte Werte eingibt, als beim Aufruf der Funktion `scanf()` verarbeitet werden. Die

restlichen Werte verbleiben in diesem Fall im Zwischenspeicher und würden beim nächsten Aufruf von `scanf()` noch vor der eigentlich erwarteten Eingabe verarbeitet werden. Eine Abhilfe hierfür schafft die Funktion `fflush()`, die nach jedem Aufruf von `scanf()` aufgerufen werden sollte und ein Löschen aller noch im Zwischenspeicher abgelegten Werte bewirkt.

Beim Einlesen von Zeichenketten mittels `%s` ist das wortweise Einlesen von `scanf()` oftmals hinderlich, da in der mit `%s` verknüpften Variable nur Text bis zum ersten Whitespace-Zeichen (Leerzeichen, Tabulator-Zeichen oder Neues-Zeile-Zeichen) gespeichert wird. Ganze Zeilen, die aus beliebig vielen Wörtern bestehen, sollten daher bevorzugt mittels `gets()` oder `fgets()` eingelesen werden.

`gets()` und `fgets()` – Einzelne Zeichenketten einlesen

Um eine Textzeile auf einmal einzulesen, kann die Funktion `gets()` aus der Standard-Bibliothek *stdio.h* verwendet werden. Diese Funktion liest eine Textzeile vom Bildschirm ein und speichert sie in der angegebenen Variablen ein:

```
int mystring[81];

gets(mystring);
```

Ein Neues-Zeile-Zeichen `\n` am Ende des Eingabestrings wird von `gets()` automatisch abgeschnitten, das Zeichen `\0` zum Beenden der Zeichenkette automatisch angefügt. Wichtig ist allerdings bei der Verwendung von `gets()`, dass der angegebene String-Pointer auf ein ausreichend großes Feld zeigt. Im obigen Beispiel darf die eingelesene Zeile somit nicht mehr als 80 Zeichen haben, da auch noch Platz für das Zeichen `\0` bleiben muss. Werden die Feldgrenzen überschritten, kann dies ein unkontrolliertes Verhalten des Programms oder gar einen Programmabsturz zur Folge haben.[4]

Als bessere Alternative zu `gets()` kann die Funktion `fgets()` verwendet werden, welche die Anzahl der maximal eingelesenen Zeichen beschränkt:

```
int mystring[81];
int n = 80;

fgets(mystring, n, stdin);
```

Im Unterschied zu `gets()` speichert `fgets()` das Neue-Zeile-Zeichen `\n` mit in der eingelesenen Zeichenkette, was unter Umständen bei der Längenangabe n berücksichtigt werden muss. Die Funktion `fgets()` gibt, wenn eine Zeichenkette erfolgreich eingelesen wurde, einen Zeiger als Ergebnis zurück, der mit der Speicheradresse der angegebenen Stringvariablen übereinstimmt; bei einem Fehler wird `NULL` als Ergebnis zurück gegeben. Um eine Textzeile auf einmal einzulesen, kann die Funktion `gets()` aus der Standard-Bibliothek

[4] Im neuen C11-Standard wird `gets()` aufgrund seiner Fehleranfälligkeit nicht mehr als Standard gelistet, den ein Compiler abdecken *muss*. Da die Funktion in sehr vielen Programmcodes vorkommt, wird `gcc` wohl auch in absehbarer Zukunft diese Funktion unterstützen. In C11 wurde dafür die ähnliche Funktion `gets_s()` im optionalen Teil von `stdio.h` aufgenommen, die jedoch ebenfalls nicht jeder Compiler zwingend unterstützen muss. Dies ist ein weiterer Grund, bevorzugt `fgets()` zu verwenden.

stdio.h verwendet werden. Diese Funktion liest eine Textzeile vom Bildschirm ein und speichert sie in der angegebenen Variablen ein:

`getchar()` – Einzelne Zeichen einlesen

Um einzelne Zeichen vom Standard-Eingang (Bildschirm bzw. Tastatur) zu lesen, kann die Funktion `getchar()` verwendet werden.[5] Ebenso wie bei der Funktion `scanf()` gibt die Funktion erst dann das gelesene Zeichen als Ergebnis zurück, wenn der Benutzer die `Enter`-Taste drückt; dies lässt sich beispielsweise für eine Abfrage der Art `[Yn]` für `"Yes"` oder `"No"` nutzen, wobei üblicherweise der groß geschriebene Buchstabe als Vorauswahl gilt und gesetzt wird, wenn keine explizite Eingabe vom Benutzer erfolgt.

Wird das Zeichen nach einer Umlenkung des Standard-Eingangs (beispielsweise mittels *freopen()*) nicht von der Tastatur, sondern von einer Datei eingelesen, so wird so lange jeweils ein einzelnes Zeichen zurückgegeben, bis ein Fehler auftritt oder die Funktion auf das Ende des Datenstroms bzw. der Datei trifft; in diesem Fall wird `EOF` als Ergebnis zurückgegeben.

... to be continued ...

[5] Streng genommen handelt es sich bei `getchar()` nicht um eine Funktion, sondern um ein *Makro*. Letztlich wird `getchar()` vom Präprozessor durch einen Funktionsaufruf von `fgetc(stdin)` ersetzt. Die Funktion `fputc()` wird im Abschnitt *Dateien und Verzeichnisse* näher beschrieben.

Operatoren und Funktionen

Operatoren

Mit einem Operator werden üblicherweise zwei Aussagen oder Variablen miteinander verknüpft. Ist die Anwendung des Operators für die angegebenen Variablen erlaubt, so kann dieser – je nach Operator – einen einzelnen Rückgabewert als Ergebnis liefern. Beispielsweise wird durch den *Zuweisungsoperator* `=` das Ergebnis des Ausdrucks auf der rechten Seite in der links vom Istgleich-Zeichen stehende Variablen gespeichert.

In C existieren auch Operatoren, die nur auf eine einzelne Variable angewendet werden, beispielsweise der *Adressoperator* `&`, der die Speicheradresse einer Variablen oder einer Funktion als Ergebnis liefert, oder der *Inhaltsoperator* `*`, der den an einer Speicherstelle abgelegten Wert ausgibt.

Die wichtigsten Operatoren werden in den folgenden Abschnitten kurz beschrieben.

Mathematische Operatoren

Die mathematischen Grundrechenarten Addition, Subtraktion, Multiplikation und Division lassen sich in C erwartungsgemäß mittels der Operatoren `+`, `-`, `*` und `/` durchführen; dabei werden jeweils zwei numerische Variablen oder Ausdrücke zu einem neuen Ergebnis verknüpft. Als Einziges ist die Division durch Null nicht erlaubt, sie führt zu Fehlermeldungen beim Compilieren oder kann das Abstürzen des Programms zur Folge haben. Neben den vier Operatoren für die Grundrechenarten existiert zusätzlich der Modulo-Operator `%`, der den ganzzahligen Divisions-Rest angibt; er liefert somit stets einen Wert vom Typ `int` als Ergebnis.

Operator	Beschreibung
`+`	Addition zweier Zahlen
`-`	Subtraktion zweier Zahlen
`*`	Multiplikation zweier Zahlen
`/`	Division zweier Zahlen (Division durch Null nicht erlaubt!)
`%`	Ganzzahliger Rest bei der Division zweier Zahlen

Darüber hinaus existieren in C die beiden weiteren Operatoren `++` und `--`, die jeweils auf eine einzige ganzzahlige Variable angewendet werden. Der Inkrement-Operator `++` erhöht den Wert der Variablen um `1`, der Dekrement-Operator `--` erniedrigt den Wert der Variablen um `1`. Beide Operatoren werden üblicherweise verwendet, um beispielsweise in

Schleifen den Wert einer Zählvariablen schrittweise um Eins zu erhöhen beziehungsweise erniedrigen und dabei den Variablenwert mittels des Zuweisungsoperators = einer anderen Variablen zuzuweisen:

```
// Erhöht zunächst x um 1, weist anschließend y den Wert von x zu:
y = ++x

// Weist zunächst y den Wert von x zu, erhöht anschließend x um 1:
y = x++
```

Wie das obige Beispiel zeigt, ist es bei der Anwendung der Operatoren `++` und `--` von Bedeutung, ob der Operator vor oder nach der jeweiligen Variablen steht; im ersten Fall wird die Variable erst inkrementiert beziehungsweise dekrementiert und anschließend zugewiesen, im zweiten Fall ist es umgekehrt.

Die Operatoren `++` und `--` haben für *Zeiger auf Felder* eine eigene Bedeutung: Sie erhöhen den Wert des Zeigers nicht um 1, sondern um die Länge des Datentyps, der in dem Array gespeichert ist, also beispielsweise um `size(int)` für ein Array mit `int`-Variablen. Somit können in Schleifen auch Felder mit dem Inkrement- bzw. Dekrement-Operator durchlaufen werden.

Zuweisungsoperatoren

Der wichtigste Zuweisungsoperator ist das Istgleich-Zeichen `=`: Es weist den Wert des Ausdrucks, der rechts des Istgleich-Zeichens steht, der links stehenden Variablen zu.

Operator	Beschreibung
`=`	Wertzuweisung (von rechts nach links)
`+=`	Erhöhung einer Variablen (um Term auf der rechten Seite)
`-=`	Reduzierung einer Variablen
`*=`	Vervielfachung einer Variablen
`/=`	Teilung einer Variablen (durch Term auf der rechten Seite)
`%=`	Ganzzahliger Rest bei Division (durch Term auf der rechten Seite)

Neben diesem einfachen Zuweisungsoperator existieren zusätzlich noch die kombinierten Zuweisungsoperatoren `+=`, `-=`, `*=`, `/=` und `%=`. Sie werten jeweils zunächst den Ausdruck auf der rechten Seite aus, führen anschließend die jeweilige Operation mit der links stehenden Variablen aus, und weisen schließlich das Ergebnis wieder der links stehenden Variablen zu. Somit ist beispielsweise `x -= 1` eine Kurzschreibweise für `x = x - 1`.

Vergleichsoperatoren

Vergleichsoperatoren dienen zum Wertevergleich zweier Variablen oder Ausdrücke. Ist der Vergleich wahr, so liefern sie „wahr“ als Ergebnis zurück, in C also einen von Null verschiedenen Wert. Ist im umgekehrten Fall der Vergleich nicht wahr, so wird als Ergebnis „falsch“ (also der Wert Null) zurück geliefert.

Operator	Beschreibung
==	Test auf Wertgleichheit
!=	Test auf Ungleichheit
<	Test, ob kleiner
<=	Test, ob kleiner oder gleich
=>	Test, ob größer oder gleich
>	Test, ob größer

Vergleichsoperatoren werden vor allem in Bedingungen von *if-Anweisungen* eingesetzt.

Logische Operatoren

Wie in der Aussagenlogik der Mathematik lassen sich auch in C mehrere Ausdrücke mittels logischer Operatoren zu einem Gesamt-Ausdruck kombinieren. Die jeweiligen Symbole für die logischen Verknüpfungen Und, Oder und Nicht sind in der folgenden Tabelle aufgelistet.

Operator	Beschreibung
!	Negation
&&	Logisches Und
||	Logisches Oder

Das `!`-Zeichen als logisches Nicht bezieht sich auf den unmittelbar rechts stehenden Ausdruck und kehrt dabei den Wahrheitswert des Ausdrucks um. Die anderen beiden Operatoren `&&` und `||` verknüpfen den unmittelbar links und den unmittelbar rechts stehenden Ausdruck zu einer Gesamt-Aussage. Eine Und-Verknüpfung ist genau dann wahr, wenn beide Teil-Ausdrücke wahr sind, eine Oder-Verknüpfung ist wahr, wenn mindestens einer der beiden Ausdrücke wahr ist.

Zur besseren Lesbarkeit sowie zur Vermeidung von Fehlern ist es empfehlenswert, die durch logische Ausdrücke verknüpften Aussagen stets in runde Klammern zu setzen, also beispielsweise `(ausdruck_1 && ausdruck_2)` zu schreiben.

Der Bedingungs-Operator

Der Bedingungs-Operator ist der einzige Operator in C, der drei Ausdrücke miteinander verbindet. Er hat folgenden Aufbau:

```
bedingung ? anweisung1 : anweisung2
```

Wenn der Bedingungs-Ausdruck wahr ist, also einen Wert ungleich Null als Ergebnis liefert, so wird `anweisung1` ausgeführt, ist der Bedingungs-Ausdruck falsch, so wird `anweisung2` ausgeführt. Beim Bedingungs-Operator handelt es sich somit um eine sehr kurze Schreibform einer *if-else-Anweisung*. Er kann unter anderem bei der Zuweisung von Werten eingesetzt werden, um beispielsweise einer neuen Variablen den größeren Wert zweier anderer Variablen zuzuweisen:

```
// Die größere der beiden Variabeln var_1 und var_2 in my_var abspeichern:
my_var = ( var_1 > var_2 ) ? var_1 : var_2;
```

Der Cast-Operator

Mittels des so genannten Cast-Operators kann eine Variable mit einem bestimmten Datentyp manuell in einen anderen Datentyp umgewandelt werden.

Von C werden auch automatisch derartige Umwandlungen vorgenommen, beispielsweise wenn ein `int`-Wert mit einem `float`-Wert multipliziert werden soll; hierbei wird der `int`-Wert zunächst in einen `float`-Wert gewandelt, damit der Operator auf zwei syntaktisch gleichwertige Objekte angewendet wird. Ebenso werden *enum*-Konstanten automatisch nach `int` konvertiert.

Während eine automatische Konvertierung in den jeweils nächst „größeren“ Datentyp ohne Probleme möglich ist (beispielsweise `float -> double` oder `double -> long double`), so ist eine Konvertierung in einen kleineren Datentyp oftmals mit Verlusten behaftet; beispielsweise kann der `float`-Wert 3.14 nur gerundet als `int`-Wert dargestellt werden. Eine solche derartige Umwandlung erfolgt in C dadurch, dass man bei der Zuweisung vor den Ausdruck auf der rechten Seite den gewünschten Datentyp in runden Klammern angibt:

```
int n;
float pi=3.14;

n = (int) pi;
```

Die runde Klammer mit dem darin enthaltenen Ziel-Datentyp wird hierbei als Cast-Operator bezeichnet. Am häufigsten werden Casts wohl beim *dynamischen Reservieren von Speicherplatz* verwendet: Hierbei wird zunächst ein unbestimmter Zeiger auf den reservierten Speicherplatz erzeugt, der dann in einen Zeiger des gewünschten Typs umgewandelt wird.

Der `sizeof`-Operator

Der `sizeof`-Operator gibt die Größe des anschließend angegebenen Datentyps oder der anschließend angegebenen Variablen an. Die Angabe eines Datentyp muss dabei (wie beim *cast*-Operator) mit runden Klammern erfolgen; dies liegt daran, dass ansonsten nicht zwischen der Bezeichnung eines Datentyps und einem Variablennamen unterschieden werden kann. Beispielsweise würde also `sizeof (float);`, je nach Rechner-Architektur, den Wert `4` liefern. Wendet man den `sizeof`-Operator hingegen auf einen Variablennamen an, so können runde Klammern um den Variablennamen wahlweise gesetzt oder auch weggelassen werden.

Mit dem `sizeof`-Operator kann auch die Größe von *Feldern* oder *Zusammengesetzten Datentypen* ermittelt werden; sie entspricht der Summe der Größen aller darin vorkommenden Elemente.

Das Ergebnis von `sizeof` hat als Datentyp `size_t`, was gleichbedeutend mit `unsigned int` ist.

Der Komma-Operator

In C wird das Komma meist als Trennungszeichen für Funktionsargumente oder bei der Deklaration von Variablen verwendet. Es kann allerdings auch als Operator genutzt werden, wenn es zwischen zwei Ausdrücken steht. Hierbei wird zunächst der links vom Komma stehende Ausdruck ausgewertet, anschließend der rechte. Als Ergebnis wird der Wert des rechten Ausdrucks zurückgegeben.

Am häufigsten wird der Komma-Operator in *for-Schleifen* eingesetzt.

Rangfolge der Operatoren

In der folgenden *Tabelle* ist aufgelistet, welche Operatoren mit welcher Priorität ausgewertet werden (ebenso wie „Punkt vor Strich“ in der Mathematik). Operatoren mit einem hohen Rang, die weiter oben in der Tabelle stehen, werden vor Operatoren mit einem niedrigen Rang ausgewertet. Haben zwei Operatoren den gleichen Rang, so entscheidet die so genannte Assoziativität, in welcher Reihenfolge ein Ausdruck auszuwerten ist:

- Bei der Assoziativität „von links nach rechts“ wird der Ausdruck der Reihe nach abgearbeitet, genau so, wie man den Code liest.
- Bei der Assoziativität „von rechts nach links“ wird zunächst der Ausdruck auf der rechten Seite des Operators ausgewertet, und erst anschließend der Operator auf den sich ergebenden Ausdruck angewendet.

Rang	Operator	Assoziativität
1	Funktionsaufruf `()`, Array-Operator `[]`, Strukturzugriff `.` und `->`	von links nach rechts
2	Adress-Operator `&`, Inhalts-Operator `*`, Vorzeichen-Operator `+` und `-`, Negation `!`, Inkrement `++` und Dekrement `--`, Einerkomplement `~`, `sizeof`, `(cast)`	von rechts nach links
3	Multiplikation `*`, Division `/`, Modulo `%`	von links nach rechts
4	Addition `+`, Subtraktion `-`	von links nach rechts
5	Bitweises Schieben `>>` und `<<`	von links nach rechts
6	Werte-Vergleich `> < >= <=`	von links nach rechts
7	Werte-Vergleich `==` und `!=`	von links nach rechts
8	Binäres Und `&`	Von links nach rechts
9	Binäres Entweder-Oder `^`	von links nach rechts
10	Binäres Oder `|`	von links nach rechts
11	Logisches Und `&&`	von links nach rechts
12	Logisches Oder `||`	von links nach rechts
13	Bedingungsoperator `?:`	Von rechts nach links
14	Zuweisungsoperator `= *= /= %= += -= ^= |= &= <<= >>=`	von rechts nach links
15	Sequenzoperator `,`	von links nach rechts

Enthält ein Ausdruck mehrere Operatoren mit gleicher Priorität, so werden die meisten Operatoren von links nach rechts ausgewertet. Beispielsweise haben im Ausdruck `3 *`

`4 % 5 / 2` alle Operatoren die gleiche Priorität, sie werden gemäß ihrer Assoziativität von links nach rechts ausgewertet, so dass der Ausdruck formal mit `((3 * 4) % 5) / 2` identisch ist; somit ist das Ergebnis gleich `(12 % 5) / 2 = 2 / 2 = 1`.

Zur besseren Lesbarkeit können Teil-Aussagen die durch einen Operator mit höherer Priorität verbunden sind jederzeit, auch wenn es nicht notwendig ist, in runde Klammern gesetzt werden, ohne den Wert der Aussage zu verändern.

Funktionen

Funktionen werden verwendet, um einzelne, durch geschweifte Klammern begrenzte Code-Blöcke mit einem Namen zu versehen. Damit können Funktionen an beliebigen anderen Stellen im Programm aufgerufen werden.

Eine Funktion kann somit als „Unterprogramm“ angesehen werden, dem gegebenenfalls ein oder auch mehrere Werte als so genannte „Argumente“ übergeben werden können und das je nach Definition einen Wert als Ergebnis zurück gibt.

Die Definition einer Funktion hat folgenden Aufbau:

```
// Definition einer Funktion:
rueckgabe_typ funktionsname( arg1, arg2, ... )
{
    Anweisungen
}
```

Der Rückgabe-Typ gibt den Datentyp an, den die Funktion zurück gibt, beispielsweise `int` für ein ganzzahliges Ergebnis oder `char *` für eine Zeichenkette. Liefert die Funktion keinen Wert zurück, wird `void` als Rückgabe-Typ geschrieben. Die Argumentenliste der Funktion kann entweder leer sein oder eine beliebige Anzahl an zu übergebenden Argumenten beinhalten, wobei jedes Argument aus einem Argument-Typ und einem Argument-Namen besteht. Beim Aufruf der Funktion müssen die Datentypen der übergebenen Werte mit denen der bei der Deklaration angegebenen Argumentliste übereinstimmen.[1]

Bezüglich der Anweisungen innerhalb eines Funktionsblocks bestehen kaum Einschränkungen, außer dass es nicht möglich ist, innerhalb einer Funktion weitere Funktionen zu definieren. Neue Variablen, deren Gültigkeit auf die jeweilige Funktion beschränkt ist, müssen stets zu Beginn des Funktionsblocks definiert werden. Am Ende der Funktion verlieren diese „lokalen“ Variablen standardmäßig wieder ihre Gültigkeit; soll eine Variable ihren Wert jedoch bis zum nächsten Aufruf der Funktion behalten, muss bei der Definition der Variablen das Schlüsselwort *static* verwendet werden.

Soll eine Funktion einen Wert als Ergebnis zurückzugeben, so muss innerhalb der Funktion das Schlüsselwort `return` gesetzt werden, gefolgt von einem C-Ausdruck. Wenn die Funktion an einer `return`-Anweisung ankommt, wird der Ausdruck ausgewertet und das Ergebnis an die aufrufende Stelle im Programm zurück gegeben. Zu beachten ist lediglich,

[1] Streng genommen werden die Argumente bei der Definition als „formale Parameter“ bezeichnet, die beim Aufruf übergebenen Werte hingegen werden „aktuelle Parameter“ oder schlicht Argumente genannt.

dass der von `return` zurück gelieferte Wert mit dem in der Funktionsdefinition angegebenen Datentyp übereinstimmt, damit der Compiler keine Fehlermeldung ausgibt.

Nach der Definition der Funktion kann diese an beliebigen Stellen im Code genutzt werden, sie kann also auch von anderen Funktionen aufgerufen werden. Um eine Funktion allerdings bereits aufrufen zu können, wenn ihre Definition erst an einer späteren Stelle der Datei erfolgt, muss am Dateianfang – wie bei Variablen – zunächst der Prototyp der Funktion deklariert werden:[2]

```
// Deklaration des Funktions-Prototyps:
rueckgabe_typ funktionsname( arg1, arg2, ... );
```

Bei C-Programmen, die nur aus einer einzigen Datei bestehen, werden die Funktions-Prototypen üblicherweise gemeinsam mit der Deklaration von Variablen an den Anfang der Datei geschrieben. Die konkrete Definition der Funktionen erfolgt dann üblicherweise nach der Definition der Funktion `main()`.

Um eine Funktion aufzurufen, wird der Name der Funktion in Kombination mit einer Argumentliste in runden Klammern angegeben:

```
//  Aufruf einer Funktion:
funktionsname( arg1, arg2, ... );
```

Beim Aufruf einer Funktion müssen die Anzahl der übergebenen Argumente und ihre Datentypen mit der Funktions-Definition übereinstimmen.

C-Programme bestehen letztlich aus einer Vielzahl an Funktionen, die jeweils möglichst eine einzige, klar definierte Teilaufgabe übernehmen; entsprechend sollte der Funktionsname auf den Zweck der Funktion hinweisen. Eine Funktion Funktion sollte ebenfalls nicht allzu umfangreich sein, nur wenige Funktionen bestehen aus mehr als 30 Zeilen Code.[3] Auf diese Weise lassen sich einerseits einzelne Code-Teile leichter wieder verwerten, andererseits kann dadurch beim Suchen nach Fehlern der zu hinterfragende Code-Bereich schneller eingegrenzt werden.

Call by Value und Call by Reference

In C werden alle Argumente standardmäßig „by Value“ übergeben, das heißt, dass die übergebenen Werte beim Funktionsaufruf kopiert werden, und innerhalb der Funktion mit lokalen Kopien der Werte gearbeitet wird. Eine Funktion kann hierbei die Originalvariable nicht verändern.

Wenn eine Funktion übergebene Variablen jedoch verändern soll, so müssen anstelle der Variablenwerte die Adressen der jeweiligen Variablen übergeben werden. Eine derartige Übergabe wird als „Call by Reference“ bezeichnet: Anstelle der Variablen wird ein *Zeiger* auf die Variable als Argument übergeben. Ändert die Funktion den Wert der Speicher-

[2] Deklarationen von Funktionen sind für das Compilieren des Programms unerlässlich, da für jeden Funktionsaufruf geprüft wird, ob die Art und Anzahl der übergebenen Argumente korrekt ist.

[3] Eine Funktion sollte maximal 100 Zeilen umfassen. Die Hauptfunktion `main()` sollte nur Unterfunktionen aufrufen, um möglichst übersichtlich zu sein.

stelle, auf die der Pointer zeigt, so wird, wenn der Variablenwert erneut abgerufen wird, die Veränderung auch im restlichen Programmteil festgestellt.

Komplexe Datentypen, beispielsweise *Strukturen*, werden fast nie direkt, sondern meistens mittels eines Zeigers an eine Funktion übergeben; dadurch muss nicht die ganze Struktur, sondern nur die Speicheradresse (ein `unsigned int`-Wert) kopiert werden. Wird ein *Array* mittels eines Pointers an eine Funktion übergeben, so wird häufig dessen maximale Anzahl an Elementen (ein `int`-Wert) als zusätzliches Argument an die Funktion übergeben.

Lokale Variablen

Innerhalb einer Funktion können, ebenso wie am Anfang einer Quellcode-Datei, neue Variablen deklariert werden. Die in der Funktionsdefinition angegebenen Parameter-Namen werden automatisch als neue Variablen deklariert. Beim Aufruf einer Funktion werden den Parameter-Namen dann die entsprechenden Argumente als Werte zugewiesen.

Die so genannten „lokalen“ Variablen, die innerhalb einer Funktion definiert werden, sind völlig unabhängig von den Variablen, die außerhalb der Funktion existieren. Variablen des Programms können nur als Argumente an die Funktion übergeben werden, und Variablenwerte der Funktion können nur über die `return`-Anweisung an das Programm zurückgegeben werden.

Gibt es in einem Programm eine Variable `var_1`, so kann innerhalb einer Funktion also dennoch eine gleichnamige Variable `var_1` definiert werden. Die lokale Variable „überdeckt“ in diesem Fall die Programmvariable, bis die Funktion abgearbeitet ist. Mit dem Funktionsende erlischt eine lokale Variable wieder, es sei denn, sie wurde als *static* deklariert. In diesem Fall hat die lokale Variable beim nächsten Funktionsaufruf den Wert, den sie beim Beenden des vorhergehenden Funktionsaufrufs hatte.

Rekursion

Ruft eine Funktion in ihrem Anweisungsblock sich selbst auf, so spricht man von Rekursion. Das wohl bekannteste Beispiel einer rekursiven Funktion ist die so genannte Fakultät $x!$:

$$x! = x \cdot (x-1) \cdot (x-2) \cdot \ldots \cdot 2 \cdot 1$$

Diese mathematische Funktion, die für positive ganzzahlige Werte definiert ist, kann mittels einer C-Funktion für jeden beliebigen Wert x rekursiv mittels $x! = x \cdot (x-1)!$ berechnet werden:

```
unsigned int fakultaet(unsigned int x)
{
    if (a == 1)
    {
        return 1;
    }
    else
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
    {
        x *= fakultaet(x-1);
        return x;
    }
}
```

Bei diesem Beispiel wird die Funktion `fakultaet` so lange von sich selbst aufgerufen, bis das Argument `x` gleich `1` ist. Die zurückgegebenen Werte werden dabei jeweils mit Hilfe des Zuweisungsoperators `*=` mit dem als Argument übergebenen Wert von `x` multipliziert, das Ergebnis wird an die aufrufende Funktion zurückgegeben.

Rekursive Funktionen sollten, sofern möglich, vermieden werden. Der Grund liegt darin, dass der Computer bei jedem neuen Funktionsaufruf unter anderem Variablenwerte kopieren und neue Variablen initiieren muss, was zu einer Verlangsamung des Programms führt. Die Fakultäts-Funktion kann beispielsweise auch geschickter mittels einer *for*-Schleife implementiert werden, dank der insbesondere bereits berechnete Teilergebnisse nicht erneut berechnet werden müssen:

```
unsigned int fakultaet(unsigned int n)
{
    int i;
    int result = 1;

    for (i=1; i<=n; i++)
    {
        ergebnis *= i;
    }

    return result;
}
```

In manchen Fällen, beispielsweise beim „Merge-Sort“-Verfahren, ist Rekursion hingegen unvermeidbar; aufgrund der effizienteren Vorgehensweise ist dieses Sortierverfahren dem klassischen „Bubble-Sort“-Verfahren, das ohne Rekursion auskommt, bei großen Datenmengen weit überlegen.

Kontrollstrukturen

Im folgenden Abschnitt werden die grundlegenden Kontrollstrukturen vorgestellt, mit denen sich der Ablauf eine C-Programms steuern lässt.

`if`, `elif` und `else` – Bedingte Anweisungen

Mit Hilfe des Schlüsselworts `if` kann an einer beliebigen Stelle im Programm eine Bedingung formuliert werden, so dass die Anweisung(en) im unmittelbar folgenden Code-Block nur dann ausgeführt werden, sofern die Bedingung einen wahren Wert (ungleich Null) ergibt.

Eine `if`-Anweisung ist also folgendermaßen aufgebaut:

```
if (Bedingung)
{
    Anweisungen
}
```

In den runden Klammern können mittels der logischen Verknüpfungsoperatoren `and` beziehungsweise `or` mehrere Teilbedingungen zu einer einzigen Bedingung zusammengefügt werden. Bei einer einzeiligen Anweisung können die geschweiften Klammern weggelassen werden. Liefert die Bedingung den Wert Null, so wird der Anweisungsblock übersprungen und das Programm fortgesetzt.

Eine `if`-Anweisung kann um den Zusatz `else` erweitert werden. Diese Konstruktion wird immer dann verwendet, wenn man zwischen *genau* zwei Alternativen auswählen möchte.

```
if (Bedingung)
{
    Anweisungen
}
else
{
    Anweisungen
}
```

Der Vorteil einer `if-else`-Bedingung gegenüber der Verwendung zweier `if`-Anweisungen besteht darin, dass nur einmalig eine Bedingung getestet wird und das Programm somit schneller ausgeführt werden kann.

Soll neben der `if`-Bedingung eine (oder mehrere) weitere Bedingung getestet werden, so kann dies mittels des kombinierten Schlüsselworts `else if` geschehen. Die `else if`-Anweisungen werden nur dann ausgeführt, wenn die `if`-Bedingung falsch und die `elif`-Bedingung wahr ist.

```
if (Bedingung_1)
{
    Anweisungen
}
else if (Bedingung_2)
{
    Anweisungen
}
```

Allgemein können in einer `if`-Struktur mehrere `else if`-Bedingungen, aber nur ein `else`-Block vorkommen.

`switch` – Fallunterscheidungen

Mittels des Schlüsselworts `switch` kann in C eine Fallunterscheidung eingeleitet werden. Hierbei wird der nach dem Schlüsselwort `switch` in runden Klammern angegebene Ausdruck ausgewertet, und in Abhängigkeit des sich ergebenden Werts einer der folgenden Fälle ausgewählt:

```
switch (Ausdruck)
{
    case const_1:
        Anweisungen_1

    case const_2:
        Anweisungen_2

    ...

    default:
        Default-Anweisungen

}
```

Bei den Konstanten, mit denen der Wert von `Ausdruck` verglichen wird, muss es sich um `int`- oder `char`-Werte handeln, die nicht mehrfach vergeben werden dürfen. Trifft kein `case` zu, so werden die unter `default` angegebenen Anweisungen ausgeführt.

Trifft ein `case` zu, so werden die angegebenen Anweisungen ausgeführt, anschließend wird der `Ausdruck` mit den übrigen `case`-Konstanten verglichen. Möchte man dies vermeiden, so kann man am Ende der `case`-Anweisungen die Anweisung `break;` einfügen, die einen Abbruch der Fallunterscheidung an dieser Stelle zur Folge hat.

In C ist es auch möglich Anweisungen für mehrere `case`-Werte zu definieren. Die Syntax dazu lautet:

```
switch (Ausdruck)
{
    case const_1:
    case const_2:
    case const_3:
        Anweisungen

    ...
}
```

In diesem Fall werden die bei `case const_3` angegebenen Anweisungen auch aufgerufen, wenn die Vergleiche `case const_1` oder `case const_2` zutreffen.

`for` und `while` – Schleifen

Eine `for`-Schleife ist folgendermaßen aufgebaut:

```
for ( Initialisierung; Bedingung; Inkrementierung )
{
    Anweisungen
}
```

Gelangt das Programm zu einer `for`-Schleife, so werden nacheinander folgende Schritte ausgeführt:

- Zunächst wird der Initialisierungs-Ausdruck ausgewertet. Dieser ist üblicherweise eine Zuweisung, die eine Zählvariable auf einen bestimmten Wert setzt.
- Als nächstes wird der Bedingungs-Ausdruck ausgewertet. Dieser ist normalerweise ein relationaler Ausdruck (Vergleich).

  Wenn die Bedingung falsch ist, so wird die `for`-Schleife beendet, und das Programm springt zur nächsten Anweisung außerhalb der Schleife.

  Wenn die Bedingung wahr ist, so werden die im folgenden Block angegebenen Anweisung(en) ausgeführt.
- Nach der Ausführung der Anweisungen wird der Inkrementierungs-Ausdruck ausgewertet; hierbei wird beispielsweise die Zählvariable oder der Index eines Arrays mit jedem Schleifendurchlauf um 1 erhöht. Anschließend wird wiederum der Bedingungs-Ausdruck geprüft und gegebenenfalls die Ausführung der Schleifenanweisungen fortgesetzt.

Innerhalb einer `for`-Anweisung können weitere `for`-Anweisungen auftreten, so dass auch über mehrere Zählvariablen iteriert werden kann. Bei einer nur einzeiligen Anweisung können die geschweiften Klammern weggelassen werden.

Soll eine Schleife vorzeitig beendet werden, so kann dies mittels des Schlüsselworts `break` erreicht werden: Trifft das Programm auf diese Anweisung, so wird die Schleife unmittelbar beendet. [#] Möchte man die Schleife nicht beenden, sondern nur den aktuellen Schleifendurchgang überspringen, so kann man das Schlüsselwort `continue` verwenden.

Trifft das Programm auf diese Anweisung, so wird der aktuelle Schleifendurchgang beendet, und das Programm fährt mit dem nächsten Schleifendurchgang fort.

Üblicherweise werden `for`-Schleifen verwendet, um mittels der Zählvariablen für eine bestimmte Anzahl von Durchläufen zu sorgen. Ist zu Beginn der Schleife nicht bekannt, wie häufig der folgende Anweisungsblock durchlaufen werden soll, wird hingegen meist eine `while`-Schleife eingesetzt.

Eine `while`-Schleife ist folgendermaßen aufgebaut:

```
while ( Bedingung )
{
    Anweisungen
}
```

Eine `while`-Schleife führt einen Anweisungsblock aus, solange die angegebene Bedingung wahr (nicht Null) ist. Das Programm wertet dabei zunächst den als Bedingung angegebenen Ausdruck aus, und nur falls dieser einen von Null verschiedenen Wert liefert, wird der Anweisungsblock ausgeführt. Ergibt der als Bedingung angegebene Ausdruck bereits bei der ersten Auswertung den Wert Null, so wird die `while`-Schleife übersprungen, ohne dass der Anweisungsblock ausgeführt wird.

Häufig werden `while`-Schleifen als Endlos-Schleifen verwendet, die einen (zunächst) wahren Ausdruck als Bedingung verwenden. Unter einer bestimmten Voraussetzung wird dann mittels einer `if`-Anweisung innerhalb des Schleifenblocks entweder der Bedingungsausdruck auf den Wert Null gesetzt oder die Schleife mittels `break` beendet.

Soll eine gewöhnliche `while`-Schleife, unabhängig von ihrer Bedingung, mindestens einmal ausgeführt werden, so wird in selteneren Fällen eine `do-while`-Schleife eingesetzt. Eines solche Schleife ist folgendermaßen aufgebaut:

```
do
{
    Anweisungen
} while ( Bedingung )
```

Da es stets möglich ist, eine `do-while`-Schleife auch mittels einer `while`-Schleife zu schreiben, werden letztere wegen ihrer besseren Lesbarkeit meist bevorzugt.

Funktionen für Felder und Zeichenketten

`malloc()` und `calloc()` – Dynamische Speicherreservierung

Soll die Größe eines Feldes erst zur Laufzeit bestimmt werden, so ermöglichen es die Funktionen `malloc()` und `calloc()` aus der Standard-Bibliothek *stdlib.h*, nach Möglichkeit ein entsprechend großes Stück an freiem Speicherplatz („memory“) zu finden und für das Feld zu reservieren („allocate“).

Der Speicher eines Programms setzt sich allgemein zusammen aus einem Teil namens „Stack“, der für statische Variablen reserviert ist, und einem dynamischen Teil namens „Heap“, auf den mittels `malloc()` oder `calloc()` zugegriffen werden kann.

Bei der Verwendung dieser Funktionen kann *valgrind* als „Debugger“ für dynamischen Speicherplatz eingesetzt werden.

Die Funktion `malloc()`

Als Ergebnis gibt die Funktion `malloc()` einen Zeiger auf die nutzbare Speicheradresse zurück, oder `NULL`, falls keine Speicherreservierung möglich war. Bei jeder neuen Speicherreservierung sollte der Rückgabewert geprüft und gegebenenfalls eine Fehlermeldung ausgegeben werden. Im erfolgreichen Fall hat der zurück gegebene Zeiger den Typ `void *` und wird üblicherweise vom Programmierer mittels des `cast`-Operators in einen Zeiger vom gewünschten Typ umgewandelt.

Um beispielsweise einen dynamischen Speicherplatz für ein Array mit 50 `int`-Werten zu erhalten, kann man folgendes eingeben:

```
numbers = (int *) malloc(50 * sizeof (int));
```

An die Funktion `malloc()` wird allgemein die zu reservierende Speichergröße in Bytes als Argument übergeben; für beispielsweise 50 Werte vom Datentyp `int` ist damit auch das Fünfzigfache der Größe dieses Datentyps nötig. Der Rückgabewert von `malloc()`, nämlich `void *`, wird mit Hilfe des Casts `(int *)` in einen Zeiger auf `int` umgewandelt.

Wird der Speicher nicht mehr benötigt, so muss er manuell mittels `free()` wieder freigegeben werden. Als Argument wird dabei der Name des variablen Speichers angegeben,

also beispielsweise `free(numbers)`. In C gibt es keinen „Garbage Collector“, der nicht mehr benötigte Speicherbereiche automatisch wieder freigibt; es ist also Aufgabe des Programmierers dafür zu sorgen, dass Speicher nach dem Gebrauch wieder freigegeben wird und somit kein Speicherleck entsteht.

Die Funktion `calloc()`

Neben der Funktion `malloc()` gibt es in der Standardbibliothek *stdlib.h* eine weitere Funktion zur dynamischen Speicherreservierung namens `calloc()`. Beim Aufruf dieser Funktion wird als erstes Argument die Anzahl der benötigten Variablen, als zweites Argument die Größe einer einzelnen Variablen in Bytes angegeben. Bei einer erfolgreichen Reservierung wird, wie bei `malloc()`, ein `void *`-Zeiger auf den reservierten Speicher zurückgegeben, andernfalls `NULL`. Der Unterschied zwischen `malloc()` und `calloc()` liegt darin, dass `calloc()` alle Bits im Speicherbereich auf `0` setzt und dadurch sicherstellt, dass zuvor mit `free()` freigegebene Daten zufällig weiterverarbeitet werden.

Auch bei der Verwendung von `calloc()` muss Speicher, der nicht mehr benötigt wird, manuell mittels `free()` wieder freigegeben werden.

Die Funktion `realloc()`

Mit der Funktion `realloc()` kann ein mit `malloc()` oder `calloc()` reservierter Speicherbereich nachträglich in seiner Größe verändert werden.

Als erstes Argument gibt man bei `realloc()` einen Zeiger auf einen bereits existierenden dynamischen Speicherbereich an, als zweites die gewünschte neue Größe des Speicherbereichs. Kann der angeforderte Speicher nicht an der bisherigen Adresse angelegt werden, weil dort kein ausreichend großer zusammenhängender Speicherbereich mehr frei ist, dann verschiebt `realloc()` den vorhandenen Speicherbereich an eine andere Stelle im Speicher, an der noch genügend Speicher frei ist.

```
numbers = (int *) realloc(numbers, 100 * sizeof (int));
```

Als Ergebnis gibt die Funktion `realloc()` ebenfalls einen `void *`-Zeiger auf den reservierten Speicherbereich zurück, wenn die Speicherreservierung erfolgreich war, andernfalls `NULL`. Übergibt man an `realloc()` einen `NULL`-Pointer als Adresse, so ist `realloc()` mit `malloc()` identisch und gibt einen Zeiger auf einen neu erstellten dynamischen Speicherbereich zurück.

`memcmp()` und `strcmp()` – Vergleiche von Feldern

In C kann man den Inhalt zweier Felder nicht direkt vergleichen, es kann hierfür also nicht `array_1 == array_2` geschrieben werden. Bei diesem Test würden lediglich, da der Name eines Feldes auf das erste im Feld gespeicherte Element verweist, die Speicheradressen zweier Variablen verglichen werden, jedoch nicht deren Inhalt.

Für einen inhaltlichen Vergleich müssen alle Einzelelemente der Felder miteinander verglichen werden. Dies kann automatisch mit der Funktion *memcmp()* aus der Standardbibliothek *string.h* durchgeführt werden. Bei identischen Feldern wird der Wert 0 als Ergebnis zurückgegeben. Stößt die Funktion im ersten Feld auf einen Wert, der größer ist als im zu vergleichenden Feld, so wird ein positiver Wert > 0 zurückgegeben, im umgekehrten Fall ein negativer Wert < 0.

Handelt es sich bei den Feldern um Zeichenketten, so sollte anstelle von `memcmp()` bevorzugt die Funktion `strcmp()` verwendet werden. Diese prüft ebenfalls Zeichen für Zeichen, ob die beiden angegebenen Zeichenketten übereinstimmen. Anders als bei `memcmp()` wird jedoch das Überprüfen der Feldinhalte beendet, sobald das String-Ende-Zeichen `\0` erreicht wird. Mögliche Inhalte der Felder hinter diesem Zeichen werden somit nicht verglichen.

`memcpy()` und `strcpy()` – Kopieren von Feldern

Der Funktion `strcpy()` wird als erstes Argument der Name des Zielstrings, als zweites Argument eine dorthin zu kopierende Zeichenkette übergeben:

```
char target_string[50];

strcpy(target_string, "Hallo Welt!");

puts(target_string);
// Ergebnis: "Hallo Welt!"
```

Der Zielstring wird von `strcpy()` automatisch mit dem Zeichenkette-Ende-Zeichen `'\0'` abgeschlossen. Wichtig ist zu beachten, dass `strcpy()` nicht prüft, ob der Zielstring ausreichend groß ist; reicht der Platz dort nicht aus, werden die Bytes einer anschließend im Speicher abgelegten Variablen überschrieben, was unvorhersehbare Fehler mit sich bringen kann. Als Programmierer muss man somit entweder selbst darauf achten, dass nicht Zielstring ausreichend groß ist, oder die Funktion *strncpy()* verwenden, welcher als drittes Argument die Anzahl n der zu kopierenden Zeichen übergeben wird.

`strcat()` – Verknüpfen von Zeichenketten

Der Funktion `strcat()` wird als erstes Argument der Name des Zielstrings, als zweites Argument eine dort anzufügenden Zeichenkette übergeben:

```
char target_string[50];

strcpy(target_string, "Hallo Welt!");;
strcat(target_string, " Auf Wiedersehen!");

puts(target_string);
// Ergebnis: "Hallo Welt! Auf Wiedersehen!"
```

`strcat()` überschreibt automatisch das Zeichenkette-Ende-Zeichen `'\0'` des Zielstring mit dem ersten Zeichen des anzuhängenden Strings und schließt nach dem Anfügen der restlichen Zeichen den Zielstring wiederum mit `'\0'` ab.

Ebenso wie bei `strcpy()` muss auch bei Verwendung von `strcat()` auf einen ausreichend grossen Zielstring geachtet werden. Als Alternativ kann die Funktion `strncat()` verwendet werden, der als drittes Argument eine Anzahl n an anzuhängenden Zeichen übergeben wird.

Zusammengesetzte Datentypen

`typedef` – Synonyme für andere Datentypen

Mit dem Schlüsselwort `typedef` kann ein neuer Name für einen beliebigen Datentyp vergeben werden. Die Syntax lautet dabei wie folgt:

```
typedef datentyp neuer_datentyp
```

Beispielsweise kann mittels `typedef int integer` ein „neuer“ Datentyp namens `integer` erzeugt werden. Dieser kann anschließend wie gewohnt bei Deklarationen von Variablen verwendet werden, beispielsweise wird durch `integer num_1;` eine neue Variable als Integer-Wert deklariert.

Die Verwendung von `typedef` ist insbesondere bei der Definition von zusammengesetzten Datentypen hilfreich.

`enum` – Aufzählungen

Aufzählungen („enumerations“) bieten neben *#define*-Anweisungen eine einfache Möglichkeit, einzelnen Begriffen eine Nummer zuzuweisen und sie somit im Quellcode als leicht lesbare Bezeichner verwenden zu können.

Bei der Deklaration eines `enum`-Typs werden die einzelnen Elemente der Aufzählung durch Komma-Zeichen getrennt aufgelistet. Sie bekommen dabei, sofern nicht explizit andere Werte angegeben werden, automatisch die Nummern `0, 1, 2, ...` zugewiesen; bei expliziten Wertzuweisungen wird der Wert für jedes folgende Element um `1` erhöht.

```
typedef enum
{
    const1, const2, const3, ...
} enum_name;

# Beispiel:

typedef enum
{
    MONTAG = 1, DIENSTAG, MITTWOCH, DONNERSTAG, FREITAG, SAMSTAG, SONNTAG
} wochentag;
```

Allgemein müssen die Elemente eines `enum`-Typs unterschiedliche Werte besitzen. Oftmals werden die aufgelisteten Elemente zudem in Großbuchstaben geschrieben, um hervorzuheben, dass es sich auch bei ihnen um (ganzzahlige) Konstante handelt.

Nach der obigen Deklaration ist beispielsweise `wochentag` als neuer Datentyp verfügbar, der stets durch einen „benannten“ `int`-Wert repräsentiert wird:

```
wochentag heute = DIENSTAG;

// Die zugewiesene Nummer ausgeben:

printf("Heute ist der %d. Tag der Woche\n", heute);
// Ergebnis: Heute ist der 2. Tag der Woche.

// Funktionen definieren:

wochentag morgen(wochentag heute)
{
    if (heute == SONNTAG)
        return 1;
    else
        return heute++;
}
```

Es können somit nach der Deklaration des `enum`-Datentyps auch dessen Elemente als numerische Konstante im C-Code verwendet werden.

`struct` – Strukturen

Strukturen („structs“) ermöglichen es in C mehrere Komponenten zu einer Einheit zusammenenzufassen, ohne dass diese den gleichen Datentyp haben müssen (wie es bei einem *Array* der Fall ist. Der Speicherplatzbedarf einer Struktur entspricht dabei dem Speicherplatzbedarf ihrer Komponenten. In dem meisten Fällen lassen sich Strukturen folgendermaßen definieren:

```
typedef struct
{
    // ... Deklaration der Komponenten ...

} struct_name;

// Beispiel:

typedef struct
{
    char name[50];
    int laenge;
    int breite;
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
    int hoehe;
} gegenstand;
```

Nach der Deklaration einer Struktur kann diese als neuer Datentyp verwendet werden. Die einzelnen Komponenten werden nicht dabei durchnummeriert, sondern lassen sich mittels des Strukturzugriff-Operators . über bei der Definition vergebene Schlüsselwörter ansprechen:

```
// Struktur-Objekt definieren:

gegenstand tisch =
{
    "Schreibtisch", 140, 60, 75
};

// Informationen zum Objekt ausgeben:

printf( "Der Gegenstand \"%s\" ist %d cm hoch.\n", tisch.name, tisch.hoehe );
// Ergebnis: Der Gegenstand "Schreibtisch" ist 75 cm hoch.
```

Handelt es sich bei einer Struktur-Komponente um einen Zeiger, beispielsweise eine Zeichenkette, so muss der Inhalts-Operator `*` vor den Strukturnamen geschrieben werden. Im obigen Beispiel würde man also nicht `tisch.*name` schreiben (was beim Compilieren einen Fehler verursachen würde), sondern `*tisch.name`, da der Strukturzugriff-Operator `.` eine höhere *Priorität* besitzt. Zuerst wird also der Strukturzugriff ausgewertet, wobei sich eine Variable vom Typ `char *` ergibt; anschließend kann diese mit dem Inhaltsoperator dereferenziert werden. Bei `*strukturname.komponente` kann somit der Punkt wie ein Teil des Veriablennamens gelesen werden.

Strukturen können andere Strukturen als Komponenten enthalten; rekursive Strukturen, die sich selbst als Komponente beinhalten, sind nicht möglich. Eine Struktur kann allerdings einen *Zeiger* auf sich selbst enthält, so dass beispielsweise so genannte Verkettungen möglich sind. Darauf wird im Abschnitt *Dynamische Datenstrukturen* näher eingegangen.

Zeiger auf Strukturen

Eine Struktur wird selten direkt als Argument an eine Funktion übergeben, da hierbei der gesamte Strukturinhalt kopiert werden müsste. Stattdessen wird üblicherweise ein Zeiger auf die Struktur an die Funktion übergeben.

Hat man beispielsweise eine Struktur `mystruct` mit den Komponenten `int a` und `int b` und ein bereits existierendes `mystruct`-Objekt `x_1`, so kann man mittels `mystruct * x_1_pointer = &x_1;` einen *Zeiger* auf die Struktur definieren. Mittels eines solchen Pointers kann man auf folgende Weise auf die Komponenten der Struktur zugreifen:

```
// Struktur deklarieren:
typedef struct
{
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
    int a;
    int b;
} mystruct;

// Struktur-Objekt erzeugen:
mystruct x = {3, 5};

// Pointer auf Struktur-Objekt erzeugen:
mystruct * xpointer = &x;

// Wertzuweisung mittels Pointer:
(*xpointer).a = 5;
```

Im obigen Beispiel sind die Klammmern um `*x_1_pointer` notwendig, da der Strukturzugriff-Operator `.` eine höhere *Priorität* hat als der Inhalts-Operator `*`. Da Strukturen und somit auch Zeiger auf Strukturen sehr häufig vorkommen und diese Schreibweise etwas umständlich ist, gibt es in C folgende Kurzschreibweise:

```
(*xpointer).a == xpointer->a
// Ergebnis: TRUE
```

Mit dem Pfeil-Operator `->` kann also in gleicher Weise auf die Komponenten eines Struktur-Pointers zugegriffen werden wie mit `.` auf die Komponennten der Struktur selbst.

`union` – Alternativen

Mittels des Schlüsselworts `union` lässt sich ein zusammengesetzter Datentyp definieren, bei dem sich die bei der Deklaration angegebenen Elemente einen gemeinsamen Speicherplatz teilen: Es kann dabei zu jedem Zeitpunkt nur eine der angegebenen Komponenten aktiv sein. Der Speicherplatzbedarf einer Union entspricht somit dem Speicherplatzbedarf der größten angegebenen Komponente. Die Deklaration einer `union` erfolgt nach folgendem Schema:

```
typedef union
{
    // ... Deklaration der Komponenten ...

} union_name;

// Beispiel:

typedef union
{
    char text[20];
    int ganzzahl;
    float kommazahl;
} cell_value;
```

Nach der Deklaration einer Union kann diese als neuer Datentyp verwendet werden. Der Zugriff auf die einzelnen möglichen Elemente, die eine Union-Variable beinhaltet, erfolgt wie bei Strukturen, mit dem .-Operator:

```
// Union-Variablen deklarieren:

cell_value cell_1 = {"Hallo Welt!"};
cell_value cell_2 = {42};
cell_value cell_3 = {2.35813};

// Auf Inhalt einer Union zugreifen:

printf("%s\n", cell_1.text)
```

Im Falle eines Zeigers auf eine `union`-Variable kann, ebenso wie bei *Zeigern auf Strukturen*, mit dem Pfeil-Operator `->` auf die einzelnen Komponenten zugegriffen werden.

Unabhängig davon, welche Komponente aktuell in einer `union`-Variable mit einem Wert versehen ist, können stets alle möglichen Komponenten der Union abgefragt werden; dabei wird der aktuell gespeicherte Wert mittels eines automatischen *Casts* in den jeweiligen Datentyp umgewandelt. Da diese Umwandlung zu unerwarteten Ergebnissen führen kann, kann es hilfreich sein, für die einzelnen Datentypen der Union-Komponenten symbolische Konstanten zu vergeben. Fasst man dann sowohl den aktuellen Typ der Union-Variablen sowie die Union-Variable zu einer Struktur zusammen, so lässt sich bei komplexeren Datentypen nicht nur Speicherplatz sparen, es kann auch mittels einer *case*-Anweisung gezielt Code in Abhängigkeit vom aktuellen Wert aufgerufen werden:

```
typedef enum
{
    STRING=0, INTEGER=1, FLOAT=2
} u_type;

typedef struct
{
    u_type type;
    cell_value value;
} cell_content;

cell_content my_cell;

my_cell.type = FLOAT;
my_cell.value = 3.14;

switch (my_cell.type)
{
    case STRING:
        printf("In dieser Zelle ist die Zeichenkette %s gespeichert.", *my_cell.
↪value);

    case INT:
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
        printf("In dieser Zelle ist die int-Zahl %d gespeichert.", my_cell.
↪value);

    case FLOAT:
        printf("In dieser Zelle ist die float-Zahl %f gespeichert.", my_cell.
↪value);
}
```

Auf diese Weise könnte in einem „echten“ Programm die Ausgaben eines Wertes aufgrund nicht nur seines Datentyps, sondern beispielsweise auch aufgrund von Darstellungsoptionen (Anzahl an Kommastellen, Prozentwert, usw.) angepasst werden.

Dateien und Verzeichnisse

Jede Ein- und Ausgabe von Daten erfolgt in C über Datenkanäle („Files“). Beim Programmstart werden automatisch die Standard-Files `stdin`, `stdout` und `stderr` geöffnet und mit dem Bildschirm verknüpft. Somit muss in Programmen nur die Standard-Bibliothek *stdio.h* eingebunden werden, damit Daten beispielsweise mittels *printf()* auf dem Bildschirm ausgegeben oder mittels *scanf()* von der Tastatur eingelesen werden können.[1]

Dateien und File-Pointer

Auf Dateien wird in C grundsätzlich über `FILE`-Objekte zugegriffen: Sämtliche Datenfunktionen benötigen oder liefern einen Zeiger auf ein solches Objekt. Am Anfang der Quellcode-Datei muss also zunächst ein solcher File-Pointer, bisweilen auch „Stream“ genannt, definiert werden:

```
#include <stdio.h>

FILE *fp;
```

Um eine Datei zu öffnen, wird die Funktion `fopen()` verwendet. Als erstes Argument wird hierbei der Pfadname der zu öffnenden Datei übergeben, als zweites ein Zeichen, das den Zugriffsmodus auf die Datei angibt:

- `"r"`: Textdatei zum Lesen öffnen
- `"w"`: Textdatei zum Schreiben neu erzeugen (gegebenenfalls alten Inhalt wegwerfen)
- `"a"`: Text anfügen; Datei zum Schreiben am Dateiende öffnen oder erzeugen
- `"r+"`: Textdatei zum Ändern öffnen (Lesen und Schreiben)
- `"w+"`: Textdatei zum Ändern erzeugen (gegebenenfalls alten Inhalt wegwerfen)
- `"a+"`: Datei neu erzeugen oder zum Ändern öffnen und Text anfügen (Schreiben am Ende)

[1] Programme, deren einzige Aufgabe darin besteht, Daten vom Bildschirm einzulesen, zu verarbeiten, und wieder auf dem Bildschirm auszugeben, werden bisweilen auch als „Filter“ bezeichnet. Derartige Programme können unter Linux mittels des Pipe-Zeichens verbunden werden, beispielsweise kann so in einer Shell `programm_1 | programm_2 | programm_3` eingegeben werden.

Als Ergebnis gibt `fopen()` einen File-Pointer auf die Datei zurück, oder `NULL`, falls beim Öffnen ein Fehler aufgetreten ist.

```
fp = fopen("/path/to/myfile","r");

if (fp == NULL)
    fprintf(stderr,"Datei konnte nicht geoeffnet werden.\n");
```

Wird der Zugriff auf eine Datei nicht mehr benötigt, so sollte sie mittels `fclose()` wieder geschlossen werden. Hierbei muss als Argument der zur geöffneten Datei gehörende File-Pointer angegeben werden, also beispielsweise `fclose(fp)`. Bei einem Schreibzugriff ist das Schließen einer Datei mittels `fclose()` Pflicht, da hierdurch unter anderem die Modifikationszeit der Datei aktualisiert wird.

Existenz einer Datei prüfen

In C gibt es keine eigenständige Funktion, um die Existenz einer Datei zu prüfen. Man kann allerdings die Funktion `fopen()` auch zu diesem Zweck nutzen:

```
// Existenz einer Datei prüfen
// Rückgabewert: 1 falls Datei existiert, 0 sonst.
int file_exists(char *filename)
{
    FILE *fp;
    int result;

    fp = fopen(filename, "r");
    if (fp == NULL)
    {
        result = 0;
    }
    else
    {
        result = 1;
        fclose(fp);
    }

    return result;
}
```

Hierbei wurde als Zugriffsmodus `"r"` gewählt, da die Datei nicht verändert werden soll und die Methode auch mit schreibgeschützten Dateien funktionieren soll. Die Rückgabewerte wurden im obigen Beispiel so gewählt, damit sie an einer anderen Stelle im Code innerhalb einer `if`-Abfrage genutzt werden können.

Daten in eine Datei schreiben

Wie bereits im Abschnitt *Ausgabe und Eingabe* beschrieben wurde, gibt es in C mehrere Möglichkeiten, um Daten von der Tastatur beziehungsweise vom Bildschirm („stdin“) einzulesen. Ebenso gibt es in C mehrere Möglichkeiten, um Inhalte aus Dateien einzulesen oder dorthin zu schreiben. Die einzelnen Funktionen sind dabei den bereits behandelten Funktionen sehr ähnlich.

`fprintf()` – Daten formatiert schreiben

Mit `fprintf()` können normale Zeichenketten, Sonderzeichen und Werte von Variablen mittels Platzhaltern in formatierter Weise in eine Datei geschrieben werden. Die Syntax entspricht dabei derjenigen von *printf()*, wobei als erstes Argument der Name eines File-Pointers angegeben werden muss:

```
FILE *fp;

// Datei öffnen:
fp = fopen(filename, "w");

// Daten schreiben:
fprintf(fp, "Teststring!\n");

// Datei schließen:
fclose(fp);
```

Sollen bei der Verwendung von `fprintf()` mehrere Zeilen auf einmal geschrieben werden, so müssen diese mittels des Neue-Zeile-Zeichens `\n` getrennt werden. Am Ende des Schreibvorgangs muss die Datei wieder mittels `fclose()` geschlossen werden, damit die Modifikationszeit angepasst wird.

`fputs()` – Einzelne Zeichenketten schreiben

Mit `fputs()` können normale Zeichenketten in eine Datei geschrieben werden. Sonderzeichen in den Zeichenketten sind erlaubt, ein Ersetzen von Platzhaltern durch Werte von Variablen hingegen nicht.

`fputc()` – Einzelne Zeichen schreiben

Daten aus einer Datei einlesen

Auch die Funktionen zum Einlesen von Daten aus einer Datei ähneln denen im Abschnitt *Ausgabe und Eingabe* beschriebenen Funktionen zum Einlesen von Daten vom Bildschirm.

`fgetc()` – Daten zeichenweise einlesen

Die Funktion `fgetc()` liest ein einzelnes Zeichen aus einer Datei ein und gibt es als `int`-Wert zurück. Vor Verwendung von `fgetc()` muss wiederum zunächst ein File-Pointer mittels *fopen()* bereitgestellt werden:

Die Funktion `fgetc()` wird häufig in Verbindung mit einer `while`-Schleife eingesetzt, wobei als Abbruchfunktion die Funktion `feof()` genutzt wird: Diese prüft, ob der angegebene File-Pointer auf das Ende der Datei zeigt und gibt in diesem Fall einen Wert ungleich Null zurück.

Interaktionen mit dem Betriebsystem

`system()` – Externe Programme aufrufen

Mittels der Funktion `system()` aus der Standard-Bibliothek *stdlib.h* können Programme des Betriebsystems, beispielsweise Shell-Programme, aus einem C-Programm heraus aufgerufen werden. Als Argument wird der Funktion dabei eine Zeichenkette übergeben, die den Namen des aufzurufenden Programms mitsamt aller Argumente und Optionen enthält, beispielsweise `"ls -lh"`:

```c
#include <stdlib.h>

// ...

system("ls -lh");

// ...
```

Wenn das externe Programm beendet ist, wird das C-Programm weiter ausgeführt.

`exit()` und `atexit()` – Programme ordentlich beenden

Mittels der Funktion `exit()` kann ein Programm in geordneter Weise beendet werden. Als Argument wird beim Aufruf der Funktion ein `int`-Wert angegeben, der als Rückgabewert an das System dient. Der Wert `0` gilt dabei für ein normales Programmende, der Wert `1` wird üblicherweise im Falle eines Fehlers zurück gegeben.

Trifft das Programm auf eine `exit()`-Funktion, so werden automatisch alle noch nicht geschriebenen Ausgabe-Streams geschrieben, alle offenen Dateien geschlossen sowie alle mittels `tmpfile()` angelegten temporären Dateien gelöscht.

Zusätzlich können im vorangehenden Teil des Codes, häufig in der Funktion `main()`, mittels `atexit()` Pointer auf Funktionen angegeben werden, die bei einem Aufruf von `exit()` ausgeführt werden, bevor das Programm beendet wird. Das Besondere dabei ist, dass die Funktionen von hinten nach vorne durchlaufen werden, d.h. die zuletzt angegebene `atexit()`-Funktion wird als erstes aufgerufen, die als erstes angegebene `atexit()`-Funktion zuletzt.

Modularisierung

Jedes umfangreichere Programm wird normalerweise in mehrere Dateien („Module“) aufgeteilt. In einem Modul werden zusammengehörige Datenstrukturen und Funktionen zu einer logischen Einheit kombiniert.

Jedes Modul besitzt eine Schnittstelle mit „globalen“ Variablen und Funktionen und Variablen des Moduls, auf die auch von einer anderen Datei aus zugegriffen werden kann. Die anderen Funktionen und Variablen sind „lokal“, sie haben also keine direkten Auswirkungen auf andere Module.

Jedes Modul sollte möglichst wenig Funktionen oder Variablen in seiner Schnittstelle definieren, damit Änderungen an lokalen Funktionen keine Änderungen in anderen Code-Teilen zur Folge haben. Beispielsweise betrifft die Änderung einer globalen Funktionen bezüglich ihres Namens oder ihrer Anzahl an Argumenten alle Code-Teile, in denen die Funktion benutzt wird.

Die Schnittstelle eines Moduls (einer `.c`-Datei) wird üblicherweise in einer gleichnamigen Headerdatei (einer `.h`-Datei) definiert. In einer solchen Datei werden Variablen und Funktionen lediglich deklariert, eine Header-Datei enthält somit keinen ausführbaren Code.

Die Verwendung von Header-Dateien ist dann sinnvoll, wenn eine Variable oder eine Funktion von mehreren Dateien aus benutzt werden soll.

Präprozessor, Compiler und Linker

Ein klassischer C-Compiler besteht aus drei Teilen: Einem Präprozessor, dem eigentlichen Compiler, und einem Linker:

- Der Präprozessor bereitet einerseits den Quellcode vor (entfernt beispielsweise Kommentare und Leerzeilen); andererseits kann er mittels der im nächsten Abschnitt näher beschriebenen Präprozessor-Anweisungen Ersetzungen im Quellcode vornehmen.
- Der Compiler analysiert den Quellcode auf lexikalische oder syntaktische Fehler, nimmt gegebenenfalls Optimierungen vor und wandelt schließlich die aufbereiteten Quellcode-Dateien in binäre Objekt-Dateien (Endung: `.o`) um.
- Der Linker ergänzt die Objekt-Dateien um verwendete Bibliotheken und setzt die einzelnen Komponenten zu einem ausführbaren Gesamt-Programm zusammen.

Präprozessor-Anweisungen

Der Präprozessor lässt sich im Wesentlichen durch zwei Anweisungen steuern, die jeweils durch ein Hash-Symbol `#` zu Beginn der Anweisung gekennzeichnet sind und ohne einen Strichpunkt abgeschlossen werden:

`#include` – Einbinden von Header-Dateien

Mittels `#include` können weitere Quellcode-Teile in das Programm integriert werden. Diese Dateien werden vom Präprozessor eingelesen und an Stelle der `#include`-Anweisung in die Datei geschrieben.

Unterschieden wird bei `#include`-Anweisungen zwischen Bibliotheken, die sich in einem Standardpfad im System befinden und dem Compiler bekannt sind, und lokalen *Header-Dateien*, die sich üblicherweise im gleichen Verzeichnis befinden. Die Bibliotheken aus dem Standard-Pfad erhalten um ihren Namen eckige Klammern, die Namen der lokalen Header-Dateien werden in doppelte Anführungszeichen gesetzt:

```
// Standard-Bibliothek stdio.h importieren:
#include <stdio.h>

// Lokale Header-Datei input.h importieren:
#include "input.h"
```

`#define` – Definition von Konstanten und Makros

Mittels `#define` können Konstanten oder Makros definiert werden. Bei der Definition einer Konstanten wird zunächst der zu ersetzende Name anschließend der zugehörige Wert angegeben:

```
#define HALLO "Hallo Welt!"
#define PI 3.1415
```

Eine Großschreibung der Konstantennamen ist nicht zwingend nötig, ist in der Praxis jedoch zum Standard geworden, um Konstanten- von Variablennamen unterscheiden zu können. Nicht verwendet werden dürfen allerdings folgende Konstanten, die im Präprozessor bereits vordefiniert sind:

- `__LINE__`: Ganzzahl-Wert der aktuellen Zeilennummer
- `__FILE__`: Zeichenkette mit dem Namen der kompilierten Datei
- `__DATE__`: Zeichenkette mit aktuellem Datum
- `__TIME__`: Zeichenkette mit aktueller Uhrzeit

Eine Festlegung mittels `#define` bleibt allgemein bis zum Ende der Quelldatei bestehen. Soll eine erneute Definition einer Konstanten `NAME` erfolgen, so muss die bestehende Definition erst mittels `#undef NAME` rückgängig gemacht werden.

Bei der Definition eines Makros mittels `#define` wird zunächst der Name des Makros angegeben. In runden Klammern stehen dann, wie bei der Definition einer *Funktion*, die Argumente, die das Makro beim Aufruf erwartet.[1] Unmittelbar anschließend wird der Code angegeben, den das Makro ausführen soll.

```
#define QUADRAT(x) ((x)*(x))
```

Bei der Definition von Makros muss beachtet werden, dass der Präprozessor die Ersetzungen nicht wie ein Taschenrechner oder Interpreter, sondern wie ein klassischer Text-Editor vornimmt. Steht im Quellcode beispielsweise die Zeile `result = QUADRAT(n)`, so wird diese durch den Präprozessor gemäß dem obigen Makro zu `result = ((n)*(n))` erweitert. In diesem Fall erscheinen die Klammern als unnötig. Steht allerdings im Quellcode die Zeile `result = QUADRAT(n+1)`, so wird diese mit Hilfe der Klammern zu `((n+1)*(n+1))` erweitert. Ohne die zusätzlichen Klammern in der Makro-Definition würde der Ausdruck zu `n+1*n+1` erweitert werden, was ein falsches Ergebnis liefern würde.

Innerhalb von Makro-Definitionen kann ein spezieller Operatoren verwendet werden: Der Operator `#` kann auf einen Argumentnamen angewendet werden und setzt den Namen der konkret angegebenen Variablen in doppelte Anführungszeichen:[2]

[1] Zu beachten ist, dass bei der Definition eines Makros kein Leerzeichen zwischen dem Makronamen und der öffnenden runden Klammer der Argumentenliste vorkommen darf. Der Präprozessor würde ansonsten den Makronamen als Namen einer Konstanten interpretieren und den geamten Rest der Zeile als Wert dieser Konstanten interpretieren.

[2] Zudem können mit dem zweiten möglichen Makro-Operator `##` die Namen von zwei oder mehreren übergebenen Argumenten zu einer neuen Bezeichnung verbunden werden. Dieser Operator wird allerdings nur sehr selten eingesetzt.

```
#define QUADRAT(x) print("Der Quadrat-Wert von %s ist %i.\n", #x, (x)*(x))
```

Ein Minimalbeispiel für dieses Makro könnte folgendermaßen aussehen:

```
// Datei: makro-beispiel-1
// Compilieren: gcc -o makro-beispiel-1 makro-beispiel-1
// Aufruf: ./makro-beispiel-1

// Ergebnis beim Aufruf: Der Quadrat-Wert von num ist 121.

#include <stdio.h>

#define QUADRAT(x) printf("Der Quadrat-Wert von %s ist %i.\n", #x, (x)*(x))

void main()
{
    int num=11;

    QUADRAT(num);
}
```

Ist eine `#define`-Anweisung zu lange für eine einzelne Code-Zeile, so kann die Anweisung an einer Whitespace-Stelle mittels \ unterbrochen und in der nächsten Zeile fortgesetzt werden. Eventuelle Einrückungen (Leerzeichen, Tabulatoren) werden dabei vom Präprozessor automatisch entfernt.

Ein entscheidender Vorteil von `#define`-Anweisungen ist, dass so definierte Konstanten oder Makros an beliebigen Stellen im Code eingesetzt werden können und zugleich bei Bedarf nur an einer einzigen Stelle im Programm geändert werden müssen.

`#if`, `#ifdef`, `#ifndef` – Bedingte Compilierung

Mittels `#if`, `#ifdef` oder `#ifndef` können Teile einer Datei zur „bedingten Compilierung“ vorgemerkt werden. Ein solcher Code-Teil wird nur dann vom Compiler berücksichtigt, wenn die angegebene Bedingung erfüllt ist.

Beispielsweise kann auf diese verhindert werden, dass Header-Dateien oder Quellcode-Bibliotheken mehrfach geladen werden. Beispielsweise kann man in einer Header-Datei `input.h` gleich zu Beginn prüfen, ob eine Konstante `INPUT_H` definiert ist. Falls nicht, so kann wird der folgende Code berücksichtigt, wobei darin auch die Konstante `INPUT_H` mit dem Wert `1` definiert wird:

```
// Datei: input.h

#ifndef INPUT_H

#define INPUT_H = 1

// ... eigentlicher Inhalt ...
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
//#endif
```

Die Variable `INPUT_H` ist nur beim ersten Versuch, die Datei mittels `#include` zu importieren, nicht definiert. Ein mehrfaches Importieren wird somit verhindert. Ebenso kann beispielsweise mittels `#ifdef DEBUG` ein Code-Teil nur zu Testzwecken eingefügt werden (der durch eine Zeile `#define DEBUG 1` am Beginn der Datei aktiviert wird). Es kann auch ein Teil eines Codes nur in Abhängigkeit von einer Versionsnummer ausgeführt werden, indem beispielsweise `#if VERSION < 1.0` geprüft wird.

Ob weitere Präprozessor-Anweisungen vom Compiler unterstützt werden, hängt von dessen Version und vom konkreten Betriebsystem ab. Üblicherweise werden daher nur die oben genannten Anweisungen verwendet.

Compiler-Optionen

Der Standard-C-Compiler kann mit einer Vielzahl an Optionen aufgerufen werden, mit denen der Compilier-Ablauf gesteuert werden kann. Möchte man beispielsweise lediglich überprüfen, welche Ersetzungen vom Präprozessor vorgenommen wurden, aber den Quellcode nicht kompilieren, so kann die Option `-E` verwendet werden:

```
gcc -E -o mycode.i mycode.c
```

In diesem Beispiel wird die Ausgabe, die der Präprozessor bei der Verarbeitung der Datei `mycode.c` erzeugt, in die Datei `mycode.i` geschrieben. Mit der Option `-o` („output“) wird bei `gcc` allgemein der Name der Ausgabedatei angegeben.

Verlinken von Bibliotheken

Jeder Compiler bringt mehrere so genannte Bibliotheken („Libraries“) mit sich. Diese enthalten fertige Funktionen in bereits compilierter Form, die von anderen C-Programmen genutzt werden können. Der Linker sucht die benötigten Funktionen aus den Bibliotheken heraus und fügt sie dem zu compilierenden Programm hinzu.

Laufzeiten von Algorithmen

Bisweilen können für die selbe Aufgabe mehrere Lösungen gefunden werden, die sich teilweise jedoch erheblich in ihrer Effizienz unterscheiden. Bei der Effizienz-Analyse eines Algorithmus, also eines „Rezepts“ zur Lösung eines Problems, ist es insbesondere von Interesse, wie sich die Laufzeit in Abhängigkeit von der Anzahl n der zu bearbeitenden Daten ändert.

Zur Analyse von Laufzeiten sind prinzipiell zweierlei Vorgehensweisen möglich:

- Mittels eines *Benchmarks* wird ein Programm oder Algorithmus mit einem möglichst typischen Satz an Daten aufgerufen und dabei die benötigte Zeit gemessen.

  Ein Werkzeug, das hierfür unter Linux genutzt werden kann, ist *gprof*. Dieses Programm misst nicht nur die Laufzeit eines Programms und der im Programmverlauf aufgerufenen Funktionen, sondern zählt auch, wie häufig die einzelnen Funktionen aufgerufen wurden. Damit erhält man einen guten Überblick, welche Funktionen für eine weitergehende Analyse „wichtig“ sind.

- Mit einer *Laufzeit-Analyse* kann anhand der Struktur des Quellcodes, beispielsweise anhand der Anzahl an Schleifendurchläufen, Lese- oder Schreibvorgängen, die Größenordnung der Laufzeit eines Algorithmus in Abhängigkeit von der Anzahl n an zu bearbeiteten Daten abgeschätzt werden.

Die „Big-O“-Notation

Wie lange die Ausführung eines Algorithmus tatsächlich benötigt, hängt nicht zuletzt von der Rechenleistung des Computers ab, auf dem der Code ausgeführt wird; Benchmarks müssen daher auf einem einheitlichen System durchgeführt und unter Angabe der Rechnerleistung (CPU, RAM, usw.) angegeben werden. Allgemeinere Vergleiche sind hingegen möglich, welche die Laufzeit t eines Algorithmus allgemein als Funktion $t(n)$ des Datenumfangs n ausgedrückt wird. Wird beispielsweise im Verlauf eines Programms eine Funktion mit einer konstanten Laufzeit c insgesamt n mal aufgerufen, so ergibt sich dadurch eine Laufzeit von $t(n) = c \cdot n$.

Beim Zählen von Laufzeiten wird üblicherweise die vereinfachende Vereinbarung, dass die folgenden Prozess-Schritte zur Ausführung jeweils eine Zeiteinheit benötigen:

- Jede Wertzuweisung
- Jeder Wertevergleich

- Jede Iteration einer Schleifenvariablen

Finden beispielsweise beim Durchlaufen einer Schleife n Iterationen statt, so nimmt die Laufzeit für einen Aufruf einer solchen Schleife linear mit n zu. Man sagt, dass in diesem Fall die Laufzeit proportional zur Größenordnung von n ist, und schreibt hierfür in Kurzform $\mathcal{O}(n)$. Wird hingegen eine verschachtelte Liste mit n Teillisten durchlaufen, die wiederum n Einträge haben, so sind insgesamt $n \cdot n = n^2$ Iterationen nötig. Entsprechend ergibt sich für einen Aufruf einer derartigen Schleife eine Laufzeit in der Größenordnung von $\mathcal{O}(n^2)$.

Dynamische Datenstrukturen

In C sind nur die in den Abschnitten *Elementare Datentypen* und *Zusammengesetzte Datentypen* beschriebenen Datentypen vordefiniert. Damit können allerdings weitere Datentypen abgeleitet werden, die für manche Einsatzbereiche besser geeignet sind.

Verkettete Listen

Eine verkettete Liste besteht aus einer Vielzahl von Elementen, bei der jedes Element einen Zeiger seinen Nachfolger enthält; bei einer doppelt verketteten Liste besitzt jedes Element zusätzlich einen Zeiger auf seinen Vorgänger. Eine derartige Struktur bietet eine einfache Möglichkeit zusätzliche Elemente in die Liste aufzunehmen oder Elemente wieder aus der Liste zu entfernen. Verkettete Listen können somit dynamisch wachsen oder schrumpfen.

Einfach verkettete Listen

Bei einer einfach verketteten Liste hat jedes Element einen Zeiger, der auf seinen unmittelbaren Nachfolger zeigt; der Zeiger des letzten Elements zeigt auf `NULL`. Verkettete Listen haben stets einen Zeiger, der auf das erste Element („Head“) zeigt, und oftmals auch einen Zeiger auf das letzte Element der Liste („Tail“).

Die einzelnen Elemente einer verketteten Liste haben den Datentyp `struct`. Da sie allerdings bereits bei ihrer Deklaration einen Pointer auf ein weiteres Element mit gleichem Datentyp angeben, muss der Name der Struktur dem Compiler schon im Vorfeld bekannt sein. Man kann dies auf folgendem Weg erreichen:

```
struct element_prototype
{
    // Eigentlicher Inhalt (hier: int):
    int value;

    // Zeiger auf das nächste Element:
    element_prototype * next;
};

typedef element_prototype element_type;
```

Bei dieser Deklarationsform wird der Strukturname, in diesem Fall `element_prototype`, *vor* der eigentlichen Deklaration angegeben. Der Compiler kennt von diesem Moment

an zwar noch nicht die Größe der Struktur, aber zumindest ihren Namen sowie ihren Datentyp, was für die Erstellung eines Pointers bereits genügt. Anschließend kann der Strukturtyp mittels `typedef` umbenannt werden, um im Folgenden anstelle von `struct element_prototype` einfacher `element_type` für die Bezeichnung des Datentyps schreiben zu können.

Um mittels der Element-Struktur eine verkettete Liste zu erstellen, müssen mindestens zwei Elemente definiert werden: Das Head-Element `e0` sowie ein weiteres Element `e1`, das im Fall von nur zwei Einträgen zugleich auch das Schluss-Element ist:

```
// Zeiger auf Elemente deklarieren:
element_type *e0, *e1;

int init_list()
{
    // Dynamischen Speicherplatz für Elemente reservieren:
    e0 = (element_type *) malloc(sizeof *e0);
    e1 = (element_type *) malloc(sizeof *e1);

    // Fehlerkontrolle:
    if (e0 == NULL) || (e1 == NULL)
        return 1;

    // Referenzen anpassen:
    e0->next = e1;
    e1->next = NULL;

    // Normaler Rückgabewert:
    return 0;
}
```

Möchte man ein weiteres Element in die verkettete Liste aufnehmen, so muss einerseits der Speicherplatz für das zusätzliche Element reserviert werden. Andererseits muss der Zeiger des Elements, hinter dem das neue Element eingefügt werden soll, aktualisiert werden:

```
element_type * insert_element_after(element_type *e, int value_new)
{
    // Zeiger auf neues Element deklarieren:
    element_type *e_new

    // Dynamischen Speicherplatz für neues Element reservieren:
    e_new = (element_type *) malloc(sizeof *e_new);

    // Fehlerkontrolle: Kein Speicherplatz verfügbar:
    if (e_new == NULL)
        return NULL;

    // Inhalt des neuen Elements zuweisen:
    e_new->value = value_new;

    // Referenzen anpassen:
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
    e_new->next = e->next;
    e->next = e_new;

    // Zeiger auf neues Element zurückgeben:
    return e_new;
}
```

Der Zeiger des neuen Elements `e_new` muss nach dem Einfügen auf die Stelle verweisen, auf die der Zeiger des Vorgänger-Elements `e` bislang gezeigt hat. Dafür muss der Zeiger des Vorgänger-Elements `e` nach dem Einfügen auf das neue Element `e_new` verweisen.

Um das Nachfolger-Element eines bestimmten Element aus einer einfach verketteten Liste zu entfernen, muss einerseits der Zeiger des dieses Elements auf das übernächste Element umgelenkt werden; andererseits muss der dynamisch reservierte Speicherplatz für das zu entfernende Element wieder freigegeben werden:

```
int delete_element_after(element_type *e)
{
    // Fehlerkontrolle (e letztes Element der Liste):
    if (e->next == NULL)
        return 1;

    // Referenzen anpassen:
    e->next = e->next->next;

    // Speicherplatz freigeben:
    free(e->next);

    // Normaler Rückgabewert:
    return 0;
}
```

Soll nicht das Nachfolger-Element eines angegebenen Elements, sondern dieses selbst gelöscht werden, so muss zuerst der Vorgänger des Elements ermittelt werden. Dies kann man erreichen, indem man vom Head-Element aus die Zeigerwerte der einzelnen Elemente mit dem Zeigerwert des angegebenen Elements vergleicht:

```
element_type * find_previous_element(element_type *e)
{
    // Temporären und Vorgänger-Zeiger deklarieren:
    element_type *e_pos;
    element_type *e_prev;

    // Temporären Zeiger auf Head-Element setzen:
    e_pos = e0;

    // Temporären Zeiger mit Zeigern der Listenelemente vergleichen:
    while ( (e_pos != NULL) && (e_pos != e) )
    {
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
        e_prev = e_pos;         // Zeiger auf bisheriges Element␣
↪zwischenspeichern
        e_pos  = e_pos->next;   // Temporären Zeiger iterieren
    }

    // Die while-Schleife wird beendet, wenn die Liste komplett durchlaufen
    // oder das angegebene Element gefunden wurde; in letzterem Fall zeigt
    // e_pos auf das angegebene Element, e_prev auf dessen Vorgänger.

    // Fall 1: Liste wurde erfolglos durchlaufen (Element e nicht in Liste):
    if ( (e_pos == NULL) && (e_prev != e) )
        return NULL;

    // Fall 2: Element e ist erstes Element der Liste:
    else if (e_pos == e0)
        return NULL;

    // Fall 3: Element e0 wurde an anderer Stelle gefunden:
    else
        return e_prev;
}
```

Das Löschen eines Elements kann mit Hilfe der obigen Funktion beispielsweise folgendermaßen implementiert werden:

```
int delete_element(element_type *e)
{
    // Vorgänger-Zeiger deklarieren:
    element_type *e_prev;

    // Position des Vorgänger-Elements bestimmen:
    e_prev = find_previous_element(e)

    // Fehlerkontrolle: Element e nicht in Liste:
    if ( (e_prev == NULL) && e != e0)
        return 1;

    // Angegebenes Element wurde gefunden:

    if (e == e0)        // Angegebenes Element ist erstes Element der Liste
    {
        e0 = e0->next;      // Neues Head-Element festlegen
    }
    else                    // Angegebenes Element ist nicht erstes Element
    {
        e_prev->next = e->next; // Vorgänger-Element mit
    }                               // Nachfolger-Element verketten

    // Speicherplatz freigeben:
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
    free(e);

    // Normaler Rückgabewert:
    return 0;
}
```

Offensichtlich ist das Löschen eines bestimmten Elements bei einfach verketteten Listen mit einigem Rechenaufwand verbunden, da im ungünstigsten Fall die gesamte Liste durchlaufen werden muss. Das Suchen nach einem bestimmten Wert in der Liste funktioniert auf ähnliche Weise:

```
element_type * search_content(int value)
{
    // Temporären Zeiger definieren:
    element_type *e_pos = e0;

    // Wert des Elements e_pos mit angegebenem Wert vergleichen:
    while ( (e_pos->value != value) &&  (e_pos != NULL) )
    {
        e_pos  = e_pos->next;   // Temporären Zeiger iterieren
    }

    // Die while-Schleife wird entweder beendet, wenn die Liste komplett
    // durchlaufen oder der angegebene Wert gefunden wurde; in ersten Fall ist
    // e_pos gleich NULL, im zweiten Fall zeigt e_pos auf das entsprechende
    // Element.

    return e_pos;
}
```

Auch beim Suchen eines bestimmten Werts muss die verkettete Liste im ungünstigsten Fall komplett durchlaufen werden. Um eine verlinkte Liste wieder zu löschen, werden nacheinander die einzelnen Elemente mittels `free()` wieder freigegeben:

```
void delete_list()
{
    // Temporäre Zeiger definieren:
    element_type *e_pos;
    element_type *e_tmp;

    // Temporären Zeiger auf Head-Element setzen:
    e_pos = e0;

    // Alle Elemente der Liste durchlaufen:
    while ( e_pos != NULL )
    {
        e_tmp = e_pos->next;
        free(e_pos);
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
        e_pos = tmp;
    }
```

Doppelt verkettete Listen

Enthält jedes jedes Element einer verketteten Liste nicht nur einen Zeiger auf seinen Nachfolger, sondern ebenso einen Zeiger auf seinen Vorgänger, so spricht man von einer doppelt verketteten Liste. Die Deklaration eines Listenelements sowie die Erzeugung einer Liste ist im Wesentlichen mit der einer einfach verketteten Liste identisch:

```
struct element_prototype
{
    // Eigentlicher Inhalt (hier: int):
    int value;

    // Zeiger auf das vorheriges und nächste Element:
    element_prototype * prev;
    element_prototype * next;
};

typedef element_prototype element_type;
```

```
// Zeiger auf Elemente deklarieren:
element_type *e0, *e1;

int init_list()
{
    // Dynamischen Speicherplatz für Elemente reservieren:
    e0 = (element_type *) malloc(sizeof *e0);
    e1 = (element_type *) malloc(sizeof *e1);

    // Fehlerkontrolle:
    if (e0 == NULL) || (e1 == NULL)
        return 1;

    // Referenzen anpassen:
    e0->prev = NULL;
    e0->next = e1;

    e1->prev = e0;
    e1->next = NULL;

    // Normaler Rückgabewert:
    return 0;
}
```

Ein Vorteil von doppelt verketteten Listen liegt darin, dass man sowohl vor- als auch rückwärts in der Liste nach Inhalten suchen kann. Ebenso kann man – im Vergleich zu

einfach verketteten Listen – ein bestimmtes Listenelement mit weniger Aufwand an einer bestimmten Stelle einfügen oder löschen.

Hilfreiche Werkzeuge

Im folgenden werden kurz einige Programme beschrieben, die bei der Entwicklung von C-Programmen hilfreich sein können. Bei den meisten Linux-Systemen (Debian, Ubuntu, Linux Mint) lassen sich diese unmittelbar mittels apt installieren:

```
aptitude install astyle cdecl cflow doxygen gdb graphviz splint valgrind
```

Anschließend können die jeweiligen Programme mittels einer Shell im Projekt-Verzeichnis aufgerufen beziehungsweise auf Quellcode-Dateien angewendet werden.

`astyle` – Code-Beautifier

Das Programm `astyle` kann verwendet werden, um C-Code in eine einheitliche Form zu bringen. Die Syntax dafür lautet:

```
astyle option < sourcefile > output_file
```

Als Option kann mittels `-A1` bis `-A12` ein gewünschter Code-Style angegeben werden. Eine Übersicht über die möglichen Style-Varianten ist in der Dokumentation des Programms zu finden. In den Beispielen dieses Tutorials wird der Codestyle „Allman“ (Option `-A1`) verwendet.

Um beispielsweise alle `c`-Dateien eines Verzeichnisses mittels `astyle` in den gewünschten Code-Style zu bringen, kann folgendes Mini-Skript verwendet werden (die existierenden Dateien werden dabei ueberschrieben, bei Bedarf vorher Sicherheitskopie anlegen!):

```
for i in *.c ; \
do astyle -A1 < $i > $(basename $i).tmp && mv $(basename $i).tmp $i; \
done
```

`cdecl` – Deklarations-Übersetzer

Das Programm `cdecl` kann verwendet werden, um komplexe Deklarationen, auf die man beispielsweise beim Lesen von Quellcode stoßen kann, in einfachem Englisch zu beschreiben. Umgekehrt kann man durch die Angabe eines Strukturtyps in entsprechender Englisch-Syntax die entsprechende C-Deklaration zu erhalten.

Üblicherweise wird `cdecl` mittels der Option `-i` im interaktiven Modus gestartet:

```
cdecl -i
```

Anschließend kann durch Eingabe von `explain` und einer beliebigen C-Deklaration diese in einfachem Englisch angezeigt werden, beispielsweise liefert `explain int myfunc(int, char *);` als Ergebnis: `declare myfunc as function (int, pointer to char) returning int`. Umgekehrt kann `declare` in Verbindung mit einer solchen Englisch-Syntax aufgerufen werden, um C-Code zu erhalten, beispielsweise liefert `declare mylist as array 20 of pointer to char` das Ergebnis `char *mylist[20]`.

Mit `help` kann Hilfe angezeigt werden, mit `quit` wird `cdecl` wieder beendet.

`cflow` – Funktionsstruktur-Viewer

Mittels `cflow` kann angezeigt werden, welche Funktionen schrittweise von einer Quelldatei aufgerufen werden, und falls es sich um externe Funktionen handelt, in welcher Datei und an welcher Stelle sich diese befinden.

Die Syntax von `cflow` lautet:

```
cflow quelldatei.c
```

`doxygen` – Dokumentations-Generator

Mittels `doxygen` kann eine Dokumentation eines C-Projekts erzeugt werden, ohne dass innerhalb der Code-Dateien irgendeine Markup-Sprache verwendet werden muss. Dafür werden beispielsweise Übersichts- und Strukturdiagramme automatisch erzeugt, sofern auch das Programm `graphviz` installiert ist.

Um eine Dokumentation mit Doxygen zu erstellen, wechselt man in das Projektverzeichnis und gibt `doxygen -g Doxyfile` ein, um eine Konfigurationsdatei (üblicherweise: `Doxyfile`) zu generieren. Die erzeugte Beispieldatei ist anhand vieler Kommentare weitgehend selbst erklärend und kann einfach mit einem Texteditor bearbeitet werden; unnötige Kommentare oder Optionen können dabei zur besseren Übersicht gelöscht werden. Alternativ kann man eine leere `Doxyfile` erzeugen und darin wichtige Optionen aktivieren.

Möchte man die von `doxygen` erstellte Dokumentation in einem eigenen Ordner abgelegt haben, so sollte man zudem beispielsweise mittels `mkdir doxygen` im Projektverzeichnis einen neuen Unterordner erstellen.

Als Optionen zur Erzeugung von C-Code-Übersichten halte ich für sinnvoll:

Option in der Doxyfile	Beschreibung
`PROJECT_NAME = Toolname`	Namen des Projekts angeben
`OUTPUT_DIRECTORY = ./doxygen`	Verzeichnis für HTML- und LaTeX-Dokumentation festlegen
`OUTPUT_LANGUAGE = German`	Sprache auswählen
`EXTRACT_ALL = YES`	Alle Informationen des Quellcodes verwenden
`SOURCE_BROWSER = YES`	Immer Links zu den entsprechenden Funktionen und Dateien erzeugen
`HAVE_DOT = YES`	Nützliche Aufrufdiagramme mittels `graphviz` erzeugen
`CALL_GRAPH = YES`	Funktionsaufrufe als Graphen erzeugen
`CALLER_GRAPH = YES`	Als Graphen darstellen, von wo aus die einzelnen Funktionen aufgerufen werden
`FILE_PATTERNS = *.c *.h`	Alle `.c` und `.h`-Dateien berücksichtigen

Nach dem Anpassen der `Doxyfile` muss im Projektpfad nur `doxygen` ohne weiteren Argumente aufgerufen werden, um die Dokumentation zu erstellen und im `doxygen`-Unterverzeichnis abzulegen. Anschließend kann man die Indexdatei `./doxygen/html/index.html` mit Firefox oder einem anderen Webbrowser öffnen.

gdb – Debugger

Fehler übersieht man gerne. Bei der Fehlersuche in C-Code kann der Debugger `gdb` eingesetzt werden, um das Verhalten eines Programms schrittweise zu überprüfen sowie Teile des Quellcodes, die als Fehlerquelle in Frage kommen, näher eingrenzen zu können.

Um den `gdb`-Debugger nutzen zu können, muss das zu untersuchende Programm mit der Option `-g` oder `-ggdb` compiliert werden, um für den Debugger relevante Informationen zu generieren.

```
# Compilieren zu Debug-Zwecken:
gcc -ggdb -o myprogram myprogram.c
```

Die Option `-ggdb` erzeugt ausführlichere, auf `gdb` zugeschnittene Informationen und dürfte in den meisten Fällen zu bevorzugen sein.

Anschließend kann das compilierte Programm mit `gdb` geladen werden:[1]

```
gdb myprogram
```

Der Debugger wird dabei im interaktiven Modus gestartet. Um das angegebene Programm `myprogram` zu starten, kann `run` (oder kurz: `r`) eingegeben werden; dabei können dem Programm mittels `run arg_1 arg_2 ...` beliebig viele Argumente übergeben werden, als ob der Aufruf aus der Shell heraus erfolgen würde. Das Programm kann dabei abstürzen,

[1] Alternativ kann man `gdb` auch ohne Angabe eines Programmnamens starten und dieses im interaktiven Modus mittels `file myprogram` öffnen.

wobei eine entsprechende Fehlermeldung und die für den Absturz relevante Code-Zeile angezeigt wird, oder (anscheinend) fehlerfrei durchlaufen.

Wird ein Fehler angezeigt, beispielsweise eine „Arithmetic exception“, wenn versucht wird durch Null zu dividieren, so kann mittels `print varname` der Wert der angegebenen Variable zu diesem Zeitpunkt ausgegeben werden.

Verwendung von Breakpoints

Um sich den Programmablauf im Detail anzuschauen, können mit `break` (oder kurz: `b`) so genannte „Breakpoints“ gesetzt werden. An diesen Stellen stoppt das Programm, wenn es mit `run` gestartet wird, automatisch. Die Breakpoints werden von `gdb` automatisch ausgewählt, beispielsweise werden sie vor Funktionsaufrufen gesetzt, um mittels `print` die Werte der übergebenen Variablen prüfen zu können.

Mittels eines Aufrufs von `break num` kann auch eine weiterer Breakpoint unmittelbar vor der Code-Zeile `num` manuell gesetzt werden. Ist in dem Programm eine Funktion `myfunc()` definiert, so werden mittels `break myfunc` Breakpoints vor jeder Stelle gesetzt, an denen die angegebene Funktion aufgerufen wird.

Ist man nach dem Setzen der Breakpoints und dem Aufruf von `run` am ersten Breakpoint angekommen, so kann man mittels `continue` (oder kurz: `c`) bis zum nächsten Breakpoint mit der Ausführung des Programms fortfahren. Alternativ kann `next` (oder kurz: `n`) beziehungsweise `step` (oder kurz: `s`) eingegeben werden, um nur die unmittelbar nächste Quellcode-Zeile auszuführen. Der Unterschied zwischen `next` und `step` liegt darin, dass `next` die nächste Code-Zeile als eine einzige Anweisung ausführt, während `step` im Falle eines Funktionsaufrufs den Code der Funktion zeilenweise durchläuft.

Drückt man in `gdb` die Enter-Taste, so wird die unmittelbar vorher gegebene Anweisung erneut ausgeführt. Dies kann insbesondere in Verbindung mit `next` oder `step` viel Schreibarbeit ersparen.. ;-)

Werte von Variablen beobachten

Ebenso wie Breakpoints die Ausführung des Programms an bestimmten Code-Zeilen gezielt unterbrechen, kann man mit so genannten „Watchpoints“ das Programm jedes mal automatisch stoppen, wenn sich der Wert einer angegebenen Variablen ändert. Befindet sich beispielsweise im Programm eine Variable `myvar`, so kann mittels `watch myvar` ein zu dieser Variablen passender Watchpoint definiert werden.

Backtraces

Wird eine Funktion aufgerufen, so erzeugt `gdb` einen so genannten „frame“, in dem der Funktionsname und die übergebenen Argumente festgehalten werden, beispielsweise existiert immer ein Frame für die Funktion `main`, der gegebenenfalls die beim Aufruf übergebenen Argumente `argv` sowie ihre Anzahl `argc` beinhaltet. Mit jedem Aufruf einer weiteren Funktion wird, solange deren Ausführung dauert, ein weiterer Frame angelegt.

Tritt ein Fehler auf, so genügt es unter Umständen, wenn die Zeile des Codes angezeigt wird, die den Fehler verursacht hat. Mitunter ist es jedoch auch gut zu wissen, wie das Programm zur fehlerhaften Zeile gelangt ist. Dies kann in `gdb` mittels einer Eingabe von `backtrace` (oder kurz: `bt`) geprüft werden. Ein solcher Backtrace gibt in umgekehrter Reihenfolge an, durch welche Funktionsaufruf das Programm an die Fehlerstelle gelangt ist. Somit können beim nächsten Durchlauf von `gdb` gezielt Breakpoints gesetzt bzw. Variablenwerte überprüft werden.

In sehr verschachtelten Programmen können mittels `backtrace n` nur die „inneren“ n Frames um die Fehlerstelle herum angezeigt werden, mittels `backtrace -n` die n äußeren Frames.

`ddd` als graphisches Frontend für `gdb`

Möchte man `gdb` mit einer graphischen Oberfläche nutzen, so können optional die Pakete `ddd` und `xterm` via apt installiert werden:

```
sudo aptitude install ddd xterm
```

Anschließend kann man `ddd` als Debugger-Frontend aufrufen.

`gprof` – Profiler

Der Profiler `gprof` kann verwendet werden, um zu untersuchen, wie häufig die einzelnen Funktionen eines Programms aufgerufen werden und wie viel Zeit sie dabei für ihre Ausführung benötigen. Dies soll kurz anhand des folgenden Beispielprogramms gezeigt werden:

```
// Datei: gprof_test.c

#include<stdio.h>

void new_func1(void);

void func_1(void)
{
    int i;
    printf("\n Now: Inside func_1 \n");

     for(i=0; i<1000000000; i++)
         ;

     return;
}

static void func_2(void)
{
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
    int i;
    printf("\n Now: Inside func_2 \n");

    for(i=0 ;i<2000000000; i++)
        ;

    return;
}

int main(void)
{
    int i;
    printf("\n Now: Inside main()\n");

    for(i=0; i<10000000; i++)
        ;

    func_1();
    func_2();

    return 0;
}
```

Um `gprof` nutzen zu können, muss als erstes das zu untersuchende Programm zunächst mit der Option `-pg` compiliert werden, um für den Profiler relevante Informationen zu generieren; als zweites muss das Programm einmal aufgerufen werden, um die für `gprof` relevante Datei `gmon.out` zu erzeugen:

```
gcc -o gprof_test -pg gprof_test.c

./gprof_test
```

Anschließend kann der Profiler mittels `gprof ./gprof_test` aufgerufen werden. Ruft man `gprof` allerdings ohne zusätzliche Optionen auf, so wird eine ziemlich lange Ausgabe auf dem Bildschirm erzeugt, wobei die meisten beschreibenden Kommentare in den Regel nicht benötigt werden; `gprof` sollte daher mit der Option `-b` aufgerufen werden, um die ausführlichen Kommentare auszublenden. Verwendet man zusätzlich die Option `-p`, so wird die Ausgabe auf ein Minimum reduziert:

```
gprof -b -p ./gprof_test

# Ergebnis:
# Flat profile:
#
# Each sample counts as 0.01 seconds.
#   %   cumulative   self              self     total
#  time   seconds   seconds    calls   s/call   s/call  name
#  67.28      4.89     4.89        1     4.89     4.89  func_2
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
#  33.71      7.34     2.45        1     2.45     2.45  func_1
#   0.28      7.36     0.02                             main
```

Bei dieser Ausgabe sieht man auf den ersten Blick, welche Funktion im Laufe des Programms am meisten Zeit benötigt beziehungsweise wie viel Zeit sie je Aufruf braucht. Wird anstelle der Option `-p` die Option `-P` verwendet, so wird neben dieser Aufgliederung angezeigt, an welcher Stelle eine Funktion aufgerufen wird:

```
gprof -b -P ./gprof_test

# Ergebnis:
#                     Call graph
#
#
# granularity: each sample hit covers 2 byte(s) for 0.14% of 7.36 seconds
#
# index % time    self  children    called     name
#                                                 <spontaneous>
# [1]    100.0    0.02    7.34                 main [1]
#                 4.89    0.00       1/1           func_2 [2]
#                 2.45    0.00       1/1           func_1 [3]
# -----------------------------------------------
#                 4.89    0.00       1/1           main [1]
# [2]     66.4    4.89    0.00       1         func_2 [2]
# -----------------------------------------------
#                 2.45    0.00       1/1           main [1]
# [3]     33.3    2.45    0.00       1         func_1 [3]
# -----------------------------------------------
#
#
# Index by function name
#
#    [3] func_1                  [2] func_2                  [1] main
```

Unmittelbar im Anschluss an die Optionen `-p` oder `-P` kann auch ein Funktionsname ausgegeben werden, um die Ausgabe von `gprof` auf die angegebene Funktion zu beschränken; zudem kann mittels der Option `-a` die Aufgabe auf alle nicht als statisch (privat) deklarierten Funktionen beschränkt werden.

`make` – Compilier-Hilfe

Das Shell-Programm `make` ist ein äußert praktisches Hilfsmittel beim Compilieren von C-Quellcode zu fertigen Programmen. Die grundlegende Funktionsweise von `make` ist unter Linux und Open Source: Makefiles beschrieben.

splint – Syntax Checker

Wendet man den Syntax-Prüfer `lint` oder die verbesserte Variante `splint` auf eine C-Datei an, so reklamiert dieser nicht nur Fehler, sondern auch Stilmängel.

```
splint quelldatei.c
```

Bisweilen kann `splint` auch Code-Zeilen beanstanden, in denen man bewusst gegen einzelne „Regeln“ verstoßen hat. In diesem Fall muss man das Ergebnis der Syntax-Prüfung selbst interpretieren und/oder gegebenenfalls Warnungen mittels der jeweiligen Option abschalten (diese wird bei der Ausgabe von `splint` gleich als Möglichkeit mit angegeben).

time – Timer

Der Timer `time` kann verwendet werden, um die Laufzeit eines Programms zu messen. Dies ist nützlich, um verschiedene Algorithmen hinsichtlich ihrer Effizienz zu vergleichen. Als Beispiel soll die Laufzeit zweier Algorithmen verglichen werden, welche alle Primzahlen zwischen 1 und 10000 bestimmen sollen:

```
// Datei: prim1.c
// (Ein nicht sehr effizienter Algorithmus)

#include <stdio.h>

#define N 10000

int main()
{
    int num, factor;
    int is_prim;

    for(num = 2; num <= N; num++)               // Alle Zahlen testen
    {
        is_prim = 0;                            // Vermutung: keine Primzahl

        for(factor = 2; factor < N; factor++)   // Alle möglichen Faktoren␣
↪ausprobieren
        {
            if (num % factor == 0)              // Test, ob num den Faktor␣
↪factor enthält
            {
                if(num == factor)               // num ist genau dann Primzahl,␣
↪wenn sie
                    is_prim = 1;                // nur sich selbst als Faktor␣
↪enthält
                else
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
                    break;                          // sonst nicht
            }
        }

        if (is_prim == 1)                           // Wenn num Primzahl ist,
           printf("%d ", num);                      // dann Ausgabe auf Bildschirm
    }

    printf("\n");
    return 0;
}
```

Übersetzt man dieses Programm mittels `gcc -o prim1 prim1.c` und ruft anschließend `time ./prim1` auf, so erhält man (neben der Auflistung der Primzahlen) folgende Ausgabe:

```
gcc -o prim1.c && time ./prim1

# Ergebnis:
# ...
# real      0m0.179s
# user      0m0.175s
# sys       0m0.003s
```

Die Ausgabe besagt, dass das Programm zur Ausführung insgesamt $0,179\,\mathrm{s}$ benötigt hat, wobei die zur Ausführung von Benutzer- und Systemanweisungen benötigten Zeiten getrennt aufgelistet werden. Beide zusammen ergeben (von Rundungsfehlern abgesehen) die Gesamtzeit.

Im Vergleich dazu soll ein zweiter, wesentlich effizienterer Algorithmus getestet werden:[2]

```
// Datei: prim2.c
// (Ein wesentlich effizienterer Algorithmus)
// ("Das Sieb des Eratosthenes")

#include <stdio.h>

#define N 10000

int main()
{
    int num = 1;
    int factor_1, factor_2;
    int numbers[N];
```

(continues on next page)

[2] Eratosthenes entwickelte ein einfaches Schema zur Bestimmung aller Primzahlen kleiner als 100: Zunächst schrieb er die Zahlen in zehn Zeilen mit je zehn Zahlen auf ein Blatt. Anschließend strich er zunächst alle geraden Zahlen (jede jede zweite) durch, dann alle durch 3 teilbaren Zahlen (also jede dritte), dann alle durch 5 teilbaren Zahlen (die 4 war ja bereits durchgestrichen), usw. Alle verbleibenden Zahlen mussten Primzahlen sein, denn sie waren nicht als Vielfache einer anderen Zahl darstellbar.

(Fortsetzung der vorherigen Seite)

```
    for (numbers[1] = 1; num < N; num++)        // Alle Zahlen zunächst
        numbers[num] = 1;                       // als Primzahlen vermuten

    for (factor_1 = 2; factor_1 < N/2; factor_1++)
    {
        for (factor_2 = 2; factor_2 <= N / factor_1; factor_2++)
        {
            numbers[factor_1 * factor_2] = 0;   // Alle möglichen Produkte
        }                                       // aus factor_1 und factor_2
                                                // sind keine Primzahlen
    }

    for (num = 1; num <= N; num++)
    {
        if (numbers[num] == 1)                  // Jede verbleibende Zahl 1
        {                                       // entspricht einer Primzahl
            printf("%d ", num);                 // Alle Primzahlen ausgeben
        }
    }
    printf("\n");
    return 0;
}
```

In diesem Fall liefert `time` nach dem Compilieren folgendes Ergebnis:

```
gcc -o prim1.c && time ./prim1

# Ergebnis:
# ...
# real      0m0.003s
# user      0m0.002s
# sys       0m0.001s
```

Der zweite Algorithmus gibt das gleiche Ergebnis aus, benötigt dafür aber nur rund 1/60 der Zeit. Dieser Unterschied im Rechenaufwand wird noch wesentlich deutlicher, wenn man in den Quelldateien den Wert `N` statt auf 10 000 auf 100 000 setzt: In diesem Fall ist der erste Algorithmus auf meinem Rechner erst nach 14.397 s (!!) fertig, während der zweite nur 0,032 s benötigt.

`valgrind` - Speicher-Testprogramm

Das Programm `valgrind` prüft bei einem ausführbaren Programm, wieviel Speicher dynamisch reserviert bzw. wieder freigegeben wurde.

```
valgrind programmname
```

Man kann `valgrind` auch auf Standard-Programme anwenden, beispielsweise wird mittels `valgrind ps -ax` der Speicherbedarf des Programms `ps` analysiert, wenn dieses mit der Option `-ax` aufgerufen wird.

Die C-Standardbibliothek

Im folgenden Abschnitt sind diejenigen C-Bibliotheken beschrieben, die jederzeit vom `gcc`-Compiler gefunden werden und somit in C-Programme mittels `#include` eingebunden werden können, ohne dass weitere Pfadanpassungen notwendig sind.

`assert.h` – Einfache Tests

- `void assert(logical_expression);`

  Diese Funktion kann – wie eine `if`-Bedingung – an beliebigen Stellen im Code eingesetzt werden. Ergibt der angegebene logische Ausdruck allerdings keinen wahren (von Null verschiedenen) Wert, so bricht `assert()` das Programm ab und gibt auf dem `stderr`-Kanal als Fehlermeldung aus, welche Zeile beziehungsweise notwendige Bedingung den Absturz verursacht hat.

`math.h` – Mathematische Funktionen

- `double sin(double x)`

  Gibt den Sinus-Wert eines in Radiant angegebenen x-Werts an.

- `double cos(double x)`

  Gibt den Cosinus-Wert eines in Radiant angegebenen x-Werts an.

- `double tan(double x)`

  Gibt den Tangens-Wert eines in Radiant angegebenen x-Werts an.

- `double asin(double x)`

  Gibt den Arcus-Sinus-Wert eines x-Werts an, wobei $x \in [-1; +1]$ gelten muss.

- `double acos(double x)`

  Gibt den Arcus-Cosinus-Wert eines x-Werts an, wobei $x \in [-1; +1]$ gelten muss.

- `double atan(double x)`

  Gibt den Arcus-Tangens-Wert eines x-Werts an.

- `double sinh(double x)`

  Gibt den Sinus-Hyperbolicus-Wert eines x-Werts an.

- `double cosh(double x)`

  Gibt den Cosinus-Hyperbolicus-Wert eines x-Werts an.

- `double tanh(double x)`

  Gibt den Tangens-Hyperbolicus-Wert eines x-Werts an.

- `double exp(double x)`

  Gibt den Wert der Exponentialfunktion e^x eines x-Werts an.

- `double log(double x)`

  Gibt den Wert der natürlichen Logarithmusfunktion $\ln(x)$ an, wobei $x > 0$ gelten muss.

- `double log10(double x)`

  Gibt den Wert des Logarithmus zur Basis 10 an, wobei $x > 0$ gelten muss.

- `double pow(double x)`

  Gibt den Wert von x^y an. Ein Argumentfehler liegt vor, wenn $x = 0$ und $y < 0$ gilt, oder wenn $x < 0$ und y nicht ganzzahlig ist.

- `double sqrt(double x)`

  Gibt den Wert der Quadratwurzel eines x-Werts an, wobei $x \leq 0$.

- `double ceil(double x)`

  Gibt den kleinsten ganzzahligen Wert als `double` an, der nicht kleiner als x ist.

- `double floor(double x)`

  Gibt den größten ganzzahligen Wert als `double` an, der nicht größer als x ist.

- `double fabs(double x)`

  Gibt den Absolutwert $|x|$ eines x-Werts an.

- `double ldexp(double x, n)`

  Gibt den Wert des Ausdrucks $x \cdot 2^n$ an.

- `double frexp(double x, int *exp)`

  Zerlegt x in eine normalisierte Mantisse im Bereich $[\frac{1}{2}; 1]$, die als Ergebnis zurückgegeben wird, und eine Potenz von 2, die in `*exp` abgelegt wird. Ist x gleich Null, sind beide Teile des Resultats Null.

- `double modf(double x, double *ip)`

Zerlegt x in einen ganzzahligen Teil und einen Rest, die beide das gleiche Vorzeichen wie x besitzen. Der ganzzahlige Teil wird bei `*ip` abgelegt, der Rest wird als Ergebnis zurückgegeben.

- `double fmod(double x, double y)`

  Gibt den Gleitpunktrest von $\frac{x}{y}$ an, mit dem gleichen Vorzeichen wie x. Wenn y gleich Null ist, hängt das Resultat von der Implementierung ab.

`cmath.h` – Mathe-Funktionen für komplexe Zahlen

- `double creal(double complex z)`

  Gibt den Realteil der komplexen Zahl z als Ergebnis zurück.

- `double cimag(double complex z)`

  Gibt den Imaginärteil der komplexen Zahl z als Ergebnis zurück.

- `double cabs(double complex z)`

  Behandelt die komplexe Zahl z wie einen zweidimensionalen Vektor in der Zahlenebene; gibt den Betrag (die Länge) dieses Vektors als Ergebnis zurück.

- `double casin(double complex z)`

  Gibt den Arcus-Sinus-Wert der komplexen Zahl z an.

- `double cacos(double complex z)`

  Gibt den Arcus-Cosinus-Wert der komplexen Zahl z an, wobei der Realteil von z im Bereich $[-1; +1]$ liegen muss.

- `double catan(double complex x)`

  Gibt den Arcus-Tangens-Wert einer komplexen Zahl z an.

`string.h` – Zeichenkettenfunktionen

In der Definitionsdatei `<string.h>` gibt es zwei Gruppen von Funktionen für Felder und Zeichenketten. Die Namen der ersten Gruppe von Funktionen beginnen mit `mem`; diese sind allgemein zur Manipulation von Feldern vorgesehen. Die Namen der zweiten Gruppe von Funktionen beginnen mit `str` und ist speziell für Zeichenketten gedacht, die mit dem Zeichen `\0'` abgeschlossen sind.

Wichtig: Bei der Verwendung der `mem`- und `str`-Funktionen muss der Programmierer darauf achten, dass sich die Speicherplätze der zu kopierenden oder zu vergleichenden Zeicherketten nicht überlappen, da das Verhalten der Funktionen sonst nicht definiert ist.

mem-Funktionen

Die `mem`-Funktionen sind zur Manipulation von Speicherbereichen gedacht. Sie behandeln den Wert `\0` wie jeden anderen Wert, daher muss immer eine Bereichslänge angegeben werden.

- `void * memcpy(void *str_1, const void *str_2, size_t n)`

  Kopiert die ersten n Zeichen aus dem Array `str_2` in das Array `str_1`; gibt `str_1` als Ergebnis zurück.

- `void * memmove(void *str_1, const void *str_2, size_t n)`

  Kopiert ebenso wie *memcpy()* die ersten n Zeichen des Arrays `str_2` in das Array `str_1`; gibt `str_1` als Ergebnis zurück. `memmove()` funktioniert allerdings auch, wenn sich die Speicherplätze beider Arrays überlappen.

- `int memcmp(const void *str_1, const void *str_2, size_t n)`

  Vergleicht die ersten n Zeichen des Arrays `str_1` mit dem Array `str_2`; gibt als Ergebnis einen Wert < 0 zurück falls `str_1 < str_2` ist, den Wert 0 für `str_1 == str_2`, oder einen Wert > 0 falls `str_1 > str_2` ist.

  Die Bereiche werden nach den ASCII-Codes der Anfangsbuchstaben verglichen, nicht lexikalisch.

- `void * memchr(const void *str, char c, size_t n)`

  Gibt einen Zeiger auf das erste Byte mit dem Wert `c` im Array `str` zurück, oder `NULL`, wenn das Byte innerhalb der ersten n Zeichen nicht vorkommt.

- `void * memset(void *str, char c, size_t n)`

  Setzt die ersten n Bytes des Arrays `str` auf den Wert `c`; gibt `str` als Ergebnis zurück.

str-Funktionen

- `char * strcpy(char *str_1, const char *str_2)`

  Kopiert eine Zeichenkette `str_2` in ein Array `str_1`, inklusive `\0`; gibt `str_1` als Ergebnis zurück.

- `char * strncpy(char *str_1, const char *str_2, size_t n)`

  Kopiert höchstens n Zeichen aus der Zeichenkette `str_2` in die Zeichenkette `str_1`, und gibt `str_1` als Ergebnis zurück. Dabei wird `str_1` mit `\0` abgeschlossen, wenn `str_2` weniger als n Zeichen hat.

- `char * strcat(char *str_1, const char *str_2)`

  Hängt die Zeichenkette `str_2` hinten an die Zeichenkette `str_1` an; gibt `str_1` als Ergebnis zurück.

- `char * strncat(char *str_1, const char *str_2, size_t n)`

Fügt höchstens n Zeichen der Zeichenkette `str_2` hinten an die Zeichenkette `str_1` an und schließt `str_1` mit `\0` ab. Gibt `str_1` als Ergebnis zurück.

- `int strcmp(const char *str_1, const char *str_2)`

  Vergleicht die beiden Zeichenketten `str_1` und `str_2` miteinander; gibt als Ergebnis einen Wert < 0 zurück falls `str_1 < str_2` ist, den Wert 0 für `str_1 == str_2`, oder einen Wert > 0 falls `str_1 > str_2` ist.

  Die Zeichenketten werden nach den ASCII-Codes der Anfangsbuchstaben verglichen, nicht lexikalisch.

- `int strncmp(const char *str_1, const char *str_2, size_t n)`

  Vergleicht höchstens n Zeichen der Zeichenkette `str_1` mit der Zeichenkette `str_2`; gibt einen Wert < 0 zurück falls `str_1 < str_2` ist, den Wert 0 für `str_1 == str_2`, oder einen Wert > 0 falls `str_1 > str_2` ist.

  Die Zeichenketten werden nach den ASCII-Codes der Anfangsbuchstaben verglichen, nicht lexikalisch.

- `char * strchr(const char *str, char c)`

  Gibt einen Zeiger auf das erste Zeichen `c` in der Zeichenkette `str` als Ergebnis zurück, oder `NULL`, falls `c` nicht in der Zeichenkette enthalten ist.

- `char * strrchr(const char *str, char c)`

  Gibt einen Zeiger auf das letzte Zeichen `c` in der Zeichenkette `str` als Ergebnis zurück, oder `NULL`, falls `c` nicht in der Zeichenkette enthalten ist.

- `size_t strspn(const char *str_1, const char *str_2)`

  Gibt die Anzahl der Zeichen am Anfang der Zeichenkette `str_1` als Ergebnis zurück, die in dieser Reihenfolge ebenfalls in der Zeichenkette `str_2` vorkommen.

- `size_t strcspn(const char *str_1, const char *str_2)`

  Gibt die Anzahl der Zeichen am Anfang der Zeichenkette `str_1` als Ergebnis zurück, die in dieser Reihenfolge *nicht* in der Zeichenkette `str_2` vorkommen.

- `char * strpbrk(const char *str_1, const char *str_2)`

  Gibt einen Zeiger auf die Position in der Zeichenkette `str_1` als Ergebnis zurück, an der irgendein Zeichen aus der Zeichenkette `str_2` erstmals vorkommt, oder `NULL`, falls keines dieser Zeichen vorkommt.

- `char * strstr(const char *str_1, const char *str_2)`

Gibt einen Zeiger auf erstes Vorkommen von der Zeichenkette `str_2` innerhalb der Zeichenkette `str_1` als Ergebnis zurück, oder `NULL`, falls diese nicht vorkommt.

- `size_t strlen(const char *str)`

  Gibt die Länge der Zeichenkette `str` ohne `\0` an.

- `char * strerror(size_t n)`

  Gibt einen Zeiger auf diejenige Zeichenkette als Ergebnis zurück, die dem Fehler mit der Nummer n zugewiesen ist.

- `char * strtok(char *str_1, const char *str_2)`

  Durchsucht die Zeichenkette `str_1` nach Zeichenfolgen, die durch Zeichen aus der Zeichenkette `str_2` begrenzt sind.

`stdio.h` – Ein- und Ausgabe

Die Datei `stdio.h` definiert Typen und Funktionen zum Umgang mit Datenströmen („Streams“). Ein Stream ist Quelle oder Ziel von Daten und wird mit einer Datei oder einem angeschlossenen Gerät verknüpft.

Unter Windows muss zwischen Streams für binäre und für Textdateien unterschieden werden, unter Linux nicht. Ein Textstream ist eine Folge von Zeilen, die jeweils kein oder mehrere Zeichen enthalten und jeweils mit `'\n'` abgeschlossen sind.

Ein Stream wird mittels der Funktion `open()` mit einer Datei oder einem Gerät verbunden; die Verbindung wird mittels der Funktion `close()` wieder aufgehoben. Öffnet man eine Datei, so erhält man einen Zeiger auf ein Objekt vom Typ `FILE`, in welchem alle Information hinterlegt sind, die zur Kontrolle des Stream nötig sind.

Wenn die Ausführung eines Programms beginnt, sind die drei Standard-Streams `stdin`, `stdout` und `stderr` bereits automatisch geöffnet.

Dateioperationen

Die folgenden Funktionen beschäftigen sich mit Datei-Operationen. Der Typ `size_t` ist der vorzeichenlose, ganzzahlige Resultattyp des *sizeof*-Operators.

- `FILE *fopen(const char *filename, const char *mode)`

  Öffnet die angegebene Datei; gibt als Ergebnis einen Datenstrom zurück, oder `NULL` falls das Öffnen fehlschlägt.

  Als Zugriffsmodus `mode` kann angegeben werden:

  - `"r"`: Textdatei zum Lesen öffnen
  - `"w"`: Textdatei zum Schreiben neu erzeugen (gegebenenfalls alten Inhalt wegwerfen)

  - `"a"`: Text anfügen; Datei zum Schreiben am Dateiende öffnen oder erzeugen
  - `"r+"`: Textdatei zum Ändern öffnen (Lesen und Schreiben)
  - `"w+"`: Textdatei zum Ändern erzeugen (gegebenenfalls alten Inhalt wegwerfen)
  - `"a+"`: Datei neu erzeugen oder zum Ändern öffnen und Text anfügen (Schreiben am Ende)

- `FILE *freopen(const char *filename, const char *mode, FILE *stream)`

  Öffnet die angegebene Datei für den angegebenen *Zugriffsmodus* und verknüpft den Datenstrom `stream` damit. Als Ergebnis wird `stream` zurück gegeben, oder `Null` falls ein Fehler auftritt.

  Mit `freopen()` ändert man normalerweise die Dateien, die mit `stdin`, `stdout` oder `stderr` verknüpft sind.

- `int fflush(FILE *stream)`

  Sorgt bei einem Ausgabestrom dafür, dass gepufferte, aber noch nicht geschriebene Daten geschrieben werden; bei einem Eingabestrom ist der Effekt undefiniert. Die Funktion gibt normalerweise `NULL` als Ergebnis zurück, oder `EOF` (Konstante mit Wert `-1`), falls ein Schreibfehler auftritt.

  `fflush(NULL)` bezieht sich auf alle offenen Dateien.

- `int feof(FILE *stream);`

  Prüft, ob der angegebene File-Pointer auf das Ende einer Datei zeigt. Die Funktion gibt normalerweise `0` als Ergebnis zurück, oder einen Wert ungleich Null, wenn das Ende der Datei erreicht ist.

- `int ferror(FILE *stream);`

  Jede `FILE`-Struktur besitzt eine Steuervariable („Flag“) namens `error`. `ferror()` prüft, ob dieses Flag gesetzt ist, was beispielsweise durch einen Fehler beim Lesen oder Schreiben verursacht wird. Die Funktion gibt normalerweise `0` als Ergebnis zurück, oder einen Wert ungleich Null, wenn das Fehler-Flag des File-Pointers gesetzt ist.

- `int fclose(FILE *stream)`

  Schreibt noch nicht geschriebene Daten für `stream`, wirft noch nicht gelesene, gepufferte Eingaben weg, gibt automatisch angelegte Puffer frei und schließt den Datenstrom. Die Funktion gibt normalerweise `NULL` als Ergebnis zurück, oder `EOF` (Konstante mit Wert `-1`), falls ein Fehler auftritt.

- `int remove(const char *filename)`

  Löscht die angegebene Datei, so dass ein anschließender Versuch, sie zu öffnen, fehlschlagen wird. Bei einem Fehler gibt die Funktion einen von Null verschiedenen Wert zurück.

- `int rename(const char *oldname, const char *newname)`

Ändert den Namen einer Datei. Bei einem Fehler gibt die Funktion einen von Null verschiedenen Wert zurück.

- `FILE * tmpfile(void)`

  Erzeugt eine temporäre Datei mit Zugriffsmodus `"wb+"`, die automatisch gelöscht wird, wenn der Zugriff abgeschlossen wird, oder wenn das Programm normal zu Ende geht. Als Ergebnis gibt `tmpfile()` einen Datenstrom zurück, oder `NULL` falls die Datei nicht erzeugt werden konnte.

- `char * tmpnam(char s[L_tmpnam])`

  `tmpnam(NULL)` erzeugt eine Zeichenkette, die nicht der Name einer existierenden Datei ist, und gibt einen Zeiger auf einen internen Vektor im statischen Speicherbereich als Ergebnis zurück.

  `tmpnam(s)` speichert die Zeichenkette in `s` und gibt `s` als Ergebnis zurück; in `s` müssen wenigstens `L_tmpnam` Zeichen abgelegt werden können.

  Bei jedem Aufruf erzeugt die Funktion einen anderen Namen; man kann höchstens von `TMP_MAX` verschiedenen Namen während der Ausführung des Programms ausgehen. Zu beachten ist, dass ein Name und keine Datei erzeugt wird.

- `int setvbuf(FILE *stream, char *buf, int mode, size_t size)`

  Kontrolliert die Pufferung bei einem Datenstrom; die Funktion muss aufgerufen werden, bevor gelesen oder geschrieben wird, und vor allen anderen Operationen. Hat `mode` den Wert `_IOFBF`, so wird vollständig gepuffert, `_IOLBF` sorgt für zeilenweise Pufferung bei Textdateien und `_IONBF` verhindert Puffern. Wenn `buf` nicht gleich `NULLi` ist, wird `buf` als Puffer verwendet; andernfalls wird ein Puffer angelegt. `size` legt die Puffergröße fest.

  Bei einem Fehler gibt die Funktion einen von Null verschiedenen Wert zurück.

- `void setbuf(FILE *stream, char *buf)`

  Wenn `buf` den Wert `NULL` hat, wird der Datenstrom nicht gepuffert; andernfalls ist `setbuf` äquivalent zu `(void) setvbuf(stream, buf, _IOFBF, BUFSIZ)`.

Ändern bedeutet, dass die gleiche Datei gelesen und geschrieben werden darf; `fflush()` oder eine Funktion zum Positionieren in Dateien muss zwischen einer Lese- und einer Schreiboperation oder umgekehrt aufgerufen werden. Dateinamen sind auf `FILENAME_MAX Zeichen` begrenzt, höchstens `FOPEN_MAX` Dateien können gleichzeitig offen sein.

Aus- und Eingabe

- `int fputs(const char *str, FILE *stream)`

  Schreibt die Zeichenkette `str` in die mit dem File-Pointer angegebene Datei. Als Ergebnis gibt die Funktion einen nicht-negativen Wert als Ergebnis

zurück, oder `EOF` (Konstante mit Wert `-1`), wenn ein Fehler aufgetreten ist.

- `int fprintf(FILE *stream, const char *format, ...)`

  Die Funktion `fprintf()` wandelt Ausgaben um und schreibt sie in `stream` unter Kontrolle von `format`. Als Ergebnis gibt sie die Anzahl der geschriebenen Zeichen zurück; der Wert ist negativ, wenn ein Fehler aufgetreten ist.

- `int printf(const char *format, ...)`

  `printf(...)` ist äquivalent zu `fprintf(stdout, ...)`. Die formatierte Ausgabe der `printf()`-Funktion ist im Abschnitt *Ausgabe und Eingabe* näher beschrieben.

- `int sprintf(char *s, const char *format, ...)`

  Die Funktion `sprintf()` funktioniert wie `printf()`, nur wird die Ausgabe in das Zeichenarray `s` geschrieben und mit `\0` abgeschlossen. `s` muss groß genug für das Resultat sein. Im Ergebniswert wird `\0` nicht mitgezählt.

`stdlib.h` – Hilfsfunktionen

Die Definitionsdatei `<stdlib.h>` vereinbart Funktionen zur Umwandlung von Zahlen, für Speicherverwaltung und ähnliche Aufgaben.

- `double atof(const char *s)`

  Wandelt die Zeichenkette `s` in `double` um. Beendet die Umwandlung beim ersten unbrauchbaren Zeichen.

- `int atoi(const char *s)`

  Wandelt die Zeichenkette `s` in `int` um. Beendet die Umwandlung beim ersten unbrauchbaren Zeichen.

- `long atol(const char *s)`

  Wandelt die Zeichenkette `s` in `long` um. Beendet die Umwandlung beim ersten unbrauchbaren Zeichen.

- `double strtod(const char *s, char **endp)`

  Wandelt den Anfang der Zeichenkette `s` in `double` um, dabei wird Zwischenraum am Anfang ignoriert. Die Umwandlung wird beim ersten unbrauchbaren Zeichen beendet. Die Funktion speichert einen Zeiger auf den eventuell nicht umgewandelten Rest der Zeichenkette bei `*endp`, falls `endp` nicht `NULL` ist. Falls das Ergebnis zu groß ist, (also bei einem Overflow), wird als Resultat `HUGE_VAL` mit dem korrekten Vorzeichen geliefert; liegt das Ergebnis zu dicht bei Null (also bei einem Underflow), wird Null geliefert. In beiden Fällen erhält `errno` den Wert `ERANGE`.

- `long strtol(const char *s, char **endp, int base)`

Wandelt den Anfang der Zeichenkette `s` in `long` um, dabei wird Zwischenraum am Anfang ignoriert. Die Umwandlung wird beim ersten unbrauchbaren Zeichen beendet. Die Funktion speichert einen Zeiger auf den eventuell nicht umgewandelten Rest der Zeichenkette bei `*endp`, falls `endp` nicht `NULL` ist. Hat `base` einen Wert zwischen 2 und 36, erfolgt die Umwandlung unter der Annahme, dass die Eingabe in dieser Basis repräsentiert ist.

Hat `base` den Wert Null, wird als Basis 8, 10 oder 16 verwendet, je nach `s`; eine führende Null bedeutet dabei oktal und `0x` oder `0X` zeigen eine hexadezimale Zahl an. In jedem Fall stehen Buchstaben für die Ziffern von 10 bis `base-1`; bei Basis `16` darf `0x` oder `0X` am Anfang stehen. Wenn das Resultat zu groß werden würde, wird je nach Vorzeichen `LONG_MAX` oder `LONG_MIN` geliefert und `errno` erhält den Wert `ERANGE`.

- `unsigned long strtoul(const char *s, char **endp, int base)`

  Funktioniert wie `strtol()`, nur ist der Resultattyp `unsigned long` und der Fehlerwert ist `ULONG_MAX`.

- `int rand(void)`

  Gibt als Ergebnis eine ganzzahlige Pseudo-Zufallszahl im Bereich von 0 bis `RAND_MAX` zurück; `RAND_MAX` ist mindestens `32767`.

- `void srand(unsigned int seed)`

  Benutzt `seed` als Ausgangswert für eine neue Folge von Pseudo-Zufallszahlen. Der erste Ausgangswert ist 1.

- `void * calloc(size_t nobj, size_t size)`

  Gibt als Ergebnis einen Zeiger auf einen Speicherbereich für einen Vektor von `nobj` Objekten zurück, jedes mit der Größe `size`, oder `NULL`, wenn die Anforderung nicht erfüllt werden kann. Der Bereich wird mit Null-Bytes initialisiert.

- `void * malloc(size_t size)`

  Gibt einen Zeiger auf einen Speicherbereich für ein Objekt der Größe `size` zurück, oder `NULL`, wenn die Anforderung nicht erfüllt werden kann. Der Bereich ist nicht initialisiert.

- `void * realloc(void *p, size_t size)`

  Ändert die Größe des Objekts, auf das der Pointer `p` zeigt, in `size` ab. Bis zur kleineren der alten und neuen Größe bleibt der Inhalt unverändert. Wird der Bereich für das Objekt größer, so ist der zusätzliche Bereich nicht initialisiert. `realloc()` liefert einen Zeiger auf den neuen Bereich oder `NULL`, wenn die Anforderung nicht erfüllt werden kann; in diesem Fall wird der Inhalt nicht verändert.

- `void free(void *p)`

Gibt den Bereich frei, auf den der Pointer `p` zeigt; die Funktion hat keinen Effekt, wenn `p` den Wert `NULL` hat. `p` muss auf einen Bereich zeigen, der zuvor mit `calloc()`, `malloc()` oder `realloc()` angelegt wurde.

- `void abort(void)`

  Sorgt für eine anormale, sofortige Beendigung des Programms.

- `void exit(int status)`

  Beendet das Programm normal: Dabei werden *atexit()*-Funktionen in der umgekehrten Reihenfolge ihrer Hinterlegung aufgerufen, Puffer offener Dateien werden geschrieben, offene Ströme abgeschlossen, und die Kontrolle geht an die Umgebung des Programms zurück. Welcher `status` an die Umgebung des Programms geliefert wird, hängt von der Implementierung ab, aber `Null` gilt als erfolgreiches Ende. Die Werte `EXIT_SUCCESS` (Wert: `0`) und `EXIT_FAILURE` (Wert: `1`) können ebenfalls angegeben werden.

- `int atexit(void (*fcn)(void))`

  Hinterlegt die Funktion `fcn`, damit sie aufgerufen wird, wenn das Programm normal endet, und liefert einen von Null verschiedenen Wert, wenn die Funktion nicht hinterlegt werden kann.

- `int system(const char *s)`

  Gibt die Zeichenkette `s` an die Umgebung zur Ausführung. Hat `s` den Wert `NULL`, so liefert `system()` einen von Null verschiedenen Wert, wenn es einen Kommandoprozessor gibt. Wenn `s` von `NULL` verschieden ist, dann ist der Resultatwert von der Implementierung abhängig.

- `char * getenv(const char *name)`

  Gibt die zu `name` gehörende Zeichenkette aus der Umgebung als Ergebnis zurück, oder `NULL`, wenn keine Zeichenkette existiert. Die Details hängen von der Implementierung ab.

- `void * bsearch(const void *key, const void *base, size_t n, size_t size, int (*cmp)(const void *keyval, const void *datum))`

  Durchsucht `base[0]` bis `base[n-1]` nach einem Eintrag, der gleich `*key` ist. Die Funktion `cmp` muss einen negativen Wert liefern, wenn ihr erstes Argument (der Suchschlüssel) kleiner als ihr zweites Argument (ein Tabelleneintrag) ist, Null, wenn beide gleich sind, und sonst einen positiven Wert.

  Die Elemente des Arrays base müssen aufsteigend sortiert sein. In `size` muss die Größe eines einzelnen Elements übergeben werden. `bsearch()` gibt als Ergebnis einen Zeiger auf das gefundene Element zurück, oder `NULL`, wenn keines existiert.

- `void qsort(void *base, size_t n, size_t size, int (*cmp)(const void *, const void *))`

Sortiert ein Array `base[0]` bis `base[n-1]` von Objekten der Größe `size` in aufsteigender Reihenfolge. Für die Vergleichsfunktion `cmp` gilt das gleiche wie bei `bsearch()`.

- `int abs(int x)`

  Gibt den den absoluten Wert (Betrag) $|x|$ von x als `int` an.

- `long labs(long x)`

  Gibt den absoluten Wert (Betrag) $|x|$ von x als `long` an.

- `div_t div(int n, int z)`

  Gibt den Quotienten und Rest von $\frac{n}{z}$ an. Die Ergebnisse werden in den `int`-Komponenten `quot` und `rem` einer Struktur vom Typ `div_t` abgelegt.

- `ldiv_t ldiv(long n, long z)`

  Gibt den Quotienten und Rest von $\frac{n}{z}$ an. Die Ergebnisse werden in den `long`-Komponenten `quot` und `rem` einer Struktur vom Typ `ldiv_t` abgelegt.

`time.h` – Funktionen für Datum und Uhrzeit

Die Definitionsdatei `time.h` vereinbart Typen und Funktionen zum Umgang mit Datum und Uhrzeit. Manche Funktionen verarbeiten die Ortszeit, die von der Kalenderzeit zum Beispiel wegen einer Zeitzone abweicht. `clock_t` und `time_t` sind arithmetische Typen, die Zeiten repräsentieren, und `struct tm` enthält die Komponenten einer Kalenderzeit:

```
struct tm
{
    // Sekunden nach der vollen Minute (0, 61)
    // (Die zusätzlich möglichen Sekunden sind Schaltsekunden)
    int tm_sec;

    // Minuten nach der vollen Stunde (0, 59)
    int tm_min;

    // Stunden seit Mitternacht (0, 23)
    int tm_hour;

    // Tage im Monat (1, 31)
    int tm_mday;

    // Monate seit Januar (0, 11)
    int tm_mon;

    // Jahre seit 1900
    int tm_year;
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
    // Tage seit Sonntag (0, 6)
    int tm_wday;

    // Tage seit dem 1. Januar (0, 365)
    int tm_yday;

    // Kennzeichen für Sommerzeit
    int tm_isdst;
}
```

`tm_isdst` ist positiv, wenn Sommerzeit gilt, Null, wenn Sommerzeit nicht gilt, und negativ, wenn die Information nicht zur Verfügung steht.

- `clock_t clock(void)`

  Gibt die Rechnerkern-Zeit an, die das Programm seit Beginn seiner Ausführung verbraucht hat, oder `-1`, wenn diese Information nicht zur Verfügung steht.

  `clock()/CLOCKS_PER_SEC` ist eine Zeit in Sekunden.

- `time_t time(time_t *tp)`

  Gibt die aktuelle Kalenderzeit an, oder `-1`, wenn diese nicht zur Verfügung steht. Wenn `tp` von `NULL` verschieden ist, wird der Resultatwert auch bei `*tp` abgelegt.

- `double difftime(time_t time2, time_t timel)`

  Gibt die Differenz der Zeitangaben `time2 - timel` in Sekunden an.

- `time_t mktime(struct tm *tp)`

  Wandelt die Ortszeit in der Struktur `*tp` in Kalenderzeit um, die so dargestellt wird wie bei `time()`. Die Komponenten erhalten Werte in den angegebenen Bereichen. `mktime()` gibt die Kalenderzeit als Ergebnis zurück, oder den Wert `-1`, wenn diese nicht dargestellt werden kann.

- `size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)`

  Formatiert Datum und Zeit aus `*tp` in der Zeichenkette `s` gemäß `fmt`, analog zu einem `printf`-Format. Gewöhnliche Zeichen (insbesondere auch das abschließende `\0`) werden nach `s` kopiert. Jedes `%...` wird gemäß der unten folgenden Liste ersetzt, wobei Werte verwendet werden, die der lokalen Umgebung entsprechen.

  Es werden höchstens `smax` Zeichen in der Zeichenkette `s` abgelegt. Als Ergebnis gibt `strftime()` die Anzahl der resultierenden Zeichen zurück, mit Ausnahme von `\0`. Wenn mehr als `smax` Zeichen erzeugt wurden, gibt `strftime` den Wert Null als Ergebnis zurück.

  Umwandlungszeichen für den Formatstring `fmt`:

`%a`	abgekürzter Name des Wochentags.	`%A`	voller Name des Wochentags.
`%b`	abgekürzter Name des Monats.	`%B`	voller Name des Monats.
`%c`	lokale Darstellung von Datum und Zeit.	`%d`	Tag im Monat (01 - 31).
`%H`	Stunde (00 - 23).	`%I`	Stunde (01 - 12).
`%j`	Tag im Jahr (001 - 366).	`%m`	Monat (01 - 12).
`%M`	Minute (00 - 59).	`%p`	lokales Äquivalent von AM oder PM.
`%S`	Sekunde (00 - 61).	`%U`	Woche im Jahr (Sonntag ist erster Tag) (00 - 53).
`%w`	Wochentag (0 - 6, Sonntag ist 0).	`%W`	Woche im Jahr (Montag ist erster Tag) (00 - 53).
`%x`	lokale Darstellung des Datums.	`%X`	lokale Darstellung der Zeit.
`%y`	Jahr ohne Jahrhundert (00 - 99).	`%Y`	Jahr mit Jahrhundert.
`%Z`	Name der Zeitzone, falls diese existiert.	`%%`	%. (Gibt ein % aus)

Die folgenden vier Funktionen liefern Zeiger auf statische Objekte, die von anderen Aufrufen überschrieben werden können.

- `char * asctime(const struct tm *tp)`

  Konstruiert aus der Zeit in der Struktur `*tp` eine Zeichenkette folgender Form: `Sun Jan 3 15:14:13 1988\n\0`

- `char * ctime(const time_t *tp)`

  Verwandelt die Kalenderzeit `*tp` in Ortszeit; dies ist äquivalent zu `asctime(localtime(tp))`

- `struct tm * gmtime(const time_t *tp)`

  Verwandelt die Kalenderzeit `*tp` in Coordinated Universal Time (UTC). Die Funktion liefert `NULL`, wenn UTC nicht zur Verfügung steht. Der Name `gmtime` hat historische Bedeutung.

- `struct tm * localtime(const time_t *tp)`

  Verwandelt die Kalenderzeit `*tp` in Ortszeit.

Curses

Die C-Bibliothek Curses beziehungsweise ihre neuere Version NCurses bietet die Möglichkeit, textbasierte Benutzeroberflächen zu erzeugen. Curses wird daher in vielen Shell-Programmen verwendet, darunter aptitude, cmus, mc, usw.

Curses starten und beenden

Um Curses zu starten, muss zunächst die Funktion `initscr()` aufgerufen werden. Diese Funktion erzeugt einen leeres Fenster und weist ihm den Namen `stdscr` („standard screen“) zu. Damit das neue Fenster angezeigt wird, muss anschließend die Funktion `refresh()` aufgerufen werden, so dass das Shell-Fenster aktualisiert wird und die Änderungen sichtbar werden.

Mit der `refresh()`-Anweisung werden in Curses zwei Teilfunktionen aufgerufen: Zunächst werden mittels der ersten Funktion `wnoutrefresh()` nur die veränderten Teile eines Curses-Fensters in einem „virtuellen“ Fenster aktualisiert. Anschließend wird dieses mittels der zweiten Funktion `douptate()` auf den Bildschirm übertragen. Somit wird immer nur der Teil des Fensters aktualisiert, der tatsächlich verändert wurde; dies ist wesentlich effizienter, als wenn ständig das gesamte Shell-Fenster aktualisiert werden müsste.

Um ein Curses-Programm wieder zu beenden, verwendet man die Funktion `endwin()`. Diese löscht den Bildschirm und stellt automatisch die vorgefundenen Shell-Einstellungen wieder her. Da `endwin()` insgesamt zahlreiche Aufräumarbeiten übernimmt, sollte Curses stets mit dieser Funktion beendet werden.

Ein minimales Curses-Programm, das nur kurz einen leeren Bildschirm erzeugt, auf diesem „Hallo Welt“ ausgibt und sich nach kurzer Zeit selbst beendet, kann folgendermaßen aussehen:

```
// Datei: curses-beispiel-1.c

#include <ncurses.h>

int main(void)
{
    initscr();
    printw("Hallo Welt!");
    refresh();
    napms(3000);
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
    endwin();
    return 0;
}
```

In diesem Beispiel wurde zudem die Curses-Funktion `napms()` verwendet, die eine weitere Ausführung des Programms um die angegebene Anzahl in Millisekunden verzögert.

Ausgeben und Einlesen von Text

Zur Ausgabe von Text gibt es in Curses im Wesentlichen drei Funktionen:

- Mittels `addch(c)` kann ein einzelnes Zeichen auf dem Bildschirm ausgegeben werden.
- Mittels `addstr(*str)` kann eine ganze Zeichenkette auf dem Bildschirm ausgegeben werden. (Dabei wird intern die Funktion `addch()` aufgerufen, bis die Zeichenkette abgearbeitet ist.)
- Mittels `printw()` kann Text in der gleichen Weise in einem Curses-Fenster ausgegeben werden, wie dies mittels der Funktion *printf()* auf dem Standard-Ausgang der Fall ist.

Damit der Text an der richtigen Stelle im Curses-Fenster erscheint, kann man mittels der Funktion `move()` den Cursor an eine bestimmte Stelle bewegen. Als erstes Argument wird dabei die Zeilennummer `y`, als zweites die Spaltennummer `x` angegeben, also `move(y, x)`.[1] Da Curses, wie in C üblich, bei der Nummerierung mit Null beginnt, entspricht `move(0,0)` einem Bewegen des Cursors in die obere linke Ecke; die erlauben Maximalwerte für die Zeilen- und Spaltennummer in `move()` sind entsprechend um 1 kleiner als die Zeilen- und Spaltenanzahl des Fensters. Diese beiden Werte können mittels der Funktion `getmaxyx(stdscr, maxrow, maxcol)` bestimmt werden, wobei `maxrow` und `maxcol` im Voraus als `int` deklariert werden müssen:[2]

```
// Datei: curses-beispiel-2.c

#include <ncurses.h>

int maxrow, maxcol;


int main(void)
    {
    initscr();

    // Größe des Curses-Fensters bestimmen:
    getmaxyx(stdscr, maxrow, maxcol);
```

(continues on next page)

[1] Eine „Spalte“ in Curses der Breite eines Textzeichens; die meisten Fenster haben daher mehr Spalten als Zeilen.

[2] Für die Größe des Hauptfensters `stdscr` sind in Curses auch die Makros `LINES` und `COLS` definiert, die vom Compiler durch die beim Programmstart vorliegenden Werte ersetzt werden.

(Fortsetzung der vorherigen Seite)

```
    // Größe des Curses-Fensters ausgeben:
    move(0,0);
    printw("Das Fenster hat %d Zeilen und %d Spalten.", maxrow, maxcol);
    refresh();

    napms(3000);
    endwin();
    return 0;
    }
```

Die Kombination von `move()` mit einer der Print-Anweisungen kommt in Curses-Anwendungen sehr häufig vor; daher gibt es zu den drei Ausgabefunktionen `addch()`, `addstr()` und `printw()` auch die kombinierten Funktionen `mvaddch()`, `mvaddstr()` und `mvprintw()`. Diesen wird beim Aufruf zunächst die gewünschte Position des Cursor angegeben, die übrigen Argumente sind mit den Basisfunktionen identisch. Beispielsweise sind die folgenden beiden Aufrufe identisch:

```
// Text in Zeile 0, Spalte 3 ausgeben:
move(0,3)
addstr("Hallo Curses!")

// Kurzschreibweise:
mvaddstr(0, 3, "Hallo Curses!")
```

Zur Eingabe von Text gibt es in Curses ebenfalls drei grundlegende Funktionen:

- Mittels `getch(c)` kann ein einzelnes Zeichen vom Bildschirm eingelesen werden; das Zeichen wird dabei automatisch eingelesen, ohne dass die `Enter`-Taste gedrückt werden muss.
- Mittels `getstr(*str)` und `getnstr(*str, n)` kann eine ganze Zeichenkette vom Curses-Fenster eingelesen werden, wie es mit *gets()* von der Standard-Eingabe der Fall ist. Die Funktion `getnstr()` beschränkt die Anzahl an eingelesenen Zeichen dabei auf `n` Stück, so dass sichergestellt werden kann, dass das Array, in dem die Zeichenkette abgelegt werden soll, ausreichend groß ist.
- Mittels `scanw()` kann Text in der gleichen Weise von einem Curses-Fenster eingelesen werden, wie dies mittels der Funktion *scanf()* aus dem Standard-Eingang der Fall ist.

Als Standard geben alle Eingabefunktionen die vom Benutzer eingegebenen Zeichen unmittelbar auf dem Bildschirm aus, auch ohne dass dazu die `refresh()`-Funktion aufgerufen werden müsste; zusätzlich stoppt das Programm, bis die Eingabe vom Benutzer erfolgt ist. Ist dies nicht gewünscht, so müssen diese Einstellung, wie im folgenden Abschnitt beschrieben, deaktiviert werden.

Modifizierung der Ein- und Ausgabe

In Curses gibt es folgende Funktionen, die das Verhalten des Programms hinsichtlich Eingabe und Ausgabe anzupassen:

- `raw()` und `cbreak()`:

  Normalerweise speichert die Shell die Eingabe des Benutzers in einem Puffer, bis ein Neues-Zeile-Zeichen oder ein Carriage-Return-Zeichen (Enter-Taste) erscheint. Die meisten interaktiven Programme benötigen die eingegebenen Zeichen allerdings unmittelbar. Die beiden Funktionen `raw()` und `cbreak()` deaktivieren beide das Puffern von eingegebenen Zeichen, wobei sie sich in einem Detail unterscheiden: Eingegebene Zeichen wie `Ctrl z` („Suspend“) oder `Ctrl c` („Interrupt“), die von der Shell normalerweise als Kontrollsequenzen interpretiert werden, werden auch bei der Verwendung von `cbreak()` zunächst von der Shell ausgewertet. Bei Verwendung von `raw()` werden auch diese Zeichen direkt ans Programm weitergeleitet und dort interpretiert.

- `echo()` und `noecho()`:

  Diese beiden Funktionen beeinflussen, ob vom Benutzer eingegebene Zeichen unmittelbar auf dem Bildschirm erscheinen sollen oder nicht. Diese Funktionen sind insbesondere in Verbindung mit der Curses-Funktion *getch()* von Bedeutung, um beispielsweise in interaktiven Programmen die unnötige Wiedergabe der vom Benutzer gedrückten Tasten auf dem Bildschirm zu vermeiden. Meist wird `noecho()` zu Beginn des Programms aufgerufen, und der Echo-Modus nur im Bedarfsfall (beispielsweise beim zeichenweise Einlesen von Text) aktiviert.

- `keypad()`:

  Diese Funktion sollte von jedem interaktiven Curses-Programm aufgerufen werden, denn sie ermöglicht die Verwendung der Funktions- und Pfeiltasten. Um beispielsweise die Funktion für den Standard-Bildschirm `stdscr` zu aktivieren, gibt man `keypad(stdscr, TRUE);` ein.[3]

- `curs_set()`:

  Diese Funktion kann verwendet werden, um den Cursor unsichtbar oder wieder sichtbar zu machen. Mit `curs_set(0);` wird der Cursor unsichtbar, mit `curs_set(1);` wieder sichtbar.

- `halfdelay(n)`:

  Mit dieser nur in Ausnahmefällen verwendeten Funktion kann festgelegt werden, dass beim dem Einlesen eines Zeichens mittels *getch()* oder einer Zeichenkette maximal n Zehntel Sekunden gewartet wird. Wird in dieser Zeit kein Text eingegeben, so fährt das Programm fort. Dies kann beispielsweise für eine Timeout-Funktion bei einer Passwort-Eingabe verwendet werden.

- `nodelay()`:

[3] Die Konstanten `TRUE` und `OK` beziehungsweise `FALSE` sind in der Datei `ncurses.h` als `1` beziehungsweise `0` definiert.

Diese Funktion wird von den meisten interaktiven Curses-Programm zu Beginn aufgerufen, denn sie verhindert, dass das Programm bei der Verwendung der Funktion *getch()* anhält. Anstelle dessen liefert `getch()` kontinuierlich den Wert `ERR` (entspricht dem Wert `-1`) zurück, sofern der Benutzer keine Taste gedrückt hat.

Mit Hilfe von `nodelay(stdscr, TRUE)` kann beispielsweise eine `mainloop()` programmiert werden, die einzelne von der Tastatur aus eingegebene Zeichen über eine *switch*-Anweisung mit bestimmten Anweisungen verknüpft:[4]

```
// Datei: curses-beispiel-3.c

#include <ncurses.h>

int main()
{
    int c;
    int quit = FALSE;

    initscr();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);
    nodelay(stdscr, TRUE);

    mvprintw(0,0, "Bitte Taste eingeben oder Programm mit \'q\' beenden.");

    while( !quit )
    {
        c = getch();
        switch(c)
        {
            case ERR:
                napms(10);
                break;
            case 'q':
                quit = TRUE;
                break;

            default:
                mvprintw(3, 0, "ASCII-Code des Zeichens: %3d;", c);
                mvprintw(3, 30, "Zeichen wird dargestellt als: \'%c\'.", c);
                break;
        }

        refresh();
    }

    endwin();
```

(continues on next page)

[4] Mit `nodelay(stdscr, FALSE)` kann das ursprüngliche Verhalten von `getch()` wieder hergestellt werden.

(Fortsetzung der vorherigen Seite)

```
    return 0;
}
```

Im obigen Beispielprogramm wird zunächst Curses gestartet und das Bildschirm-Verhalten angepasst. Anschließend wird mittels der `while`-Schleife kontinuierlich eine Tastatureingabe vom Benutzer abgefragt:

- Wird keine Taste gedrückt (Rückgabewert: `ERR`), so wartet das Programm durch Aufruf von `napms(10)` zehn Millisekunden lang, bis es mit der Ausführung fortfährt. Ohne eine derartige Verzögerung würde das Programm die Schleife kontinuierlich mit maximaler Geschwindigkeit abarbeiten und somit ständig maximale CPU-Last verursachen; mit „nur“ zehn Millisekunden Pause reduziert sich die CPU-Auslastung auf circa 1%.
- Wird eine beliebige Taste außer `q` gedrückt, so wird der *ASCII-Wert* des Zeichens und das Zeichen selbst ausgegeben. Die Darstellung funktioniert nur bei alphabetischen und numerischen Zeichen wie gewohnt, bei Funktions- und Sondertasten kann zumindest der ASCII-Wert des eingegebenen Zeichens abgefragt werden.
- Entspricht das eingegebene Zeichen dem Zeichen `q` (beziehungsweise dem ASCII-Wert `113`), so wird die Variable `quit` auf `TRUE` gesetzt. Damit ist die Negation `!quit` gleich `FALSE`, und die Schleife wird nicht fortgesetzt.

Schließlich wird das Curses-Programm mittels `endwin()` beendet.

Editor-Funktionen

Die Curses-Bibliothek stellt, da sie auf textbasierte Programme ausgerichtet ist, einige Funktionen bereit, die das Eingeben von Text ziemlich komfortabel gestalten.

Um einzelne Zeichen oder Zeilen einzugeben oder zu löschen, gibt es in Curses folgende Funktionen:

- `insch()`

  Mit `insch(c)` kann ein einzelnes Zeichen an der Stelle des Cursors eingefügt werden; der Rest der Zeile wird dabei automatisch um eine Zeichenbreite nach rechts verschoben.

- `delch()`

  Mit `delch()` wird das Zeichen an der Stelle des Cursors gelöscht; der Rest der Zeile wird dabei automatisch um eine Zeichenbreite nach links verschoben.

- `insertln()`

  Mit `insertln()` kann eine neue Zeile an der Stelle des Cursors eingefügt werden; alle folgenden Zeilen werden dabei automatisch um eine Zeile nach unten verschoben.

- `deleteln()`

Mit `deleteln()` wird die Zeile an der Stelle des Cursors gelöscht; alle folgenden Zeilen werden dabei automatisch um eine Zeile nach oben verschoben.

Möchte man an der gleichen Stelle am Bildschirm aufeinander folgende Textstellen mit unterschiedlicher Länge ausgeben, so werden durch `refresh();` nur die jeweils neu darzustellenden Zeichen auf dem Bildschirm aktualisiert; wird an der gleichen Startpositiion zunächst eine lange und danach eine kurze Textstelle ausgegeben, so bleibt bei der Ausgabe der kurzen Textstelle ein Rest der langen Textstelle bestehen.

Um den Bildschirm zu säubern, gibt es daher in Curses folgende Funktionen:

- `clrtoeol()`

  Mit `clrtoeol()` werden alle Zeichen von der Cursor-Position aus bis zum Ende der Zeile gelöscht („clear to end of line“).

- `clrtobot()`

  Mit `clrtobot()` werden alle Zeilen von der Cursor-Position aus bis zum Ende des Fensters gelöscht („clear to bottom of window“).

- `erase()` und `clear()`

  Mit `erase()` und `clear()` werden alle Zeichen auf dem gesamten Fenster gelöscht. Beide Funktionen sind nahezu identisch, `clear()` ist allerdings etwas „gründlicher“ und bewirkt, dass das Fenster beim nächsten Aufruf von `refresh()` komplett neu ausgegeben wird.

Attribute und Farben

Text kann in Curses auf den meisten Shells auch farbig oder fettgedruckt dargestellt werden. Eine solche Modifizierung wird mittels der folgenden Funktionen vorgenommen werden:

- `attron(attr)`

  Mit dieser Funktion wird das angegebene Attribut `attr` aktiviert.

- `attroff(attr)`

  Mit dieser Funktion wird das angegebene Attribut `attr` deaktiviert.

- `attrset(attr)`

  Mit dieser Funktion wird das angegebene Attribut `attr` aktiviert; alle sonstigen Attribute werden deaktiviert.

Die obigen Funktionen wirken sich auf die weitere Darstellung aller Zeichenketten aus. Um den ausgegebenen Text wieder in „normaler“ Form darzustellen, kann `attrset(A_NORMAL)` verwendet werden. Eine Übersicht aller Textattribute ist in der folgenden Tabelle zusammengestellt.

`A_NORMAL`	Normaler Text
`A_BOLD`	Text in Fettschrift und mit erhöhter Helligkeit
`A_DIM`	Text mit verringerter Helligkeit (wird nicht von jeder Shell unterstützt)
`A_REVERSE`	Text mit vertauschter Vorder- und Hintergrundfarbe
`A_UNDERLINE`	Unterstrichener Text
`A_BLINK`	Blinkender Text (wird nicht von jeder Shell unterstützt)
`A_STANDOUT`	Hervorgehobener Text (entspricht meist `A_REVERSE`)

Um mehrere Attribute miteinander zu kombinieren, können diese entweder nacheinander mittels `attron()` aktiviert werden, oder in einer einzigen `attrset()`-Anweisung durch ein binäres Oder verbunden werden; beispielsweise wird durch `attrset(A_UNDERLINE | A_BOLD);` Text künftig unterstrichen und in Fettdruck ausgegeben.

Farbiger Text

Um Text farbig auszugeben, sollte zunächst geprüft werden, ob eine farbige Darstellung von der Shell unterstützt wird. Dazu gibt es in Curses die Funktion `has_colors()`, die entweder `TRUE` oder `FALSE` als Ergebnis liefert. Ist farbiger Text auf der Shell möglich, so kann in Curses die Farbunterstützung mittels der Funktion `start_color()` freigeschaltet werden; dabei werden zugleich die in der folgenden Tabelle angegebenen Farbnamen als symbolische Konstanten definiert.

Nummer	Name	Farbe
0	`COLOR_BLACK`	Schwarz
1	`COLOR_RED`	Rot
2	`COLOR_GREEN`	Grün
3	`COLOR_YELLOW`	Gelb
4	`COLOR_BLUE`	Blau
5	`COLOR_MAGENTA`	Magenta
6	`COLOR_CYAN`	Cyan
7	`COLOR_WHITE`	Weiss

Aus diesen üblicherweise 8 Farben können mittels `init_pair()` anschließend so genannte „Farb-Paare“ definiert werden. In einem solchen Paar besteht aus einer Farbnummer für den Vordergrund (der Schriftfarbe) und einer Farbnummer für den Hintergrund, wobei anstelle der Nummern auch die oben aufgelisteten symbolischen Konstanten verwendet werden können. Beispielsweise wird mit `init_pair(1, COLOR_YELLOW, COLOR_BLUE)` ein Farben-Paar mit der Nummer `1` definiert, bei dessen Verwendung Text in gelber Farbe auf blauem Hintergrund ausgegeben wird.

Jedes so definierte Farbenpaar kann mittels `attron()` beziehungsweise `attrset()` als Text-Attribut aktiviert werden:

```
if ( has_colors() == FALSE )
    printw("Kein farbiger Text moeglich!");
else
    start_color();

init_pair(1, COLOR_YELLOW, COLOR_BLUE );
attrset( COLOR_PAIR(1) );

printw("Farbiger Text, sofern moeglich!");
```

Neben der Angabe von `COLOR_PAIR(n)`, die für das Farben-Paar mit der Nummer n steht, können ebenfalls weitere Attribute mittels eines binärem Oders angegeben werden. Wird ein Farbenpaar mit dem Attribut `A_BOLD` kombiniert, so erscheint der Text nicht nur fettgedruckt, sondern auch in einer etwas helleren Farbe; aus Schwarz wird als Vordergrundfarbe beispielsweise Grau. Bei einer gezielten Verwendung kann damit das Farbspektrum etwas erweitert werden.

Es ist auch möglich dem Hintergrund ein Farben-Paar zuzuweisen; damit ändert sich das Aussehen des Curses-Fensters, auch wenn kein Text ausgegeben wird. Die Attribute für den Hintergrund werden mit der Funktion `bkdg()` gesetzt. Wird neben einem Farbenpaar und einem binärem Oder zusätzlich ein beliebiges Zeichen angegeben, so wird der Hintergrund standardmäßig mit diesem Zeichen bedruckt:

```
bkgd( COLOR_PAIR(1) | '+' );
```

In diesem Fall würde mit den obigen Definitionen das Curses-Fenster blau erscheinen und an allen Stellen ohne Text mit gelben +-Zeichen aufgefüllt werden.

Fenster und Unterfenster

In vielen interaktiven Programmen kann man zwischen verschiedenen Ansichtsfenstern wechseln, um beispielsweise eine Datei aus einem Filebrowser-Fenster auszuwählen oder eine Hilfe-Seite zu betrachten. Für eine bessere Übersichtlichkeit im Quellcode und eine bessere Effizienz ist es empfehlenswert, für jeden derartigen Zweck ein eigenes Fenster zu verwenden, das bei einem Wechsel nicht neu geschrieben, sondern nur wieder aktualisiert werden muss.

Ein neues Fenster wird mittels der Funktion `newwin()` erstellt. Als Rückgabewert liefert diese Funktion entweder einen Zeiger auf ein `WINDOW`-Objekt, oder `NULL`, falls beim Erstellen des Fensters ein Fehler aufgetreten ist. Als Argumente für `newwin()` werden die Anzahl an Zeilen und Spalten sowie die Startposition der oberen linken Ecke des Fensters angegeben:

```
int nrows = 5;
int ncols = 20;
int starty = 3;
int startx = 5;
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
mywin = newwin(nrows, ncols, starty, startx);
wrefresh(mywin);
```

Ein neues Fenster darf nicht größer sein als das Standard-Fenster `stdscr`, und muss mindestens eine Zeile und eine Spalte beinhalten. Gibt man allerdings `newwin(0,0,0,0);` ein, so wird ein neues Fenster erzeugt, das genauso groß ist wie das Fenster `stdscr`. Damit das neue Fenster auf dem Bildschirm sichtbar wird, muss die Funktion `wrefresh()` mit dem entsprechenden Namen des Fensters aufgerufen werden. Bei Bedarf müssen zudem die Funktionen *keypad()* und *nodelay* für das jeweilige Fenster aufgerufen werden.

Die Funktionen `move()`, `addch`, `addstr()`, `printw()`, `getch()`, `getstr()` lassen sich auf ein existierende Fenster werden, wenn an ihren Funktionsname vorne ein `w` angehängt und als erstes Argument ein Zeiger auf das zu bearbeitende Fenster übergeben wird, also beispielsweise `waddstr(mywin, "Text")`.

Bei der Verwendung von mehreren sich überlappenden Fenstern ist nicht sichergestellt, dass der Text von Curses wie erwartet dargestellt wird. Es wird daher dringend empfohlen, entweder neue Fenster mit voller Fenstergröße zu erzeugen, oder das Standard-Fenster nicht zu benutzen und dafür mehrere nicht überlappende Fenster zu verwenden. Das Fenster, das zuletzt mit einem Aufruf von `wrefresh()` aktualisiert wurde, wird als „oberstes“ angezeigt und verdeckt gegebenenfalls andere Fenster.

Um ein Fenster wieder zu schließen, wird die Funktion `delwin()` verwendet, wobei als Argument wiederum ein Zeiger auf ein Fenster übergeben wird, also beispielsweise `delwin(mywin)`. Das Fenster, das nach dem Löschen aktiv angezeigt werden soll, muss dabei mittels `wrefresh()` aktualisiert werden. Gegebenenfalls muss es dazu erst mittels `touchwin(win_name)` zur vollständigen Aktualisierung vorgemerkt werden, falls ansonsten keine Änderungen vorgenommen wurden.

Unterfenster erstellen

Neben Fenstern können in Curses auch so genannte Unterfenster erstellt werden. Diese können dazu verwendet werden, um einen Teil des Hauptfensters leichter ansteuern oder mit anderen Farb- und Textattributen versehen zu können. Der Inhalt eines Unterfensters hingegen stimmt mit dem Inhalt des Hauptfensters an der jeweiligen Stelle überein.

Ein neues Unterfenster kann, ebenso wie mit *newwin()* ein neues Fenster erstellt wird, mittels `subwin()` erzeugt werden, wobei als erstes Argument der Name des übergeordneten Fensters und als weitere Argumente die Anzahl an Zeilen und Spalten sowie die Startposition der oberen linken Ecke angegeben werden:

```
// Neues Unterfenster erstellen:
my_subwin = subwin(mywin, nrows, ncols, starty, startx);

// Alternativ auch möglich:
my_subwin = derwin(mywin, nrows, ncols, starty, startx);
```

Die zweite Möglichkeit ein Unterfenster zu erstellen bietet die Funktion `derwin()`, wobei in diesem Fall die Werte `starty` und `startx` relativ zum übergeordneten Fenster (und nicht relativ zum Hauptfenster `stdscr`) angegeben werden.

Alle Funktionen, die auf ein „richtiges“ Fenster angewendet werden können, lassen sich auch auf ein Unterfenster anwenden. Unterfenster haben einen eigenen Cursor und eigene Text- und Farbattribute; sie können selbst wiederum Ausgangspunkt für neue Unterfenster sein.

Mittels `delwin(subwindow_name)` wird ein Unterfenster wieder geschlossen. Bevor ein (Haupt-)Fenster geschlossen wird, sollten zuerst auf diese Weise alle Unterfenster geschlossen werden, um Speicherlecks zu vermeiden (die Hauptfenster haben keine Informationen darüber, ob sie Unterfenster beinhalten und können diese somit nicht automatisch löschen). Der Inhalt des Subfensters, der dem Inhalt des Hauptfensters entspricht, bleibt beim Löschen erhalten.[5]

Pads

Neben normalen Fenstern gibt es in Curses auch so genannte „Pads“. Während die Funktionen für Pads weitgehend mit den für normale Fenster identisch sind, ist ihre Größe nicht auf die Größe des Hauptfensters beschränkt; die maximale Größe eines Pads ist allerdings auf 32767 Zeilen beziehungsweise Spalten beschränkt.

Ein neues Pad wird folgendermaßen erzeugt:

```
int nrows = 1000;
int ncols = 1000;
WINDOW *mypad;

// Neues Pad erstellen:
mypad = newpad(nrows, ncols);
```

Mittels den für Fenster üblichen Ausgabefunktionen, beispielsweise `waddstr()`, kann Text auf einem Pad angezeigt werden. Damit die Änderungen auf dem Bildschirm sichtbar werden, kann allerdings nicht `wrefresh()` verwendet werden, da zusätzlich angegeben werden muss, von welcher Stelle aus das Pad angezeigt werden soll: Üblicherweise ist ein Pad größer als der Bildschirm, es kann somit nur ein Ausschnitt des Pads angezeigt werden. Dies wird bei der Funktion `prefresh()` berücksichtigt:

```
prefresh(padname, pad_ymin, pad_xmin, ymin, xmin, ymax, xmax);
```

Hierbei bezeichnen `pad_ymin` und `pad_xmin` die Koordinaten der oberen linken Ecke innerhalb des Pads, von der aus der Inhalt angezeigt werden soll. Die übrigen Argumente geben die Koordinaten des Bereichs an, in dem das Pad relativ zum Hauptfenster angezeigt werden soll.

[5] Umgekehrt wird allerdings durch Funktionen wie `wclear()` der Inhalt beim Löschen des Inhalts eines Fensters automatisch auch der Inhalt aller Unterfenster gelöscht.

Subpads

Ebenso wie Fenster ein oder mehrere Unterfenster haben können, können Pads auch ein oder mehrere Subpads beinhalten. Ebenso wie bei den Unterfenstern ist der Inhalt eines Subpads mit dem Hauptpad identisch, das Subpad kann allerdings beispielsweise eigene Attribute und Farben aufweisen.

Ein neues Subpad kann mittels `subpad()` erzeugt werden:[6]

```
int nrows = 1000;
int ncols = 1000;
int subrows = 50;
int subrows = 50;
WINDOW *mypad, *my_subpad;

// Neues Pad erstellen:
mypad = newpad(nrows, ncols);

// Neues Subpad erstellen:
// Allgemeine Syntax: subpad(nrows, ncols, starty, startx)
my_subpad = subpad(mypad, 0, 0, 10, 10);
```

Bei der Verwendung von Pads und Subpads ist zu beachten, dass diese nicht innerhalb des Hauptfensters verschoben werden dürfen; die `mvwin()`-Funktion kann somit nicht auf Pads angewendet werden. Ebenso sind die `scroll()`-Funktionen für Pads nicht definiert.

Mittels `delwin(padname)` kann ein (Unter-)Pad wieder gelöscht werden. Auch hierbei sollten zunächst alle Subpads und erst zuletzt das Hauptpad gelöscht werden, um Speicherlecks zu vermeiden.

Debugging von Curses-Programmen

Curses-Programme nutzen die Shell als Ein- und Ausgabefenster; sie lassen sich daher nicht innerhalb der gleichen Shell aufrufen und mit dem *gdb*-Debugger analysieren. Folgender Trick schafft hier Abhilfe:

- Man öffnet ein zweites Shell-Fenster und gibt dort `tty` ein, um sich die Nummer dieser Shell anzeigen zu lassen; das Ergebnis lautet beispielsweise `/dev/pts/23`. Anschließend gibt man in diesem Fenster `sleep 10000000000000000000` ein, um alle weiteren Eingaben an diese Shell für eine lange Zeit zu ignorieren. (Bei Bedarf kann der Schlafmodus mittels `Strg C` abgebrochen werden.)
- Im ersten Shell-Fenster kann man dann im Projektverzeichnis wie gewohnt `gdb programmname` eingeben, um den Debugger zu starten. Als erste Debugger-Anweisung wird dann der Eingabe-und-Ausgabe-Port des zu debuggenden Programms auf den Bezeichner des zweiten Shell-Fensters festgelegt:

[6] Ein Pad kann ein Subpad, aber kein Unterfenster beinhalten. Man kann innerhalb eines Pads also mittels `subpad()` ein Subpad erzeugen, jedoch nicht mittels `subwin()` ein Unterfenster.

```
tty /dev/pts/23
```

Nun kann `run` eingeben werden, um das Programm im Debugger ablaufen zu lassen. Die Ausgabe des Programms erfolgt dabei im zweitem Shell-Fenster.

Links

Tutorials

- Ivo Oesch: Eine Einführung in die Programmiersprache C
- Wulf Alex: Einführung in C/C++
- C Tutorial
- Skript C und C++ von Prof. Mehner (PDF)
- Introduction to the C Programming Language (PDF)
- Beej's Guide to C Programming (en)

Handbücher, Nachschlagewerke

- Galileo Openbook: C von A bis Z
- The New C Standard
- The C Book
- C Wikibook
- GNU C Reference Manual
- C Standard Library with Code Examples 1
- C Standard Library with Code Examples 2

Spezielle Themen

- NCURSES Programming HOWTO
- Code-Dokumentation mittels Doxygen

Hilfe

- Stackoverflow: C Fragen und Antworten

Debugging

- Beej's Quick Guide to GDB
- Debugging with GDB (Manual)
- DGB Command Cheat Sheet

Quellen

Die Hauptquellen für diesen C-Grundkurs sind *[Oesch2008]*, *[Lopo2000]* und *[Graefe2010]*; zudem wurden die unter der Rubrik *Links* aufgeführten Seiten sowie die in der folgenden Quellenliste aufgeführten Bücher und Skripte genutzt.

Literaturverzeichnis

[Alex2008] Wulf Alex: Einführung in C/C++. General Public License, 2008.

[Erlenkoetter2003] Helmut Erlenkötter: C-Bibliotheksfunktionen sicher anwenden. Rowohlt Taschenbuch Verlag, Hamburg 2003.

[Erlenkoetter2005] Helmut Erlenkötter: C - Programmieren von Anfang an. Rowohlt Taschenbuch Verlag, Hamburg 2005.

[Gookin2007] Dan Gookin: Programmer's Guide to NCurses. Wiley, 2007.

[Graefe2010] Martin Gräfe: C und Linux. Hanser Verlag, 2010.

[Hall2007] Brian „Beej“ Hall: Beej's Guide to C Programming. Creative Commons License, 2007.

[Krucker2004] Gerhard Krucker: Einführung in die Programmiersprache C, Vorlesungskript 2004.

[Lopo2000] Erik de Castro Lopo, Peter Aitken und Bradley L. Jones: C-Programmierung für Linux in 21 Tagen. Markt+Technik Verlag, 2000.

[Oesch2008] Ivo Oesch: Eine Einführung in die Programmiersprache C und die Grundlagen der Informatik. Skript Version 2.4, 2008.

[Wolf2009] Jürgen Wolf: C von A bis Z. Rheinwerk Verlag, 2009.

Stichwortverzeichnis

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z