C von A bis Z

Das umfassende Handbuch

Kapitel 9
Kontrollstrukturen

Mit dem bisherigen Wissen können Sie die Programme immer nur sequenziell ablaufen lassen. Sie laufen also immer Zeile für Zeile ab. In diesem Kapitel wird nun der sequenzielle Programmfluss gebrochen.

Die allerersten Computer, die gebaut wurden (z. B. die Z1 von Konrad Zuse) waren sehr beschränkt in ihren Möglichkeiten. Sie waren vor allem deshalb beschränkt, weil sie nur streng sequenziell arbeiteten. Sequenziell heißt, dass die Programme (die meistens auf Lochkarten gestanzt wurden) stur von vorne nach hinten abgearbeitet wurden und dass man die einzelnen Schritte nicht in trivialer Weise wiederholen konnte (z. B. durch die Anweisung, zu einem bestimmten Schritt zurückzuspringen). Später wurden diese Mängel dann behoben, indem man die bedingten Sprünge einführte. Das heißt, es gibt im Prozessor bestimmte Vergleichsoperationen, die z. B. feststellen können, ob A größer als B ist. Ist dies der Fall, kann man davon abhängig (also immer dann, wenn A größer als B ist) an eine bestimmte Speicheradresse springen. Da alle Programme im Speicher stehen, ist es also durchaus möglich, ein Programm an einer beliebigen anderen Adresse fortzusetzen, je nachdem welche Bedingungen gerade auftreten. Hieraus ergeben sich sogenannte *Kontrollflüsse* innerhalb des Programms, die man auch auf einer höheren Sprachebene wie C abbilden kann. Die Kontrollstrukturen, die C verwendet, um den Programmfluss zu steuern, werden im Folgenden erläutert.

Sie haben folgende drei Möglichkeiten, den sequenziellen Programmfluss bewusst zu unterbrechen:

- *Verzweigungen*: Im Programm wird eine Bedingung definiert, die entscheidet, an welcher Stelle das Programm fortgesetzt werden soll.
- *Schleifen (Iteration)*: Ein Anweisungsblock wird so oft wiederholt, bis eine bestimmte Abbruchbedingung erfüllt wird.
- *Sprünge*: Die Programmausführung wird mithilfe von Sprungmarken an einer anderen Position fortgesetzt. Sprünge in einem Programm gelten mittlerweile als schlechter Stil und werden wahrscheinlich im nächsten C-Standard entfernt. Sie sind auch nicht notwendig. Mit dem gerade Gesagten meine ich *direkte Sprünge*. Mit Schlüsselwörtern wie `return`, `break`, `continue`, `exit` und der Funktion `abort()` können jedoch nach wie vor kontrollierte Sprünge ausgeführt werden.

9.1 Verzweigungen mit der »if«-Bedingung

Die `if`-Bedingung (engl. *wenn/falls*) hat folgende Syntax:

```
if(BEDINGUNG == wahr) {
   Anweisung1;
 }
Anweisung2;
```

Wenn die Bedingung zwischen den Klammern des `if`-Schlüsselwortes wahr (*true*) ist, wird `Anweisung1` im Anweisungsblock ausgeführt. Anschließend wird das Programm bei `Anweisung2`, also nach dem Anweisungsblock, fortgesetzt. Ist die Bedingung unwahr (*false*), wird `Anweisung1` nicht ausgeführt, und das Programm fährt sofort mit `Anweisung2` fort. Die geschweiften Klammern kann man nicht weglassen, es sei denn, `Anweisung1` befindet sich in derselben Zeile wie das `if` und die Zeile schließt mit einem Semikolon ab. Es ist aber trotzdem ratsam, die geschweiften Klammern immer zu setzen, denn das erhöht die Lesbarkeit des Codes. Auch ist es ratsam, die Anweisungen, die sich innerhalb des `if`-Blocks befinden, mit der Tabulatortaste einzurücken. Auch dies erhöht die Lesbarkeit des Codes.

9.1.1 Anweisungsblöcke

In einem Anweisungsblock werden Anweisungen (mindestens eine) zusammengefasst. Als Anweisungsblock gelten alle Anweisungen, die zwischen geschweiften Klammern ({}) stehen. Anweisungsblöcke lassen sich auch ineinander verschachteln. Ich empfehle Ihnen, beim Erstellen eines Programms auf eine saubere Strukturierung (d. h. entsprechendes Einrücken bei den Anweisungen eines Blocks) zu achten, um den Überblick zu wahren.

Den Programmablauf der `if`-Bedingung können Sie sich schematisch so vorstellen wie in Abbildung 9.1. Diese Darstellung wird auch als *Programmablaufplan* (kurz: PAP) bezeichnet. Es gibt für jedes Betriebssystem freie Tools, mit denen man solche Diagramme erzeugen kann.

Sehen Sie sich dazu das folgende Programmbeispiel an:

```
/* if_1.c */
#include<stdio.h>

int main(void) {
   unsigned int alter;

   printf("Wie alt sind Sie: ");
   scanf("%u", &alter);
```

```
   /* ... noch jünger als 18? */
   if(alter < 18) {
      printf("Sie sind noch nicht volljährig\n");
   }
    printf("Bye\n");
    return 0;
}
```

Listing 9.1 »if_1.c« demonstriert eine Altersabfrage mit der »if«-Anweisung.

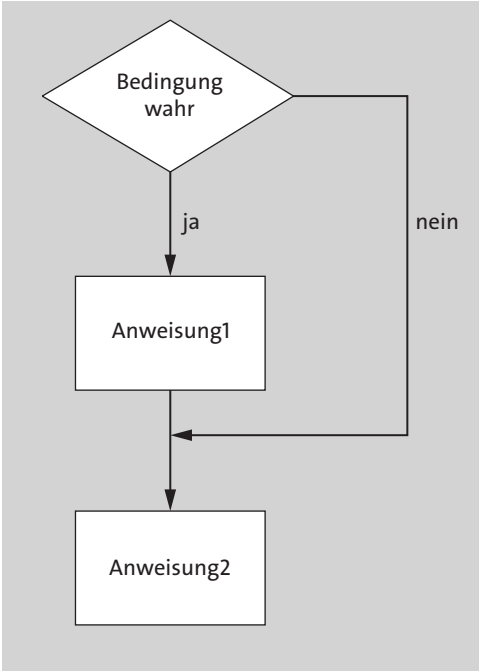

Abbildung 9.1 Programmablaufplan zur »if«-Anweisung

Das Programm fordert Sie auf, Ihr Alter einzugeben. In der Bedingung `if(zahl < 18)` wird überprüft, ob Sie jünger als 18 Jahre sind. Sind Sie jünger, wird die `printf()`-Anweisung im Anweisungsblock mit entsprechender Ausgabe bearbeitet. Wurde aber ein Wert eingegeben, der größer oder gleich 18 ist, wird nur die `printf()`-Anweisung hinter dem Anweisungsblock ausgeführt, die »Bye« ausgibt. Abbildung 9.2 zeigt den Programmablaufplan zu diesem Beispiel:

Jetzt soll das Programm um einige `if`-Bedingungen erweitert werden:

```
/* if_2.c */
#include<stdio.h>
```

```
int main(void) {
   unsigned int alter;

   printf("Wie alt sind Sie: ");
   scanf("%u", &alter);

   if(alter < 18)  {
      printf("Sie sind noch nicht volljährig\n");
   }
   if(alter > 18)  {
      printf("Sie sind volljährig\n");
   }
   if(alter == 18) {
      printf("Den Führerschein schon bestanden?\n");
   }
   printf("Bye\n");
   return 0;
}
```

Listing 9.2 »if_2.c« erweitert die Altersabfrage durch eine zweite »if«-Abfrage, um zu ermitteln, ob man den Führerschein mit 18 schon bestanden hat.

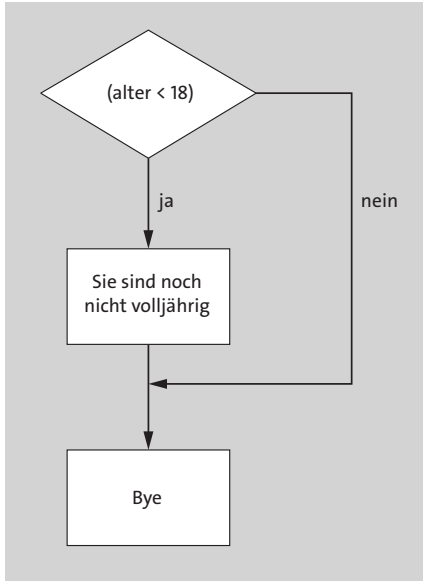

Abbildung 9.2 Programmablaufplan zum Listing

Es ist offensichtlich, wie das Programm vorgeht: Sie geben das Alter ein, und mithilfe der einzelnen `if`-Bedingungen testet es, ob die eingegebene Zahl größer, kleiner oder

gleich 18 ist. Bevor das Programm etwas genauer analysiert wird, sollten Sie sich Tabelle 9.1 mit den bisher verwendeten Vergleichsoperatoren und ihrer jeweiligen Bedeutung ansehen.

Vergleichsoperator	Bedeutung
`a < b`	wahr, wenn `a` kleiner als `b`
`a <= b`	wahr, wenn `a` kleiner oder gleich `b`
`a > b`	wahr, wenn `a` größer als `b`
`a >= b`	wahr, wenn `a` größer oder gleich `b`
`a == b`	wahr, wenn `a` gleich `b`
`a != b`	wahr, wenn `a` ungleich `b`

Tabelle 9.1 Übersicht zu Vergleichsoperatoren (relationale Operatoren)

Zurück zum Programm. Die erste Anweisung

```
if(alter < 18)
```

testet, ob die Zahl, die Sie eingegeben haben, kleiner als 18 ist. Ist die Bedingung wahr, springt das Programm unmittelbar in den für diese Bedingung geschriebenen Anweisungsblock, der in den geschweiften Klammern steht. Falls also die eingegebene Zahl kleiner als 18 ist, wird der Anweisungsblock

```
{
   printf("Sie sind noch nicht volljährig\n");
}
```

ausgeführt. Besteht der Anweisungsblock hinter der Kontrollstruktur lediglich aus einer einzigen Anweisung, so können die geschweiften Klammern auch weggelassen werden:

```
if(alter < 18)
   printf("Sie sind noch nicht volljährig\n");
```

Diese Schreibweise ist korrekt und wird auch häufig verwendet. Sie sollten sich jedoch klarmachen, dass sie sehr leicht zur Fehlerquelle werden kann – nämlich dann, wenn Sie eine weitere Anweisung hinzufügen wollen, die auch nur in dem Fall ausgeführt werden soll, wenn die `if`-Bedingung wahr ist, wie etwa im folgenden Beispiel:

```
if(alter < 18)
   printf("Sie sind noch nicht volljährig\n");
   printf("Sollte nur erscheinen, wenn (alter < 18)!\n");
```

Wenn – wie hier – die geschweiften Klammern vergessen werden, wird die zweite `printf()`-Anweisung immer ausgeführt, da die `if`-Bedingung im Wahr-Fall wirklich nur eine weitere Anweisung ausführt.

Korrekt müsste es folgendermaßen aussehen:

```
if(alter < 18) {
   printf("Sie sind noch nicht volljährig\n");
   printf("Sollte nur erscheinen, wenn (alter < 18)!\n");
}
```

Jetzt wird auch die zweite Anweisung nur ausgeführt, wenn die `if`-Bedingung wahr ist.

Die nächsten beiden `if`-Bedingungen im Programmbeispiel verhalten sich analog. Hiermit wird getestet, ob die eingegebene Zahl für das Alter kleiner bzw. gleich 18 ist. Sollte sich einer dieser Fälle als wahr herausstellen, so wird ebenfalls der jeweils zugehörige Anweisungsblock ausgeführt.

[!]

Achtung bei Vergleichen

Ein häufig gemachter Fehler ist `if(alter=18)`: Der Variablen `alter` wurde der Wert 18 zugewiesen, und der Anweisungsblock wird ausgeführt. Dabei ist es jedoch egal, ob der Wert gleich 18 ist, denn `alter` wird jetzt der Wert 18 zugewiesen. Diese `if`-Abfrage ist also immer wahr. Der Fehler ist zudem schwer auffindbar, da die Syntax absolut korrekt ist, der Compiler also keine Fehlermeldung anzeigt.

Eine oft verwendete und empfohlene Maßnahme ist es, die Überprüfung umzustellen: `if (18 == alter)` ist syntaktisch korrekt und bewirkt das Gleiche. Hiermit würde der Compiler sofort eine Fehlermeldung anzeigen, falls Sie versehentlich `if (18 = alter)` schreiben, da sich einer Zahl keine Variable zuordnen lässt. Diese Umstellungsmethode funktioniert nur bei `==`-Vergleichen, nicht bei den anderen Operatoren, und es ist außerdem Geschmackssache, ob Sie diese Methode verwenden.

9.2 Die Verzweigung mit »else if«

Was ist, wenn die erste Bedingung im Listing zuvor wahr ist, d. h. die Zahl größer als 18 ist? Dann nimmt das Programm als nächsten Schritt dennoch die Überprüfung vor, ob die Zahl kleiner als 18 und gleich 18 ist. Das ist eigentlich nicht mehr notwendig. Sie können dies mit `else if` verbessern:

```
else if(alter > 18)  {
   printf("Sie sind volljährig\n");
}
```

```
   else if(alter == 18) {
      printf("Den Führerschein schon bestanden?\n");
   }
```

Hier sehen Sie die Syntax dazu:

```
if(BEDINGUNG1 == wahr) {
    Anweisung1;
}
else if(BEDINGUNG2 == wahr) {
    Anweisung2;
}
Anweisung3;
```

Ist die `Bedingung1` wahr, wird die `Anweisung1` im Anweisungsblock ausgeführt und die Kontrollstruktur ist fertig. Ist `Bedingung1` nicht wahr, wird die `Bedingung2` überprüft. Ist `Bedingung2` wahr, wird `Anweisung2` ausgeführt und das Programm endet. Ist aber auch `Bedingung 2` nicht wahr, wird die `Anweisung 3` ausgeführt.

Das Programmbeispiel sieht in der neuen Fassung so aus:

```
/* if_3.c */
#include<stdio.h>

int main(void) {
   unsigned int alter;

   printf("Wie alt sind Sie: ");
   scanf("%u", &alter);

   if(alter < 18) {
      printf("Sie sind noch nicht volljährig\n");
   }
   else if(alter > 18)  {
      printf("Sie sind volljährig\n");
   }
   else if(alter == 18) {
      printf("Den Führerschein schon bestanden?\n");
   }
   printf("Bye\n");
   return 0;
}
```

Listing 9.3 »if_3.c« erweitert »if_2.c« durch einige »else«-Abfragen.

Abbildung 9.3 zeigt den Programmablaufplan dazu.

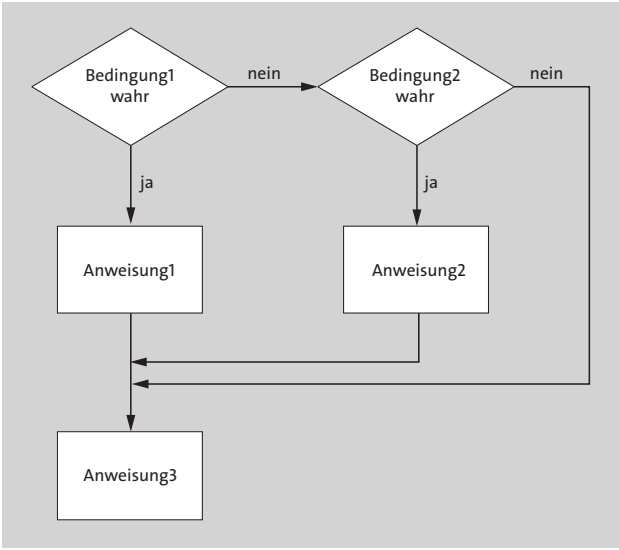

Abbildung 9.3 Programmablaufplan mit »if else if«

9.3 Die Verzweigung mit »else«

Eigentlich hätten Sie sich im vorangegangenen Programmbeispiel folgende Bedingung sparen können:

```
else if(alter == 18)
```

Denn wenn die Variable `alter` nicht größer als 18 und auch nicht kleiner als 18 ist, kann diese nur noch gleich 18 sein. Um sich eine solche Bedingung zu ersparen, gibt es die `else`-Verzweigung, was so viel heißt wie *Ansonsten nimm mich* oder *Andernfalls tue das Folgende*.

Sie könnten also die Zeile `else if(alter == 18)` ersetzen durch:

```
else {
   printf("Den Führerschein schon bestanden?\n");
}
```

Die Syntax der `else`-Verzweigung sieht folgendermaßen aus:

```
if(BEDINGUNG == wahr) {
    Anweisung1;
}
else {
    Anweisung2;
}
```

Sehen Sie sich hierzu auch den Programmablaufplan in Abbildung 9.4 an.

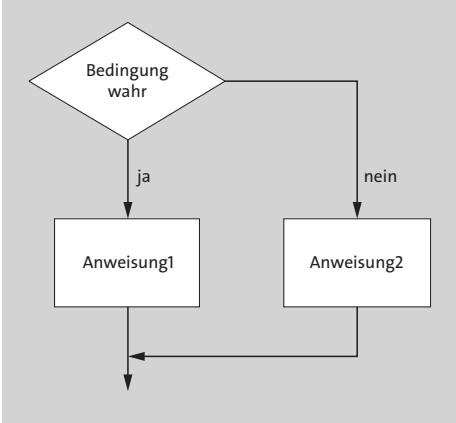

Abbildung 9.4 Programmablaufplan mit »else«

Merke

`else` folgt immer einem vorausgehenden `if` oder `else if`.

Umgeschrieben sieht das Listing jetzt wie folgt aus:

```
/* if_4.c */
#include<stdio.h>

int main(void) {
   unsigned int alter;

   printf("Wie alt sind Sie: ");
   scanf("%u", &alter);

   if(alter < 18) {
      printf("Sie sind noch nicht volljährig\n");
   }
   else if(alter > 18) {
      printf("Sie sind volljährig\n");
   }
   else {
      printf("Den Führerschein schon bestanden?\n");
   }
   printf("Bye\n");
   return 0;
}
```

Listing 9.4 »if_4.c« zeigt die »optimierte« Version von »if_3.c«.

Jetzt haben Sie die einzelnen Bedingungen im Programm optimal positioniert. Wird z. B. für das Alter der Wert 16 eingegeben, führt das Programm den entsprechenden Anweisungsblock aus. Danach folgen keine weiteren Überprüfungen. Wird hingegen für das Alter der Wert 20 genannt, ersparen Sie sich wenigstens die letzte Überprüfung, ob das Alter gleich 18 ist, und der entsprechende Anweisungsblock wird ausgeführt. Trifft keine dieser Bedingungen zu, wird der else-Anweisungsblock ausgeführt. Für das bessere Verständnis zeigt Abbildung 9.5 einen Programmablaufplan zu diesem Listing.

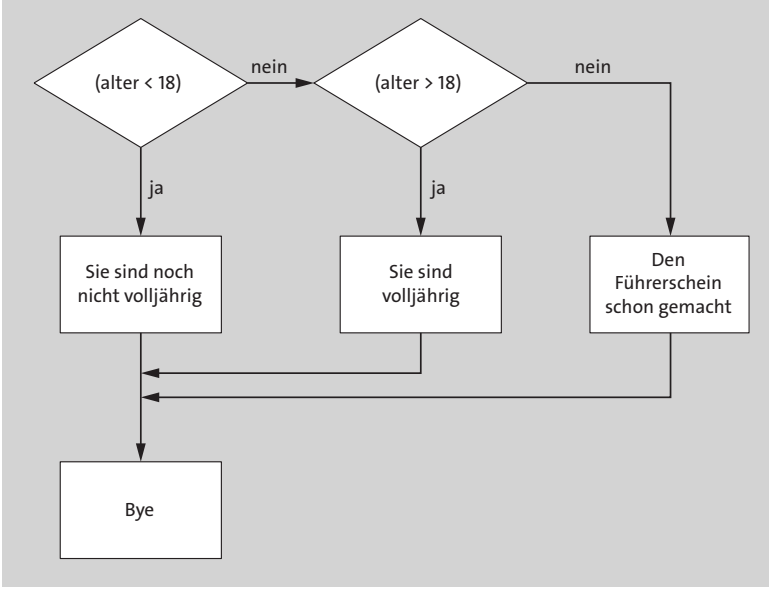

Abbildung 9.5 Programmablaufplan zu Listing 9.4

Häufig wird der Fehler gemacht, hinter eine Bedingungsanweisung ein Semikolon zu setzen:

```
else if(alter > 18);   //Fehler (das Semikolon ist fehl am Platz)
   {
      printf("Sie sind volljährig\n");
   }
```

Der Compiler wird Ihnen in solchen Fällen die Fehlermeldung ausgeben, dass else hier fehl am Platze ist, denn einer else-Verzweigung muss immer ein if bzw. else if vorausgehen. Sie können probeweise im Listing hinter der else-Verzweigung ein Semikolon setzen. Das Programm wird dann immer ausgeben, dass die Variable gleich 18 ist. Dies wird zusätzlich ausgegeben, auch wenn eine der vorangegangenen Bedingungen bereits wahr gewesen ist.

Es ist auch möglich, mehrere Anweisungen bzw. Anweisungsblöcke ineinander zu verschachteln. Das heißt, eine Anweisung mit Anweisungsblock steht innerhalb einer anderen Anweisung mit Anweisungsblock. Listing 9.5 enthält eine solche Verschachtelung:

```
/* if_5.c */
#include<stdio.h>

int main(void) {
   unsigned int alter;

   printf("Wie alt sind Sie: ");
   scanf("%u", &alter);

   if(alter <= 18) {
      if(alter == 18) {
         printf("Den Führerschein schon bestanden?\n");
      }
      else {
         printf("Sie sind noch nicht volljährig\n");
      }
   }
   else {
      printf("Sie sind volljährig\n");
   }
   printf("Bye\n");
   return 0;
}
```

Listing 9.5 »if_5.c« demonstriert, wie Sie Anweisungen ineinander verschachteln können.

Das Listing bewirkt das Gleiche wie schon das Beispiel zuvor. Mit der Anweisung

```
if(alter <= 18)
```

wird überprüft, ob `alter` kleiner oder gleich `18` ist. Sollte das der Fall sein, verzweigt das Programm weiter in den Anweisungsblock mit der Abfrage, ob `alter` gleich `18` ist. Wenn nicht, ist `alter` kleiner als `18`. Sollte die Zahl aber größer als `18` sein, geht es gleich zur `else`-Verzweigung weiter.

Wie Sie die Anweisungsblöcke anordnen, bleibt Ihnen letztlich selbst überlassen. Sehen Sie sich dennoch einmal folgendes Negativ-Beispiel an:

```
/* if_6.c */
#include<stdio.h>

int main(void) {
   unsigned int alter;
   printf("Wie alt sind Sie: ");
   scanf("%u", &alter);

   if(alter <= 18){
   if(alter == 18){
   printf("Den Führerschein schon bestanden?\n");}
   else{
        printf("Sie sind noch nicht volljährig\n");
       }}
   else{printf("Sie sind volljährig\n");}
   printf("Bye\n");
   return 0;
}
```

Das Programm ist zwar nicht falsch, aber dennoch etwas unübersichtlich.

9.4 Der !-Operator (logischer NOT-Operator)

Den logischen !-Operator (NOT-Operator) haben Sie eben schon kennengelernt. Dieser Operator wird oft falsch verstanden, weswegen ihm ein eigener Abschnitt gewidmet ist. Der !-Operator ist ein unärer Operator und kann eine Bedingung negieren. Dies bedeutet, er kann aus »wahr« »falsch« machen und umgekehrt. Dazu ein Programmbeispiel:

```
/* logik_not_1.c */
#include<stdio.h>

int main(void) {
   int checknummer;

   printf("Bitte geben Sie Ihren Codeschlüssel ein: ");
   scanf("%d", &checknummer);

   if( ! (checknummer == 4711) ) {
      printf("Error - Falscher Codeschlüssel \n");
   }
```

```
   else {
      printf("Success -  Login erfolgreich \n");
    }
   return 0;
}
```

Listing 9.6 »logik_1.c« demonstriert die Verwendung des logischen »!«-Operators.

Zur Erklärung der `if`-Bedingung im Programm:

```
if( !(checknummer == 4711) )
```

Hier wird der Ausdruck zwischen den Klammern geprüft. Das bedeutet, der `!`-Operator überprüft den Wert in der Klammer und gibt `1` (wahr) zurück, falls der Wert in der Klammer NICHT `4711` ist. Das NICHT ist sehr wichtig, denn genau dies erreicht der `!`-Operator. Ist der Wert aber gleich `4711`, dann wird `0` (falsch) zurückgegeben. Das Programm fährt daraufhin mit der `else`-Verzweigung fort und gibt aus, dass Sie die richtige Zahl eingegeben haben.

Tabelle 9.2 zeigt die verschiedenen Verwendungsmöglichkeiten.

Anweisung	==	Anweisung
`if(a != 0)`	gleich	`if(a)`
`if(a == 0)`	gleich	`if(!a)`
`if(a > b)`	gleich	`if(! (a <= b) )`
`if( (a-b) == 0)`	gleich	`if(! (a-b) )`

Tabelle 9.2 Darstellung von Wahrheitswerten

Ein weiteres Programmbeispiel verdeutlicht dies:

```
/* logik_not_2.c */
#include<stdio.h>

int main(void) {
   int zahl1, zahl2;

   printf("Bitte Zahl 1 eingeben: ");
   scanf("%d", &zahl1);
   printf("Bitte Zahl 2 eingeben: ");
   scanf("%d", &zahl2);
```

```
   if(!zahl1)
      printf("Error: Der Wert ist gleich 0!! \n");
   else if(!zahl2)
      printf("Error: Der Wert ist gleich 0!! \n");
   else
      printf("%d/%d = %f \n", zahl1, zahl2, (float) zahl1/zahl2);
   return 0;
}
```

Sie vermeiden mit diesem Programm eine Division durch 0. Sollte also keine der beiden Zahlen dem Wert 0 entsprechen, wird mit `(float) zahl1/zahl2` eine Division durchgeführt. Sie verwenden hier ein explizites Typecasting, damit der Wert nach dem Komma nicht einfach abgeschnitten wird.

Tipp: »not«

Als alternative Schreibweise für den logischen `!`-Operator können Sie auch das Makro `not` verwenden, das in der Headerdatei *<iso646.h>* definiert ist. Die Schreibweise `if( !a )` entspricht somit exakt `if( not a )`.

9.5 Logisches UND (&&) – logisches ODER (||)

Sie haben sicher schon bemerkt, dass es in C viele Operatoren gibt. So ist die Sprache zwar unter Umständen schwerer lesbar, aber Sie können mit ihr auch schneller, vielseitiger und effektiver programmieren. Sobald Sie die logischen Operatoren kennengelernt haben, werden Sie diese sehr zu schätzen wissen.

Mit dem logischen ODER-Operator (`||`) werden Operanden so miteinander verknüpft, dass der Ausdruck »wahr« zurückliefert, wenn mindestens einer der Operanden wahr ist. Dabei können die Operanden auch durchaus mathematisch komplexe Ausdrücke sein, die wiederum Operanden enthalten. C löst auch komplexe Ausdrücke automatisch auf, ohne dass Sie sich darum kümmern müssen. Die Überprüfung ist rein logischer Art und es werden im Gegensatz zu dem einfachen `&` und `|` keine Bits geändert! Genau deshalb (weil eben nur Aussagen auf ihren Wahrheitsgehalt überprüft werden) spricht man von *logischen Operatoren*. Ein kleines Beispiel dazu:

```
if( (Bedingung1) || (Bedingung2) )
   /* mindestens eine der Bedingungen ist wahr */
else
   /* keine Bedingung ist wahr */
```

Sehen Sie sich dazu in Abbildung 9.6 den Programmablaufplan des logischen ODER-Operators an.

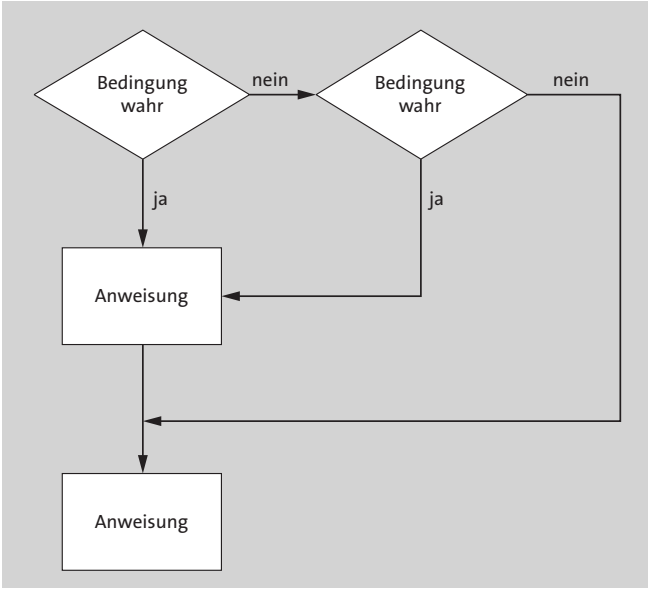

Abbildung 9.6 Der Programmablaufplan des logischen ODER-Operators

Jetzt wird der Operator in dem Programm eingesetzt, das Sie beim logischen NICHT-Operator zuvor verwendet haben:

```
/* logik_or.c */
#include<stdio.h>

int main(void) {
   int zahl1, zahl2;
   printf("Bitte Zahl 1 eingeben: ");
   scanf("%d", &zahl1);
   printf("Bitte Zahl 2 eingeben: ");
   scanf("%d", &zahl2);

   if( (!zahl1) || (!zahl2) )
      printf("Error: Einer der Werte ist gleich 0!!! \n");
   else
      printf("%d/%d = %f \n", zahl1, zahl2, (float)zahl1/zahl2);
   return 0;
}
```

Listing 9.7 »logik_or.c« demonstriert die Verwendung des logischen »||«-Operators.

Die `if`-Konstruktion des Programms sieht so aus:

```
if( (!zahl1) || (!zahl2) )
```

In Worten ausgedrückt, sähe das etwa folgendermaßen aus: Ist der Wert `zahl1` gleich `0` ODER der Wert `zahl2` gleich `0`, dann ist die Bedingung wahr, und Sie haben eine Null eingegeben. Sollte die erste Bedingung (`!zahl1`) schon wahr sein, so wird die zweite Bedingung (`!zahl2`) gar nicht mehr überprüft: Dies können Sie auch am Programmablaufplan erkennen.

Analog verhält es sich mit dem logischen UND-Operator (`&&`). Das Ergebnis dieser verknüpften Operanden gibt nur dann »wahr« zurück, wenn alle Operanden wahr sind. Den Programmablaufplan des logischen UND-Operators sehen Sie in Abbildung 9.7.

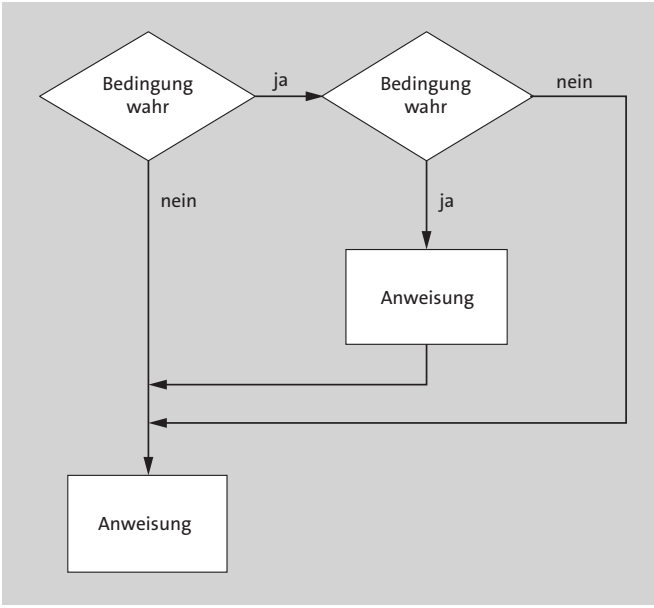

Abbildung 9.7 Programmablaufplan des logischen UND-Operators

Wiederum folgt zur Verdeutlichung ein Programmbeispiel:

```
/* logik_and.c */
#include<stdio.h>

int main(void) {
   int zahl;
   printf("Geben Sie einen Wert zwischen 10 und 20 ein: ");
   scanf("%d", &zahl);
```

```
   if( (zahl >= 10) && (zahl <= 20) )
      printf("Danke für die Eingabe! \n");
   else
      printf("Falsche Eingabe! \n");
   return 0;
}
```

Listing 9.8 »logik_and.c« demonstriert die Verwendung des logischen »&&«-Operators.

In der Zeile

```
if( (zahl >= 10) && (zahl <= 20) )
```

prüfen Sie, ob die eingegebene Zahl einen Wert zwischen 10 und 20 besitzt. In Worten: Ist es wahr, dass die Zahl größer oder gleich 10 ist UND die Zahl auch kleiner gleich 20 ist, dann ist die Bedingung wahr.

Natürlich können Sie mit dem &&-Operator und dem ||-Operator weitere Bedingungen miteinander verknüpfen. Allerdings sollten Sie dabei die Lesbarkeit eines solchen Konstrukts im Auge behalten und über folgende Frage nachdenken: Was ist lesbarer, eine verschachtelte `if`- Abfrage oder doch eine verkettete und eventuell mit einigen Klammern versehene logische Auswertung? Meistens ist eine logische Auswertung besser verständlich als verschachtelte Blöcke, aber eben nur meistens.

Tipp: »and« und »or«

Als alternative Schreibweise für die logischen Operatoren && und || können Sie seit dem C99-Standard auch die Makros `and` und `or` verwenden, die beide in der Headerdatei *<iso646.h>* definiert sind.

9.6 Der Bedingungsoperator »?:«

Der Bedingungsoperator `?:` ist der einzige ternäre Operator in ANSI C. Im Prinzip repräsentiert dieser Operator nichts anderes als eine Kurzform der `if else`-Anweisung. Seine Syntax sieht so aus:

```
<BEDINGUNG> ? <ANWEISUNG
                    1> : <ANWEISUNG
                    2>
```

Wenn die `BEDINGUNG` wahr ist, wird die `ANWEISUNG1` ausgeführt. Ansonsten wird `ANWEISUNG2` ausgeführt. Der Programmablaufplan ist identisch mit dem der `if else`-Anwei-

sung. Sie benötigen beispielsweise von zwei Zahlen den höheren Wert? Mit dem Bedingungsoperator ?: könnten Sie folgendermaßen arbeiten:

```
max = (a>b) ?a :b;
```

Diese Schreibweise ist äquivalent zu der folgenden:

```
if(a > b)
   max=a;
else
   max=b;
```

Ein Listing soll das Prinzip verdeutlichen:

```
/* max_val_1.c */
#include<stdio.h>

int main(void) {
   int a=5,b=10;
   int max;
   max = (a > b) ? a : b;
   printf("Der größte Wert ist %d \n", max);
   return 0;
}
```

Natürlich ist es auch hier möglich, die einzelnen Ausdrücke ineinander zu verschachteln:

```
/* max_val_2.c */
#include<stdio.h>

int main(void) {
   int a=8, b=3, c=76, big;
   printf("Die größte Zahl ist....");
   big = (a>b) ? ((a>c) ?a :c) : ((b>c) ?b :c);
   printf("..%d\n", big);
   return 0;
}
```

Auch hier kann bei mehrfacher Verschachtelung die Übersichtlichkeit und Nachvollziehbarkeit des Programms leiden; `if else`-Anweisungen wären besser geeignet. Sehen Sie sich folgende Codezeile näher an:

```
big = (a>b) ? ((a>c) ?a :c) : ((b>c) ?b :c);
```

Bevor die Variable big einen Wert zugewiesen bekommt, werden zuerst folgende Bedingungen überprüft: Ist der Wert von a größer als der von b, wird überprüft, ob der Wert von a auch größer als der von c ist. Ist das der Fall, so ist a der größte Wert. Ist dies nicht der Fall, ist c der größte Wert. Sollte aber in der ersten Bedingung a nicht größer als b sein, so wird überprüft, ob b größer als c ist. Ist b größer als c, haben Sie den größten Wert gefunden. Ist b nicht größer als c, bleibt nur noch c als größte Zahl übrig.

Häufig wird der Bedingungsoperator auch wie folgt verwendet:

```
printf("Bitte geben Sie eine Zahl ein: ");
scanf("%d",&zahl);
printf("Die Zahl, die Sie eingegeben haben, ist ");
(zahl%2) ? printf("ungerade \n") : printf("gerade \n");
```

9.7 Fallunterscheidung: die »switch«-Verzweigung

Was ist zu tun, wenn unterschiedliche Bedingungen mehrere verschiedene Zahlen beachten sollen? Sie könnten entsprechend viele if-Anweisungen verwenden. Aber warum so umständlich, wenn es auch einfacher geht? Für solche Aufgaben gibt es die Fallunterscheidung mit switch, was auf Deutsch so viel wie »Schalter« heißt. Die Syntax von switch sieht so aus:

```
switch(AUSDRUCK) {
     AUSDRUCK_1 : anweisung_1; break;
     AUSDRUCK_2 : anweisung_2; break;
     AUSDRUCK_3 : anweisung_3; break;
     ...
     AUSDRUCK_n : anweisung_n; break;
     default: anweisung;
  }
```

Der AUSDRUCK in den Klammern nach switch wird in dem hierauf folgenden Anweisungsblock ausgewertet. Sollte der AUSDRUCK==AUSDRUCK_1 sein, so wird anweisung_1 ausgeführt, die mit break endet. Sollte aber AUSDRUCK==AUSDRUCK_2 sein, so wird anweisung_2 ausgeführt, die mit break endet, usw.

break darf nicht fehlen, weil es auch Anweisungsblöcke mit mehreren Anweisungen hinter case geben kann, und dieser Block muss eben durch break beendet werden (und nicht durch eine geschweifte Klammer!).

Jetzt kann es natürlich auch vorkommen, dass keine einzige Bedingung erfüllt ist, die durch case definiert wurde. In diesem Fall wird mit den Anweisungen weiterge-

macht, die hinter `default` stehen (hierzu gleich mehr). Ich zeige Ihnen erst einmal ein einfaches Programm, das Sie nach den Zahlen 1 bis 5 fragt:

```
/* switch_1.c */
#include<stdio.h>

int main(void) {
   int a;
   printf("Bitte eine Zahl von 1-5 eingeben: ");
   scanf("%d", &a);

   switch(a) {
      case 1: printf("Das war eins \n");
              break;
      case 2: printf("Das war zwei \n");
              break;
      case 3: printf("Das war drei \n");
              break;
      case 4: printf("Das war vier \n");
              break;
      case 5: printf("Das war fünf \n");
              break;
      }      /* Ende switch */
   return 0;
}
```

Listing 9.9 »switch_1.c« zeigt Ihnen, wie Sie eine einfache Fallunterscheidung mit »switch« realisieren können.

Nach der Eingabe einer Zahl wird diese im »Schalter« ausgewertet mit:

```
switch(a)
```

Die Auswertung steht wieder in einem Anweisungsblock in geschweiften Klammern. Zuerst wird

```
case 1: printf("Das war eins \n");
```

ausgewertet. `case` heißt auf Deutsch »falls«, und um dies von »if« zu unterscheiden, musste ein anderer Ausdruck her. Zuerst wird geprüft, ob der Fall, der in `switch(a)` steht, zutrifft (also hier, ob `a==1`). Wenn ja, wird das Programm den entsprechenden Text ausgeben.

```
break;
```

Die break-Anweisung am Ende eines jeden Falls bedeutet: Ab hier springt das Programm aus dem Anweisungsblock heraus und setzt seine Ausführung nach dem switch-Anweisungsblock fort. Dies ist sehr wichtig, denn sonst würde unser Programm einfach das nächste case auswerten, was falsch wäre. In unserem Fall (wenn die eingegebene Zahl 1 ist) endet das Programm bereits. Wenn break also nicht nach jedem Fall angegeben wird, wird der nächste Fall auch mit ausgegeben, und das wollen wir ja nicht! Sehen Sie sich einen Programmablaufplan dazu an (siehe Abbildung 9.8).

Das Weglassen von break kann aber durchaus gewollt sein. Dies hat den Vorteil, dass Sie in gewisser Weise auf mehrere Fälle gleichartig reagieren können. Dazu wird das obige Programm leicht abgewandelt:

```
/* switch_2.c */
#include<stdio.h>

int main(void) {
   int a;
   printf("Bitte eine Zahl von 1-5 eingeben: ");
   scanf("%d", &a);

   switch(a) {
      case 1: printf("Das war eins oder...");
      case 2: printf("...zwei \n");
              break;
      case 3: printf("Das war drei \n");
              break;
      case 4: printf("Das war vier...");
      case 5: printf("...oder fünf \n");
              break;
      }      /* Ende switch */
   return 0;
}
```

Listing 9.10 »switch_2.c« demonstriert, wie Sie mit »switch« auf mehrere Fälle gleichzeitig reagieren können.

Geben Sie in diesem Programm beispielsweise den Wert 4 für die Variable a ein, so trifft sowohl der Fall case 4 als auch case 5 zu, weil bei der Fallunterscheidung case 4 das break nach der printf()-Ausgabe weggelassen wurde. Ebenso würde sich das Programm verhalten, wenn Sie den Wert 1 eingegeben hätten.

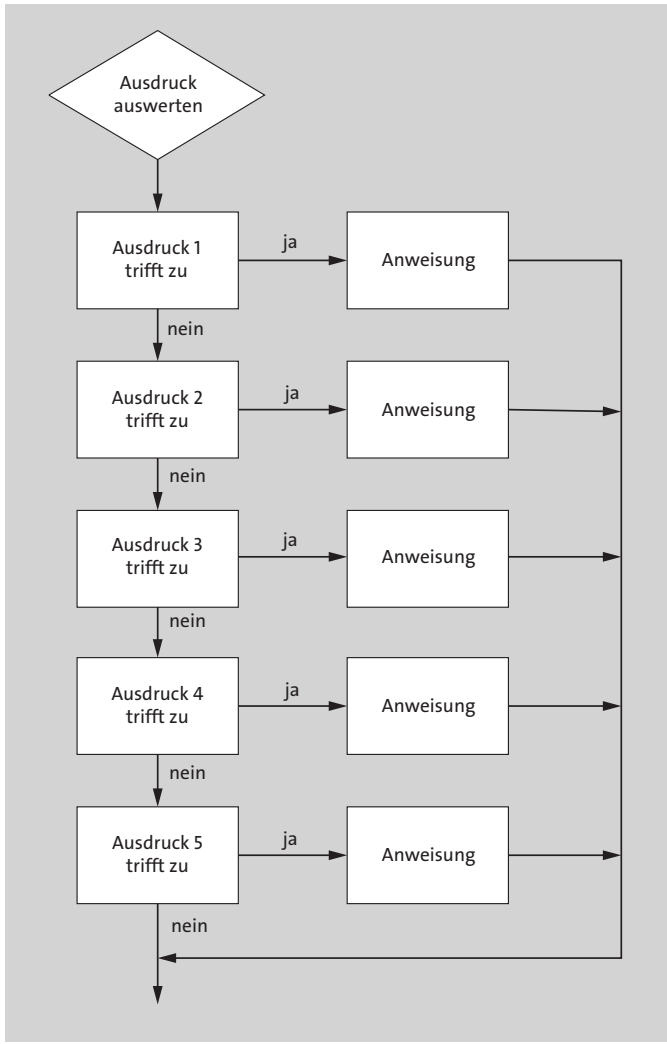

Abbildung 9.8 Programmablaufplan zur »switch«-Fallunterscheidung

9.7.1 default

Jetzt stellt sich aber nach wie vor die Frage, was geschieht, wenn Sie einen anderen Wert als 1 bis 5 eingeben. Das Programm einfach ohne eine Meldung zu beenden, ist unschön, denn auch bei falschen Werten sollte eine Reaktion erfolgen. Dafür gibt es das Schlüsselwort `default`, das so viel wie »Standardverhalten« heißt. Testen Sie es am besten gleich in einem Programm:

```
/* switch_3.c */
#include<stdio.h>
```

```
int main(void) {
   int a,b;
   char opera;
   printf("Grundrechenarten \n");
   printf(" (zahl)(Operator)(zahl) ohne Leerzeichen \n");

   printf("Rechnung bitte eingeben : ");
   scanf("%d%c%d", &a, &opera, &b); /* Bsp.: 10+12 */

   switch(opera) {
      case '+': printf("%d + %d = %d \n",a ,b ,a+b);
                break;
      case '-': printf("%d - %d = %d \n", a, b, a-b);
                break;
      case '*': printf("%d * %d = %d \n", a, b, a*b);
                break;
      case '/': printf("%d / %d = %d \n", a, b, a/b);
                break;
      default: printf("%c? kein Rechenoperator \n", opera);
      }      /* Ende switch */
 return 0;
}
```

Listing 9.11 »switch_3.c« zeigt die Verwendung einer »default«-Option bei »switch«.

Jetzt haben Sie auch gesehen, wie es möglich ist, jedes beliebige darstellbare ASCII-Zeichen mit `case` zu überprüfen. Einzelne Zeichen werden – wie beim Programm mit den Operatoren eben verwendet – zwischen einzelne Hochkommata gestellt.

Natürlich wäre es auch denkbar, statt

```
case '+': printf("%d + %d = %d \n", a, b, a+b);
```

eben

```
case 43 : printf("%d + %d = %d \n", a, b, a+b);
```

zu verwenden, da das Zeichen `'+'` den dezimalen Wert 43 hat (siehe ASCII-Tabelle im Anhang). Der ASCII-Code ordnet den einzelnen Zeichen also Zahlen zu, und diese Zahlen werden anschließend anhand einer Zeichentabelle in ein druckbares Muster verwandelt und auf dem Bildschirm dargestellt. damit Sie könnten auch die Oktalzahl `053` oder die Hexzahl `2B` verwenden. Hier ist die Schreibweise für die Hexzahl von `'+'`:

```
case 0x2B: printf("%d + %d = %d \n", a, b, a+b);
```

Im Programm haben Sie gesehen, wie `default` verwendet wurde:

```
default: printf("'%c' ist kein Rechenoperator \n", opera);
```

Falls Sie keine gültigen Rechenoperatoren eingegeben haben, bekommen Sie eine entsprechende Meldung.

`default` bedeutet hier: *Falls eine passende* `case`*-Verzweigung fehlt, nimm immer das Folgende.* Wenn also keine `case`-Anweisung greift, dann wird der Codeblock nach `default` ausgeführt. (Man spricht oft vom sogenannten *Default-Verhalten*, wenn der `default`-Block ausgeführt wird.) Sie werden in Abschnitt 9.9 noch sehen, dass sich die `switch`-Verzweigung hervorragend für den Aufbau eines einfachen Menüs in einer Konsolenanwendung eignet.

9.8 Die »while«-Schleife

Mit Schleifen können Sie Anweisungsblöcke mehrfach hintereinander ausführen lassen. Eine von vielen möglichen Arten, dies zu realisieren, ist »while«, was so viel bedeutet wie »solange«. Das Grundgerüst der Schleifenbedingung `while` sieht folgendermaßen aus:

```
while(Bedingung == wahr) {
   /* Abarbeiten von Befehlen, bis Bedingung ungleich wahr */}
```

Ist die Bedingung, die in den Klammern nach `while` folgt, wahr, so wird der Anweisungsblock in den geschweiften Klammern ausgeführt. Die Anweisungen werden so oft ausgeführt, bis die Bedingung in den Klammern falsch, also unwahr ist. Den zugehörigen Programmablaufplan sehen Sie in Abbildung 9.9.

Üblicherweise läuft die `while`-Schleife in drei Schritten ab:

- *Initialisierung* – Die Schleifenvariable bekommt einen Wert.
- *Bedingung* – Die Schleifenvariable wird daraufhin überprüft, ob eine bestimmte Bedingung bereits erfüllt ist.
- *Reinitialisieren* – Die Schleifenvariable erhält einen anderen Wert.

In der Praxis sieht dies etwa so aus:

```
int var=0;           // Initialisieren (wichtig!)
while(var < 10) {    // Solange var kleiner als 10 - Bedingung
      // weitere Anweisungen
      var++;         // Reinitialisieren
   }
```

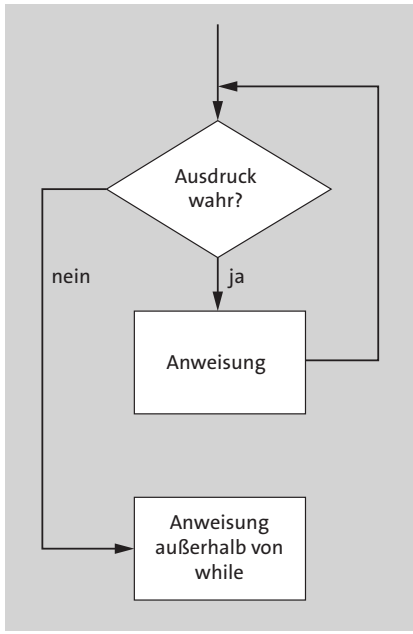

Abbildung 9.9 Programmablaufplan der »while«-Schleife

Jetzt folgt ein Listing mit einer `while`-Schleife:

```
/* while_1.c */
#include<stdio.h>

int main(void) {
   int zahl=1, tmp=0;

   while(zahl <= 10) {
      tmp=tmp+zahl;
      printf("%d+",zahl++);     /* Zahl + 1 */
   }
   printf("\b = %d\n",tmp);
   return 0;
}
```

Listing 9.12 »while_1.c« demonstriert eine Schleife mit »while«.

Das Programm führt die `while`-Schleife so lange aus, bis die Variable `zahl` kleiner oder gleich `10` ist. Hier ein kurzer Durchgang des Programms Schritt für Schritt:

```
while(zahl<=10)
```

Ist die Zahl kleiner oder gleich 10? Die Zahl hat am Anfang den Wert 1, also springt das Programm in den Anweisungsblock. Dann wird zuerst die Addition `tmp=tmp+zahl` durchgeführt. Somit ist der Wert von `tmp` 1. Jetzt wird die Variable `zahl` um 1 erhöht (`zahl++`). Der `while`-Anweisungsblock endet, und das Programm springt zurück zum Anfang der `while`-Anweisung. Es erfolgt eine erneute Überprüfung daraufhin, ob die Zahl immer noch kleiner oder gleich 10 ist. Da sie jetzt den Wert 2 hat, geht es wie vorher schon im Anweisungsblock weiter. Das wird so lange wiederholt, bis `zahl` den Wert 11 erreicht hat. In diesem Fall ist die Bedingung, ob `zahl` kleiner oder gleich 10 ist, falsch. Das Programm läuft nach dem `while`-Anweisungsblock weiter und gibt das Ergebnis `55` aus, also die Summe aller Zahlen von 1 bis 10. Was würde geschehen, wenn man am Anfang die Initialisierung vergisst? Die Antwort ist, dass in diesem Fall das Verhalten undefiniert ist. Deshalb sollten Sie nicht auf die Initialisierung der Variablen `zahl` verzichten.

9.8.1 Endlosschleifen mit »while«

Die `while`-Schleife wird manchmal absichtlich in Form einer Endlosschleife implementiert. Dies ist z. B. dann der Fall, wenn ein Programm sehr lange laufen soll (z. B. in Form einer Steuerung auf einem Mikrocontroller, die sich nur dann abschaltet, wenn ein ernster Fehler auftritt). Für das Beenden einer Endlosschleife sorgt dann nicht die Schleifenbedingung, sondern das Schlüsselwort `break`. Mit der `break`-Anweisung können Sie eine Schleife jederzeit verlassen, z. B. wenn ein Fehler auftritt. Hier sehen Sie ein Beispiel für eine Endlosschleife:

```
/* while_2.c */
#include<stdio.h>

int main(void) {
   int zahl, summe=0;
   printf("Summenberechnung\nBeenden der Eingabe mit 0 \n");

   while(1) {    /* Endlosschleife, denn: 1 ist immer wahr */
      printf("Bitte Wert eingeben > ");
      scanf("%d", &zahl);
      if(zahl == 0)    /*  Haben wir 0 eingegeben ...?   */
         break;        /* ... dann raus aus der Schleife */
      else
         summe+=zahl;
   }
   printf("Die Summe aller Werte beträgt: %d\n", summe);
   return 0;
}
```

Listing 9.13 »while_2.c« demonstriert eine Endlosschleife mit »while«.

Die Zeile

```
while(1)
```

verkörpert die Endlosschleife. Der Inhalt dieser Schleife ist immer wahr, da Sie hier den Wert `1` haben. Sie könnten dort jede beliebige Zahl außer `0` hinschreiben. Sie haben in den vorangegangenen Kapiteln gesehen, dass `1` (und alle anderen Werte ungleich 0) für »wahr« (*true*) und `0` für »unwahr« (*false*) steht.

Nun zurück zu dem Programm. Sie werden so lange aufgefordert, eine Zahl einzugeben, bis Sie den Wert »0« eingeben:

```
if(zahl == 0)    /*  Falls 0 eingegeben wurde ... */
   break;        /*  ... dann Schleife verlassen  */
```

Hat die Variable `zahl` den Wert `0`, wird mit der Anweisung `break` aus dem Anweisungsblock herausgesprungen.

9.8.2 Fehlervermeidung bei »while«-Schleifen

Natürlich können Sie mit Schleifen auch diverse Fehler machen. Im Folgenden sollen daher einige häufig auftretende Fehler angesprochen werden.

- **Schließen Sie eine Schleife niemals mit einem Semikolon ab.** Folgendes Beispiel erzeugt ungewollt eine Endlosschleife:

  ```
  int x=0;
  while(x > 10); /* Fehler durch Semikolon am Ende */
     {
        printf("Der Wert von x beträgt %d\n", x);
        x++;
     }
  ```

  Die Variable `x` wird niemals im Anweisungsblock inkrementiert werden, da die `while`-Schleife nichts anderes tut, als ständig die Bedingung zu prüfen, ob `x` größer als 10 ist, was niemals der Fall sein wird!

- **Vermeiden Sie** außerdem – wenn möglich – **Überprüfungen auf Gleichheit** in den Schleifen:

  ```
  int x=2;
  while(x == 10) {
     printf("Der Wert von x beträgt %d \n", x);
     x*=x;
  }
  ```

Die Bedingung der while-Schleife wird nie erfüllt. Verwenden Sie in einem solchen Fall besser die Überprüfung auf kleiner oder gleich, etwa wie folgt:

```
while(x <= 10)
```

- Eine weitere **Fehlerquelle** können die **logischen Operatoren** in der while-Schleife darstellen, wie das folgende Programmbeispiel verdeutlicht:

```
/* while_3.c */
#include<stdio.h>

int main(void) {
   int zahl1=0, zahl2=0;
   while((zahl1++ < 5) || (zahl2++ < 5) )
      printf("Wert von zahl1: %d zahl2: %d \n ", zahl1, zahl2);
   return 0;
}
```

Da der erste Ausdruck fünfmal wahr ist, wird wegen des logischen ODER-Operators der zweite Ausdruck nicht ausgewertet. Erst dann, wenn der erste Ausdruck unwahr, also 5 ist, wird der zweite Ausdruck fünfmal durchlaufen. Als Nebeneffekt wird aber gleichzeitig der erste Ausdruck bei jedem Schleifendurchlauf inkrementiert. Und so kommt es, dass die Variable zahl1 am Ende den Wert 10 hat, statt 5 wie ursprünglich beabsichtigt.

9.9 Die »do while«-Schleife

Die Schleife do while verhält sich wie die while-Schleife, nur dass die Bedingung am Ende des Anweisungsblocks überprüft wird. Hier die Syntax:

```
do {
      /* Anweisungen */
} while(BEDINGUNG == wahr);
```

Der Anweisungsblock wird mit dem Schlüsselwort do eingeleitet. Im Block werden dann die entsprechenden Anweisungen ausgeführt. Am Ende des Anweisungsblocks steht der bereits bekannte Ausdruck while, bei dem überprüft wird, ob die angegebene Bedingung wahr ist. Ist die Bedingung wahr, wird der Anweisungsblock erneut ausgeführt, und das Programm beginnt wieder bei do. Wenn die Bedingung unwahr ist, geht es hinter der while-Bedingung weiter.

Achten Sie auch darauf, dass Sie die do while-Schleife am Ende von while mit einem Semikolon abschließen. Das Semikolon zu vergessen, ist ein häufig gemachter Fehler. Abbildung 9.10 zeigt den Programmablaufplan zur do while-Schleife.

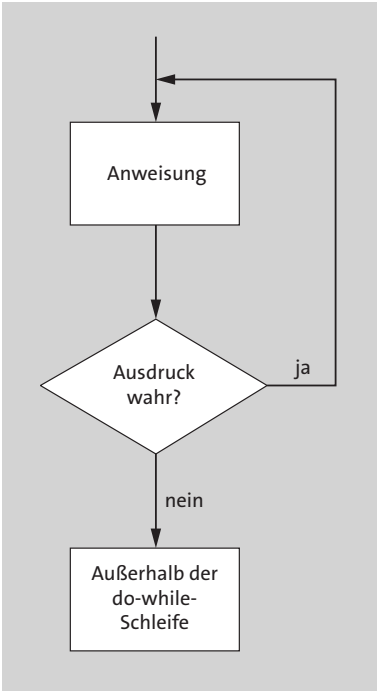

Abbildung 9.10 Programmablaufplan der »do while«-Schleife

Zur Veranschaulichung sehen Sie in Listing 9.14 ein Ratespiel:

```
/* do_while_1.c */
#include<stdio.h>
#include<stdlib.h>  /* für die Bibliotheksfunktion rand() */

int main(void) {
   int ratezahl, count=0, erraten=0;
   int zufall = (rand()%10)+1; /* Pseudo-Zufallszahl von 1-10 */

   do {
      printf("Zahleingabe bitte (1-10): ");
      scanf("%d", &ratezahl);
      if(ratezahl==zufall) {  /* Zahl richtig geraten ... ? */
         erraten=1;           /* Ja, die Zahl ist richtig   */
         count++;
      }
      else {
         (ratezahl > zufall) ?
            printf("kleiner\n") : printf("grösser\n");
         count++;
```

```
      }
   } while( (erraten != 1) && (count != 3) );

   if(erraten == 0) {
      printf("Sie haben 3 Chancen vertan ;) \n");
      printf("Die Zahl wäre %d gewesen: \n", zufall);
   }
   else
      printf("Mit %d Versuchen erraten!\n",count);
   return 0;
}
```

Listing 9.14 Ein kleines Zahlenratespiel soll Ihnen den Unterschied zwischen »while« und »do...while« veranschaulichen.

In diesem Programm wird die Headerdatei

```
#include<stdlib.h>
```

für die Funktion

```
(rand()%10)+1;
```

verwendet, womit eine *Pseudo-Zufallszahl* zwischen 1 und 10 erzeugt wird, deren Wert Sie an die Variable `zufall` übergeben. Aber was heißt »Pseudo-Zufallszahl«? Dies bedeutet, dass ein Computer keine echten Zufallszahlen erzeugen kann, sondern anhand einer mathematischen Formel nur Zahlen ausgeben kann, die so aussehen, als wären sie zufällig. Der Trick dabei ist, dass der Computer eine Zahlenreihe berechnet, bei der die folgenden Ziffern nicht vorhersehbar sind, wenn man den Anfangswert nicht kennt. Dieser Anfangswert (der sogenannte *seed*) ist aber leider immer 0, wenn das Programm startet, und dieser Wert ändert sich immer nur dann, wenn Sie `rand()` aufrufen. Probieren Sie es ruhig aus, und lassen sich mit `rand()` 100 Zahlen in einer Schleife ausgeben. Sie werden erstaunt darüber sein, dass immer dieselben Zahlen erscheinen und vielleicht auch darüber, dass in dem obigen Beispielprogramm die zu ratende Zahl immer dieselbe ist. Dies können Sie beheben, indem Sie am Anfang des Programms *<time.h>* einbinden und in der ersten Zeile des Hauptprogramms folgende Anweisung einfügen:

```
int main(void) {
   srand(clock()); // Nimmt die Systemuhr als seed für srand
```

Nach diesem kleinen Ausflug in die Welt der Zufallszahlen kehren wir zu unserem Programm zurück. Die `do while`-Schleife wird so lange wiederholt, bis die beiden Ausdrücke in `while` wahr sind:

```
while( (erraten != 1) && (count != 3) );
```

Die Bedingung für das Spiel lautet also: Sie haben maximal drei Versuche (`count != 3`), in denen Sie die Zahl erraten müssen. Solange Sie die Variable von `erraten` auf dem Wert `0` UND den Zähler für die Versuche `count` noch nicht auf `3` stehen haben, beginnt der Anweisungsblock wieder von Neuem. Ist eine dieser Bedingungen unwahr, haben Sie entweder die Zahl erraten oder mehr als drei Versuche benötigt. Dies wird anschließend nach dem Anweisungsblock ausgewertet.

Aber warum lassen Sie nicht den Computer eine Zahl raten, die Sie sich ausgedacht haben? »Künstliche Intelligenz« und »zu kompliziert«, denken Sie? Solch ein Spiel lässt sich einfacher realisieren, als Sie glauben. Hier ist das Spiel umgekehrt:

```
/* do_while_2.c */
#include <stdio.h>

int main(void) {
   char response;

   printf("Denk Dir eine Nummer zwischen 1 und 100 aus.\n");
   printf("Das errate ich in 7 oder weniger Versuchen \n\n");
   do {
      int lo = 1, hi = 100;
      int guess;
      while (lo <= hi) {
         guess = (lo + hi) / 2;
         printf("Ist es %d ",guess);
         printf(" ((h)oeher/(n)iedriger/(j)a): ");
         fflush(stdout);
         scanf("%c%*c",&response);

         if (response == 'h')
            lo = guess + 1;
         else if (response == 'n')
            hi = guess - 1;
         else if (response != 'j')
            printf("Erraten ... :-)");
         else
            break;
      }
      /* Resultat ausgeben */
      if (lo > hi)
         printf("Du schummelst!\n");
      else
```

```
        printf("Deine Nummer lautet: %d\n",guess);
      printf("Noch ein Spiel (j)a/nein : ");
      fflush(stdout);
      scanf("%c%*c",&response);
   } while( response == 'j' );
   return 0;
}
```

Listing 9.15 In »do_while_2.c« lassen Sie den Computer eine von Ihnen eingegebene Zahl raten.

Der Computer errät dabei Ihre Zahl immer in maximal sieben Versuchen. Verwendet wird dabei ein sogenanntes »Teile und herrsche«-Prinzip. Die Variable `guess` erhält durch die Berechnung `(lo+hi)/2` zuerst den Wert `50`. Ist die von Ihnen gesuchte Zahl jetzt höher, erhält die Variable `lo` einen neuen Wert mit `guess+1`, also `51`. Somit beginnt das Teilen und Herrschen von Neuem mit `(lo+hi)/2`, in Zahlen: `(51+100)/2`, also `75`. Ist jetzt die gesuchte Zahl niedriger, bekommt `hi` einen neuen Wert mit `guess-1`. Es ergibt sich beim nächsten Schleifendurchlauf `(lo+hi)/2`, in Zahlen: `(51+74)/2`, also `62`. Dies geht so weiter, bis die Zahl erraten wurde. Auf dieses »Teile und herrsche«-Prinzip werden Sie noch einige Male in diesem Buch stoßen.

Ein weiteres praktisches Anwendungsbeispiel der `do while`-Schleife in Kombination mit `switch` ist ein Benutzermenü für die Konsole:

```
/* do_while_3.c */
#include<stdio.h>

int main(void) {
   int auswahl;

   do {
      printf("-1- Auswahl1\n");
      printf("-2- Auswahl2\n");
      printf("-3- Auswahl3\n");
      printf("-4- Programmende \n\n");
      printf("\n\n Ihre Auswahl: ");
      scanf("%d", &auswahl);
      switch(auswahl) {
         case 1  :  printf("\n Das war Auswahl 1 \n"); break;
         case 2  :  printf("\n Das war Auswahl 2 \n"); break;
         case 3  :  printf("\n Das war Auswahl 3 \n"); break;
         case 4  :  printf("\n Programmende \n");      break;
         default :  printf("\n Unbekannte Auswahl \n");
      }
```

```
   } while(auswahl!=4);
   return 0;
}
```

Listing 9.16 »do_while_3.c« zeigt, wie Sie »switch ... case« und »do ... while« für die Realisierung eines Benutzermenüs auf der Konsole kombinieren können.

Meist stellt sich nach einem solchen Menübeispiel die Frage, wie der Bildschirm gelöscht werden kann. In ANSI C ist aber keine Funktion dafür vorgesehen. Das bedeutet, dass es von Compiler und Betriebssystem abhängt, ob eine (nicht standardisierte) Funktion hierfür existiert. Wenn Sie eine solche verwenden wollen, halten Sie Ausschau nach Funktionen wie `clrscr()` für Windows oder nach der Bibliothek *<ncurses.h>* für Linux. Aber auch auf der direkten Systemebene sind Ihre Programme, die den Bildschirm löschen, nicht portabel, weil die Systemfunktion für das Löschen der Konsole z. B. unter Windows `System("cls")` und unter Linux `system("clear")` heißt. Aber auch auf den Grafikspeicher können Sie unter Umständen nicht direkt zugreifen, zumindest nicht unter Windows.

9.10 Die »for«-Schleife

Mit der `for`-Schleife können Sie – wie schon mit der `while`-Schleife – Anweisungsblöcke mehrfach hintereinander ausführen. Der Unterschied ist jedoch, dass bei der `for`-Schleife die Initialisierung, die Bedingung und die Reinitialisierung der Schleifenvariable schon in der `for`-Anweisung erfolgen. Die Syntax zur `for`-Schleife ist:

```
for(Initialisierung; Bedingung; Reinitialisierung) {
   /* Anweisungen */
}
```

bzw. (in anderen Büchern auch auf die folgende Weise dargestellt):

```
for([Variable]=[Start];[Variable]<=[Ende]; [Variable]+=[Schrittweite]) {
   /* Anweisungen */
}
```

Beim Ablauf der `for`-Schleife wird in der Regel zuerst die *Schleifenvariable* (man spricht auch von einem *Schleifenzähler*) mit einem konkreten Wert initialisiert, oft ist dies einfach `0` oder `1`. Dies geschieht aber nur einmal, unabhängig davon, wie oft die Schleife wiederholt wird. Danach findet typischerweise eine Bedingungsüberprüfung statt, die oft einfach `[Variable]<=[Endwert]` ist, aber durchaus auch komplexer sein kann. Ergibt diese Bedingung »wahr«, geht die Ausführung im Anweisungsblock mit weiteren Anweisungen weiter.

Hat der Anweisungsblock alle Anweisungen ausgeführt, wird die *Reinitialisierung* der Schleifenvariable ausgeführt, die oft einfach die Startvariable um einen bestimmten Wert erhöht. Danach wird erneut die Schleifen-Bedingung überprüft, und alles beginnt von vorn. Die `for`-Schleife wird beendet, wenn die Schleifenbedingung nicht mehr wahr ist.

Die Arbeitsweise einer `for`-Schleife lässt sich gut anhand des folgenden Beispiels demonstrieren:

```
/* for_1.c */
#include<stdio.h>

int main(void) {
   int i;
   for(i=1; i <= 5; i++)
      printf("for(i=1; %d <= 5; %d++) \n", i, i);
   return 0;
}
```

Das Programm zeigt jeweils den aktuellen Variableninhalt von `i` in der `for`-Schleife. Wie Sie in diesem Beispiel gesehen haben, kann auch bei den Schleifen der Anweisungsblock mit den geschweiften Klammern weggelassen werden, wenn er nur aus einer Anweisung besteht. Der Programmablaufplan der `for`-Schleife ist in Abbildung 9.11 dargestellt.

Mit `for`-Schleifen lassen sich aber nicht nur Schleifen zum Dekrementieren (`--`) bzw. Inkrementieren (`++`) realisieren. Sie können in der `for`-Schleife auch weitere Rechenoperationen vornehmen. Im folgenden Listing sehen Sie ein Programm, das immer die letzte Dezimalstelle eines Werts abschneidet. Die Ergebniszahl soll danach spiegelverkehrt ausgegeben werden:

```
/* for_2.c */
#include<stdio.h>

int main(void) {
   int zahl;

   for(zahl=1234; zahl >= 1; zahl/=10)
      printf("%d", zahl%10);
   printf("\n");
   return 0;
}
```

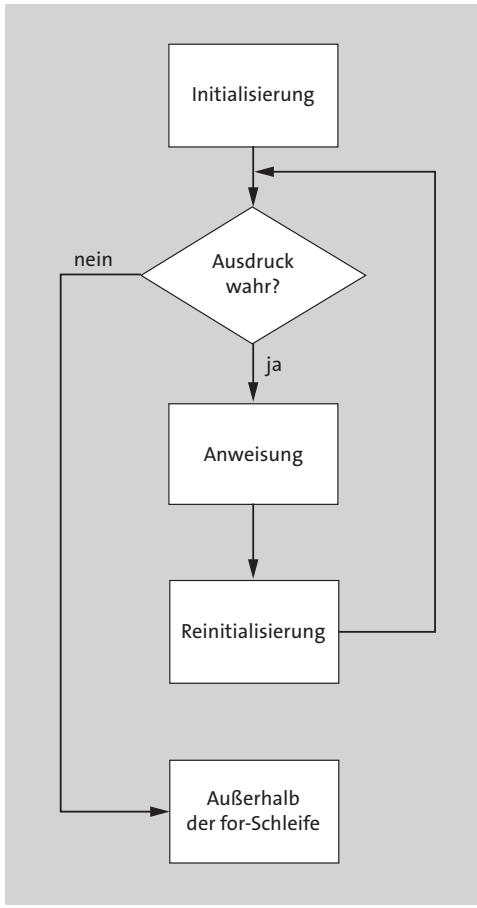

Abbildung 9.11 Programmablaufplan der »for«-Schleife

Typische Fehler bei der Verwendung einer »for«-Schleife

- Ein häufiger Fehler ist, das Semikolon zwischen der Initialisierung, der Bedingung und der Reinitialisierung der Schleifenvariable zu vergessen:

```
for(i=1 i<=5 i++)   /* korrekt-> */     for(i=1; i<=5; i++)
```

- Natürlich sollten Sie auch nicht den Fehler machen, am Ende der Schleife ein Semikolon zu setzen; außer es ist gewollt.

```
for(i=1; i<=5; i++);  /* korrekt-> */  for(i=1; i<=5; i++){  }
```

- Und das Wichtigste ist, für die richtige Abbruchbedingung zu sorgen, da sonst das Programm die `for`-Schleife nicht mehr verlässt. Man spricht dann von einer Endlosschleife.

```
for(i=2; i!=5; i+=2)   /* korrekt-> */   for(i=2; i<=5; i+=2)
```

9.10.1 Einsatzmöglichkeiten der »for«-Schleife

Als Anregung und Übung zeige ich Ihnen im Anschluss zehn verschiedene Möglichkeiten zur Verwendung einer `for`-Schleife.

1. Möglichkeit

```
/* for_3.c */
#include<stdio.h>
#include<time.h>

int main(void) {
   int sek;
   for(sek = 5; sek > 0; sek--)
   {
      printf("%d Sekunden!\n", sek);
      delay(1000); // Hier eine Sekunde warten
   }
   printf("Die Zeit ist abgelaufen! \n");
   return 0;
}
```

Listing 9.17 Erste Einsatzmöglichkeit von »for«: Warteschleifen

Hier benutzen Sie den Dekrement-Operator zum Rückwärtszählen. Die Schleife zählt so lange rückwärts, bis der Wert `sek` gleich `0` ist. Leider ist die Funktion `delay` nicht auf allen Systemen verfügbar. Fehlt sie, müssen Sie vor der Hauptfunktion folgende Funktion einfügen:

```
void delay(long int Millis) {
    long int Now=clock(); // Now=Zeit in Millisekunden beim Aufruf der
                          Schleife (für Windows)
    // Millis*=1000; Auf dem Raspberry Pi ist die Zeit in Mikrosekunden,
       hier bitte diese Zeile einfügen
    while ((clock()-Now)<Millis) { }
}
```

2. Möglichkeit

```
/* for_4.c */
#include <stdio.h>

int main(void) {
   int n;
   for(n = 0; n <= 60; n = n + 10)
```

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

Listing 9.18 Zweite Einsatzmöglichkeit von »for«: Zählschleife mit größeren Schritten als 1

Sie können innerhalb der Schleife nicht nur inkrementieren und dekrementieren, sondern auch rechnen. In diesem Fall werden die Zahlen in 10er-Schritten ausgegeben: von 0 bis 60.

3. Möglichkeit

```
/* for_5.c */
#include<stdio.h>

int main(void) {
   char ch;

   for(ch = 'A'; ch <= 'Z'; ch++)
      printf("%c, ", ch);
   printf("\n");
   return 0;
}
```

Listing 9.19 Dritte Einsatzmöglichkeit von »for«: Ausgabe von Zeichen, z. B. aller Zeichen von »A« bis »Z«

Diese Möglichkeit funktioniert auch sehr gut mit dem Datentyp `char`. Dann wird das Alphabet in Großbuchstaben ausgegeben.

4. Möglichkeit

```
/* for_6.c */
#include<stdio.h>

int main(void) {
   int cube;

   for(cube = 1; cube * cube * cube <= 216; cube++)
      printf("n=%d Volumen : %d \n", cube, cube*cube*cube);
   return 0;
}
```

Listing 9.20 Vierte Einsatzmöglichkeit von »for«: Mehrmaliges Ausführen einer Rechenvorschrift

Als Abbruchbedingung können Sie ebenso eine Berechnung verwenden. In diesem Fall soll das Volumen eines Würfels bis zur Größe von maximal 216 berechnet werden. Die Abbruchbedingung ist also erreicht, wenn `cube` bis 7 hochgezählt wurde, da 7 * 7 * 7 größer ist als 216.

5. Möglichkeit

```
/* for_7.c */
#include<stdio.h>

int main(void) {
   double zs;
   for(zs = 100.0; zs < 150.0; zs = zs * 1.1)
      printf("%.2f\n", zs);
   return 0;
}
```

Listing 9.21 Fünfte Einsatzmöglichkeit von »for«: Berechnung von Zins und Zinseszins

Bei dieser Möglichkeit werden immer 10 % vom jeweiligen Gesamtwert berechnet. Sie haben zum Beispiel 100 EUR auf der Bank und bekommen darauf 10 % Zinsen im Jahr. Sie wollen wissen, nach wie vielen Jahren der Betrag in die Nähe von 150 EUR kommt.

6. Möglichkeit

```
/* for_8.c */
#include<stdio.h>

int main(void) {
   int x,y=50;
   for(x = 0; y <= 75; y = (++x*5) + 50)
      printf("%d\n", y);
   return 0;
}
```

Listing 9.22 Sechste Einsatzmöglichkeit von »for«: Ausgabe von Werten von Reihen und Folgen

Diese etwas komplex anmutende Schleife tut nichts anderes, als einen Wert in 5er-Schritten von 50 bis 75 auszugeben. Zuerst wird in den Klammern (`x*5`) berechnet und anschließend `50` addiert. Da `x` sich bei jedem Durchlauf um den Wert 1 erhöht und mit `5` multipliziert wird, ergeben sich Werte in 5er-Sprüngen.

7. Möglichkeit

```
/* for_9.c */
#include<stdio.h>

int main(void) {
   int x = 2, y;

   for(y=2; x<20;) {
         x = x * y;
         printf("%d\n", x++);
   }
   return 0;
}
```

Listing 9.23 Siebte Einsatzmöglichkeit von »for«: Wiederholte Ausführung einer (komplexen) Berechnung abhängig von einer Bedingung

Sie sehen, dass nicht unbedingt alle Variablen einer `for`-Schleife deklariert werden müssen. Lediglich die beiden Semikolons müssen immer in der `for`-Schleife stehen.

8. Möglichkeit

```
/* for_10.c */
#include<stdio.h>

int main(void) {
   for(;;)
      printf("Endlosschleife!!\n");
   return 0;
}
```

Listing 9.24 Achte Einsatzmöglichkeit von »for«: Endlosschleifen

Listing 9.24 zeigt ein Beispiel, bei dem überhaupt nichts in der `for`-Schleife steht. Wenn Sie dieses Programm ausführen, wird so lange der String `Endlosschleife` auf dem Bildschirm ausgegeben, bis Sie das Programm selbst *gewaltsam* abbrechen. Die Schreibweise `for(;;)` ist gleichwertig mit `while(1)`. Beides sind Formen von Endlosschleifen!

9. Möglichkeit

```
/* for_11.c */
#include<stdio.h>
```

```
int main(void) {
   int n;

   for( printf("Bitte eine Zahl eingeben: "); n!=5; )
      scanf("%d", &n);
   printf("Diese Zahl wollte ich\n");
   return 0;
}
```

Listing 9.25 Neunte Einsatzmöglichkeit von »for«: Negativbeispiel für die Abfrage auf eine Zahl (hier 5 zum Beenden des Programms)

Im Beispiel benutzen Sie die `for`-Schleife zur Abfrage einer Zahl. Die Schleife wird beendet, wenn Sie die Zahl 5 eingeben. Leider ist hier die Verwendung von `for` ein sehr schlechter Programmierstil, und Sie sollten `while n!=5` einem `for` vorziehen. Außerdem wird der Text `Bitte eine Zahl eingeben` nur ein einziges Mal ausgegeben. Deshalb sollte man ja auch möglichst immer die Anweisungsblöcke in geschweifte Klammern schreiben. Ich habe dieses schlechte Beispiel trotzdem angeführt, weil es eben viele Programme gibt, die solchen fehlerhaften Code enthalten. Und Sie sollten auch fremden Code lesen (und gegebenenfalls korrigieren) können, weil Teamarbeit eben sehr verbreitet ist.

10. Möglichkeit

```
/* for_12.c */
#include<stdio.h>

int main(void) {
   int n1, n2;

   for(n1 = 1, n2 = 2; n1 <= 10; n1++)
      printf("%d\n",n1*n2);
   return 0;
}
```

Listing 9.26 Zehnte Einsatzmöglichkeit von »for«: Initialisieren von Werten

Es ist auch möglich, mehrere Werte innerhalb einer `for`-Schleife zu initialisieren. Auch dies sollten Sie möglichst vermeiden, weil es einerseits die Lesbarkeit Ihrer Programme stark mindert und andererseits wieder dazu führt, dass die Variablen am Anfang keinen eindeutigen Wert haben. Auch dieses Beispiel habe ich deshalb angeführt, weil es immer wieder vorkommt und Sie leicht in die Lage kommen können, Programme lesen zu müssen, die diese »Unart« enthalten.

Natürlich war dies nur eine Auswahl von Verwendungsmöglichkeiten einer `for`-Schleife. Der Abschnitt sollte jedoch zeigen, wie flexibel sich `for`-Schleifen verwenden lassen.

9.11 Kontrollierter Ausstieg aus Schleifen mit »break«

Es gibt vier Möglichkeiten, eine Ablaufstruktur (wie etwa Schleifen, Funktionen, Bedingungen oder gar das Programm) unmittelbar zu verlassen. Über sie kann aber nicht in eine bestimmte Anweisung verzweigt werden, sondern lediglich zur nächsten Ablaufstruktur. An dieser Stelle muss ich eine Warnung einfügen: Vermeiden Sie es möglichst, `break` und `continue` zu verwenden! Meist haben Sie einen ernsten Bug im Programm, wenn Sie eine Schleife frühzeitig verlassen müssen. Wozu behandeln wir aber dann solche »gefährlichen« Befehle? Die Antwort ist, dass `break` in manchen Situationen sehr nützlich ist, z. B. wenn Sie eventuelle Fehler abfangen möchten, die nicht immer vorhersehbar sind. Genau für diesen Fall eines unvorhersehbaren Fehlers gibt es folgende »Therapiemöglichkeiten«:

- `continue` – beendet bei Schleifen nur den aktuellen Schleifendurchlauf.
- `break` – beendet die Schleife oder eine Fallunterscheidung. Befindet sich `break` in mehreren geschachtelten Schleifen, wird nur die innerste Schleife verlassen.
- `exit` – beendet das komplette Programm.
- `return` – beendet die Iteration und die Funktion, in der `return` aufgerufen wird. Im Fall der `main()`-Funktion würde dies das Ende des Programms bedeuten.

Es sei angemerkt, dass `exit` und `return` keine schleifentypischen Anweisungen sind – im Gegensatz zu `break` und `continue`. Auf `continue` und `break` gehe ich jetzt noch etwas genauer ein.

9.11.1 continue

Die `continue`-Anweisung beendet nur die aktuelle Schleifenausführung. Das bedeutet, dass ab dem Aufruf von `continue` im Anweisungsblock der Schleife alle anderen Anweisungen übersprungen werden und die Programmausführung zur Schleife mit der nächsten Ausführung zurückspringt:

```
/* continue_1.c */
#include<stdio.h>

int main(void) {
   int i;
```

```
   for(i = 1; i <= 20; i++) {
     if(i % 2)        /* Rest bedeutet ungerade Zahl. */
        continue;     /* printf überspringen         */
     printf("%d ", i);
   }
   printf("\n");
   return 0;
}
```

Bei diesem Beispiel überprüfen Sie, ob es sich bei der Variablen `i` um eine ungerade Zahl handelt. In diesem Fall wird die `continue`-Anweisung ausgeführt. Die `printf`-Anweisung wird dabei übersprungen, und es geht zurück zum nächsten Schleifendurchlauf. Die Ausgabe des Programms bestätigt Ihnen, dass nur die geraden Zahlen »durchkommen«. So konnten Sie sich mit `continue` praktisch eine `else`-Anweisung sparen.

Bitte achten Sie bei `continue` auf die folgenden eventuell ungünstigen Seiteneffekte:

```
/* continue_2.c */
#include<stdio.h>

int main(void) {
   int i=2;
   while(i <= 10) {
      if(i % 2)        /* Rest bedeutet ungerade Zahl. */
         continue;     /* printf überspringen         */
      printf("%d ", i);
      i++;
   }
   return 0;
}
```

Dieses Programm hätte dasselbe ausführen sollen wie schon das Programmbeispiel zuvor – mit dem Unterschied, dass es sich hier um eine Endlosschleife handelt. Denn sobald

```
if(i % 2)
```

wahr ist, also der Wert ungerade ist, springt das Programm wieder zum Schleifenanfang. Die Iteration (`i++`) wird danach nie mehr ausgeführt, da sich der Wert der Variablen `i` nicht mehr ändern kann und somit immer ungerade bleibt. Das Programm stürzt ab und kann nur noch durch `kill` beendet werden.

9.11.2 break

Bisher haben Sie die `break`-Anweisung nur bei der `switch ... case`-Verzweigung verwendet. Mit `break` können aber auch Schleifen vorzeitig beendet werden. Allerdings sollte dies nur ausnahmsweise erfolgen. Denn im Regelfall sollten Sie eine Schleife immer mit einer Bedingung beenden.

Achtung: »break« in verschachtelten Schleifen

Häufig wird der Fehler gemacht, `break` in einer verschachtelten Schleife zu setzen. Wenn `break` in der innersten Schleife verwendet wird, wird auch nur diese Schleife abgebrochen.

9.12 Direkte Sprünge mit »goto«

Ich habe mir gut überlegt, ob ich `goto` überhaupt noch erwähnen soll, und diesen Befehl nun endgültig gestrichen, da der nächste C-Standard `goto` wahrscheinlich entfernen wird. Die Anweisung ist ein Relikt aus alten Zeiten und funktioniert wie das alte `GOTO` in BASIC: Es springt direkt zu einer Zeile, die Sie zuvor mit einem Label eindeutig definiert haben. Da aber das Portieren alter »Retro«-Programme auf einen modernen C-Compiler quasi einer Neuentwicklung gleichkommt, ist selbst dieser Anwendungsfall kein Grund dafür, `goto` zu verwenden.

9.13 Einige Anmerkungen zum Notationsstil, ehe Sie weiterlesen

Ich möchte noch ein paar Sätze zur Notationsform von strukturierten Anweisungen sagen. Als Beispiel soll die `for`-Schleife verwendet werden. Folgende Stile finden Sie häufiger vor:

K&R-Stil

```
for(i=0; i < 10; i++){
        /* Anweisungen */
   }
```

Whitesmith-Stil

```
for(i=0; i < 10; i++)
           {
             /* Anweisungen */
           }
```

Allman-Stil

```
for(i=0; i < 10; i++)
{
          /* Anweisungen */
}
```

GNU EMACS-Stil

```
for(i=0; i < 10; i++)
          {
                    /* Anweisungen */
          }
```

Der Stil des Autors (K&R-like)

```
for(i=0; i < 10; i++) {
   /* Anweisungen */
}
```

Welchen Notationsstil Sie letztendlich verwenden, ist wohl eine Frage des eigenen Geschmacks. Arbeiten Sie allerdings an einem größeren Projekt mit mehreren Personen, so sollte der Stil vorher abgesprochen werden, um eine Vermischung zu vermeiden. Auch erstellt die Projektleitung (dies kann sogar ein eigenes Team sein) gewöhnlich einen Style Guide, und wenn Sie selbst das Projekt leiten, sollten Sie auf jeden Fall einen solchen erstellen.

Tipp für sauberes Programmieren

Verwenden Sie möglichst immer nur eine Anweisung pro Zeile. Sollte Ihr Code einen Fehler enthalten, ist es bei dieser Schreibweise viel einfacher, den Fehler zu finden. Es gibt natürlich Ausnahmen, z. B. wenn Sie viele `if`s programmiert haben, die Sie untereinander in jeweils eine Zeile schreiben möchten und die Sie nicht in einem `switch ... case` unterbekommen (aus welchen Gründen auch immer). Aber wie so oft im Leben gibt es von jeder Regel Ausnahmen, und ein zu starres Festhalten an Regeln behindert oft mehr, als dass es nützt.

9.14 Einige Anmerkungen zu einem guten Programmierstil

Auch auf guten Programmierstil möchte ich noch eingehen. Natürlich gibt es nicht »den« guten Programmierstil, aber wenn Sie einige Grundlagen beachten, dann können auch andere, z. B. Teammitglieder, Ihre Programme gut lesen. Lassen Sie also Ihre Kollegen nicht verzweifeln und beherzigen die folgenden Ratschläge:

Vor allem sollten Sie »sprechende« Namen verwenden. Schreiben Sie also lieber `PunkteZaehler` oder `Timer1`, `Timer2`, `Timer3` … statt `a`, `b`, `c`, `d` …. Vielleicht könnten Sie sogar die einzelnen Zeitgeber präzisieren und z. B. `TimeGameRunning` oder `OutOfFuelTimer` schreiben.

Ferner sollten Sie Ihre Zeilen nicht unnötig »vollpacken«. Vermeiden Sie deshalb möglichst, mehr als 4 bis 5 Anweisungen ein eine Zeile zu quetschen (es sei denn, es geht nicht anders).

Ich empfehle Ihnen auch, Schleifen immer mit Klammern zu versehen, auch wenn sie nur in einer einzigen Zeile untergebracht sind und man deswegen keine Klammern benötigt. Auch dies erhöht die Lesbarkeit. Schreiben Sie also z. B.

```
while (Str[Pos]!=0) { Pos++; }
```

anstatt:

```
while (Str[Pos]!=0) Pos++;
```

Beide Anweisungen sind richtig, aber ich empfehle Ihnen die erste Variante, weil dort viel klarer zum Ausdruck kommt, wo die Schleife anfängt und endet. Außerdem »meckert« der Compiler bei der ersten Version, wenn Sie die Schleife nicht wieder korrekt durch eine Klammer schließen. Sie vermeiden dadurch sehr viele Fehler.

Im nächsten Kapitel erfahren Sie, was Funktionen sind. Auch mit dieser Programmiertechnik können Sie Ihr Programm sehr übersichtlich strukturieren.

Kapitel 15
Dynamische Speicherverwaltung

Bisher haben Sie Daten wie Arrays in einem statischen Speichersegment im Programm abgelegt. Wenn sich allerdings die Anzahl der Elemente zur Laufzeit verändert oder ein Speicher nicht dauerhaft benötigt wird, ist ein solcher statischer Speicher nicht geeignet. In diesem Kapitel erfahren Sie daher, wie Sie Speicherblöcke mit variabler Größe zur Laufzeit des Programms verwalten können.

Bisher wurde die Speicherverwaltung in Variablendefinitionen versteckt. Sie mussten sich so zwar keine Gedanken über die Verwaltung machen, aber spätestens dann, wenn Sie neuen Speicher für weitere Daten benötigten, musste der Code umgeschrieben werden. Darüber hinaus kann es auch Probleme mit der Gültigkeit einer Variablen geben. Eine Variable existiert nur in dem Anweisungsblock, in dem sie deklariert wurde. Die Gültigkeit des Speicherbereichs dieser Variablen verfällt, sobald dieser Anweisungsblock verlassen wird. Ausnahmen stellen globale und als `static` deklarierte Variablen dar. Aber zu viele globale Variablen sind im Allgemeinen kein guter Programmierstil, weil Sie dann die Übersicht über die zahlreichen Variablennamen verlieren, die Sie ja bei globalen Variablen nur einmal im Programm verwenden dürfen.

Bei der *dynamischen Speicherverwaltung* gibt es diese Probleme nicht mehr, denn »dynamisch« bedeutet, dass Sie zur Laufzeit Speicher anfordern oder freigeben können. Zur dynamischen Speicherverwaltung wird ein Zeiger an die Funktion `malloc()` übergeben, der später die Anfangsadresse der dynamischen Datenstruktur enthalten soll. Mit `malloc()` geben Sie außerdem an, wie viel Speicherplatz reserviert werden soll. Der Zeiger verweist also bei erfolgreicher Reservierung auf die Anfangsadresse des reservierten Speicherblocks. Die Aufgabe des Programmierers ist es, dafür zu sorgen, dass es immer einen Zeiger gibt, der auf diese Anfangsadresse verweist. Der so reservierte Speicher bleibt so lange erhalten, bis er entweder explizit mit der Funktion `free()` freigegeben wird oder bis das Programm sich beendet.

Sie müssen natürlich einen gewissen Aufwand betreiben, um die dynamische Speicherverwaltung zu realisieren. Wenn Sie dabei unvorsichtig vorgehen, haben Sie schnell eine sogenannte *Zugriffsverletzung* verursacht, unter Linux auch als *Segmentation Fault* bezeichnet. In diesem Fall wird das Programm mit einer entsprechenden System-Meldung beendet, die besagt, dass auf einen ungültigen Speicherbereich

zugegriffen wurde. Zu Problemen kann es auch kommen, wenn Sie bei einem Programm immer wieder Speicher reservieren und dieser niemals mehr freigegeben wird. Man spricht dabei von *Speicherlecks* (engl. *Memory Leaks*). Wie diese und andere Probleme vermieden werden, erfahren Sie in den folgenden Abschnitten.

15.1 Das Speicherkonzept

Bevor gezeigt wird, wie Speicher dynamisch reserviert werden kann, folgt ein Exkurs über das Speicherkonzept von laufenden Programmen. Wenn Ihnen dieser sehr hardwarelastige Exkurs mit verschiedenen Spezialbegriffen aus der Prozessorarchitektur zurzeit zu schwierig ist, können Sie diesen Abschnitt getrost überspringen. Wenn Sie jedoch schon fortgeschrittener sind und sozusagen nur auf C »umsatteln«, dann bietet Ihnen dieser Abschnitt einige nützliche Informationen.

Ein Programm besteht aus den vier Speicherbereichen, die in Tabelle 15.1 aufgeführt sind. Dies sind rein abstrakte Begriffe – das wird oft vergessen! Jeder Prozessor verwaltet den Speicher anders, aber auf irgendeinen Standard bei den Begriffen musste man sich ja schließlich einigen. Auf Intel-Maschinen gibt es z. B. wirklich Segmente für Code, Daten, Stack und Heap, weil es hierfür extra Register gibt. Auf sogenannten RISC-Maschinen dagegen (der Raspberry Pi ist z. B. eine solche) wird der Speicher oft linear adressiert und der entsprechende C-Compiler und das Betriebssystem müssen diese Dinge entsprechend dem Standard abbilden.

Speicherbereich	Verwendung
Code	Maschinencode des Programms
Daten	statische und globale Variablen
Stack	Funktionsaufrufe und lokale Variablen
Heap	dynamisch reservierter Speicher

Tabelle 15.1 Verschiedene Speicherbereiche in einem Programm

15.1.1 Codespeicher

Hier steht der auszuführende *Programmcode*, also die sogenannten *OP-Codes*, die der Prozessor direkt ausführen kann. Um den Prozessor an die gewünschte Stelle – also z. B. zu `main()` – springen zu lassen, gibt es spezielle Prozessorbefehle (wie z. B. `JMP`), und der *Programmlader* des Betriebssystems ist dafür zuständig, die Sprungbefehle korrekt auszuführen. Der Prozessor besitzt außerdem ein spezielles Register, das auf die Adresse des Befehls zeigt, der als nächster ausgeführt werden soll. Dieses Register wird als *Programm-Counter* (PC) bzw. *Instruction Pointer* (IP) bezeichnet. Das

heißt, ein versehentliches Überschreiben des Codesegments führt zu einem unvorhersehbaren Verhalten. Natürlich gibt es auch Exploits (das sind spezielle Hacker-Angriffe), die solche Dinge bewusst erzeugen und z. B. an der Stelle des gerade laufenden Programms ein Virus einschleusen.

15.1.2 Daten-Speicher

Im *Datensegment* befinden sich alle statischen Daten, die bis zum Programmende verfügbar sind (globale und statische Variablen). Auch hier ist die Organisation des Speichers vom Prozessor abhängig. Zwei oft verwendete Verfahren sind: Entweder es wird wirklich ein zusätzlicher Zeiger verwendet, der auf ein Extrasegment zeigt (Intel), oder aber das Datensegment wird beim Einstieg in das Programm durch JMP übersprungen, und das Datensegment befindet sich stets am Anfang des Programms (Motorola, ARM). Im zweiten Fall wird das Datensegment manchmal *indiziert adressiert*, das heißt durch eine Basisadresse und einen *Offset* (Versatz).

15.1.3 Stack-Speicher

Im *Stacksegment* werden die Funktionsaufrufe mit ihren lokalen Variablen verwaltet und auch Rücksprungadressen gespeichert. In Abschnitt 10.20.1, »Exkurs: Stack«, bin ich schon näher auf den Stack eingegangen, aber einiges bleibt noch zu erläutern. Der Prozessor besitzt nämlich (wie in Kapitel 10, »Funktionen«, in Bezug auf den Stack schon kurz erwähnt) ein spezielles Register, den *Stack Pointer* (SP). Wie viele Stack Pointer und damit Stacks es gibt, ist vom Prozessor abhängig; normalerweise gibt es genau einen. Natürlich wird in diesem Fall auf dem Stack alles abgelegt, auch Rücksprungadressen zu aufrufenden Funktionen. Ferner gibt es Prozessorbefehle zum Zwischenspeichern von Registerinhalten (man kann also Zugriffe auf den Stack explizit auslösen). Was passiert also, wenn der Stack aus irgendeinem Grund durcheinanderkommt? Der Prozessor findet beim Verlassen einer Funktion nicht mehr zurück, er hat quasi die Orientierung verloren. Nun kann alles passieren, nur nicht das, was man erwartet!

15.1.4 Heap-Speicher

Dem *Heap-Speicher* gebührt in diesem Kapitel das Hauptinteresse. Über ihn wird nämlich die dynamische Speicherreservierung mit Funktionen wie `malloc()` realisiert. Der Heap funktioniert ähnlich wie der Stack. Bei einer Speicheranforderung erhöht sich der Heap-Speicher, und bei einer Freigabe wird er wieder verringert. Auch hier ist die Speicherorganisation vom Prozessor abhängig. Der Heap-Speicher kann also direkt nach dem Datensegment liegen (in diesem Fall kann auch manchmal die Größe des maximalen Heap-Speichers als Parameter an den Compiler übergeben

werden) oder auch direkt nach dem Codesegment folgen. In den meisten Fällen wird aber der Heap-Speicher vom Betriebssystem verwaltet. Wenn ein Speicherblock angefordert wurde, sieht das Betriebssystem nach, ob sich im Heap noch genügend zusammenhängender freier Speicher dieser Größe befindet. Bei Erfolg wird die Anfangsadresse des passenden Speicherblocks zurückgegeben. Für den Heap ist allerdings (im Gegensatz zum Stack) eben sehr oft das Betriebssystem verantwortlich, weil es für den Heap normalerweise kein gesondertes Prozessorregister gibt. Darum ist der Zugriff auf den Heap langsamer als der Zugriff auf den Stack. Ausnahmen gibt es natürlich, z. B. für sehr alte Systeme oder für Mikrocontroller (hier verwaltet oft das Programm selbst den Speicher).

15.2 Speicherallokation mit »malloc()«

Ich habe bereits kurz erwähnt, mit welcher Funktion Speicher dynamisch reserviert werden kann. Es wird dabei auch von einer *Speicherallokation* (*allocate*, dt. *zuweisen*) gesprochen. Die Syntax dieser Funktion sieht so aus:

```
#include <stdlib.h> // Diese Bibliothek benötigt man für malloc()
void *malloc(size_t size); // Dies sind Byte!
```

Bei erfolgreichem Aufruf liefert die Funktion `malloc()` die Anfangsadresse mit der Größe `size` Byte vom Heap zurück. Da die Funktion einen `void`-Zeiger zurückliefert, hängt sie nicht von einem Datentyp ab. Hierzu ein Beispiel:

```
/* malloc_1.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int *p;
   p = malloc(sizeof(int));
   if(p != NULL) {
      *p=99;
      printf("Allokation erfolgreich ... \n");
   }
   else {
      printf("Kein virtueller RAM mehr verfügbar ...\n");
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}
```

Nach der Deklaration eines `int`-Zeigers wurde diesem mit

```
p = malloc(sizeof(int));
```

eine Anfangsadresse eines Speicherbereichs der Größe `int` zugewiesen. Bei Erfolg zeigt der Zeiger `p` auf den Anfang des reservierten Speicherbereichs. Ist dabei etwas schiefgegangen, zeigt der Zeiger auf `NULL`, und es wird ausgegeben, dass kein Speicherplatz reserviert werden konnte. Es ist in diesem Fall dann auch sinnvoll, einen Zeiger vor Verwendung im Programm vorher stets auf `NULL` abzufragen. Abbildung 15.1 verdeutlicht den Programmablauf anhand einer Grafik.

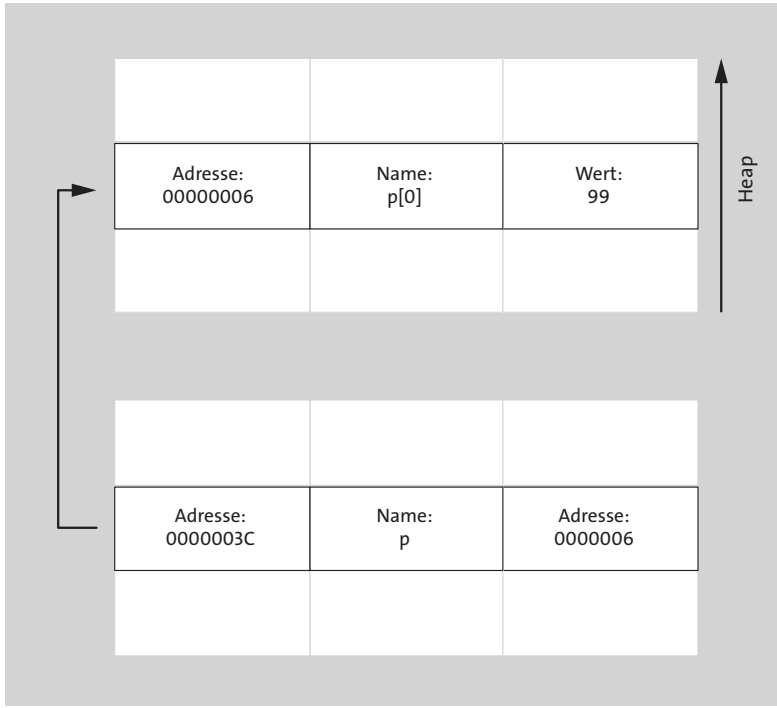

Abbildung 15.1 Dynamisch reservierter Speicher vom Heap

Hinweis: Typecasting bei »malloc()«

Ein Typecasting der Funktion `malloc()` ist in C nicht notwendig. ANSI C++ schreibt allerdings ein Casten des Typs `void *` vor (im Beispiel wäre dies C++-konform: `p=(int *) malloc(sizeof(int));`). Falls Sie also eine Fehlermeldung wie `'void *' kann nicht in 'int *' konvertiert werden` erhalten, dann haben Sie einen C++-Compiler vor sich bzw. einen Compiler, der im C++-Modus läuft. Häufig ist es aber problemlos möglich, den Compiler im C-Modus zu betreiben. Bei Visual C++ (beispielsweise die Version 2022) z. B. brauchen Sie nur die Eigenschaftsseite mit Alt+F7 aufzurufen und über Konfigurationseigenschaften • C/C++ • Erweitert die Option Kompilie-

RUNGSART auf ALS C-CODE KOMPILIEREN einstellen. Bei anderen Compilern ist dies häufig einfacher, weil man gleich ein reines C-Projekt erstellen kann. Beim GCC gibt es für C++ einen extra Compiler, den man z. B. wie folgt aufrufen kann:

```
C++  malloc_1.cpp -o malloc_1
```

»Na gut«, werden Sie vielleicht denken, »das hätte man ja auch mit einer globalen Variablen machen können.« Zur Laufzeit eines Programms Speicherplatz für einen `int`-Wert reservieren mit solch einem Aufwand? Lohnt sich wirklich die Verwendung einer separaten Funktion? Gut, dann reservieren Sie eben mehr Speicherplatz für mehrere `int`-Werte:

```
p = malloc(2 * sizeof(int));
```

Hiermit reservieren Sie Speicherplatz für zwei `int`-Werte vom Heap. Nachfolgend sehen Sie das Beispiel als Listing:

```
/* malloc_2.c */
#include<stdio.h>
#include<stdlib.h>

int main(void) {
   int *p = malloc(2 * sizeof(int));

   if(p != NULL) {
      *p=99;         /* alternativ auch p[0] = 99  */
      *(p+1) = 100;  /* alternativ auch p[1] = 100 */
      printf("Allokation erfolgreich ... \n");
   }
   else {
      printf("Kein virtueller RAM mehr verfügbar ...\n");
      return EXIT_FAILURE;
   }
   printf("%d %d\n", p[0], p[1]);
   /* Sie können die Werte auch so ausgeben lassen. */
   printf("%d %d\n", *p, *(p+1));
   return EXIT_SUCCESS;
}
```

Abbildung 15.2 soll den Sachverhalt veranschaulichen.

Der Sachverhalt, warum `*p` und `p[0]` oder `*(p+1)` und `p[1]` auf dasselbe Element zugreifen, wurde in Kapitel 13, »Zeiger (Pointer)«, geklärt. Blättern Sie notfalls einfach zu den Tabellen am Ende von Kapitel 13 zurück. Sie sehen vielleicht an dieser Stelle,

dass in Tabelle 13.2 der Heap nach oben wächst, im Gegensatz zum Stack, der nach unten wächst. Das liegt daran, dass auf den meisten Prozessoren der Stack Pointer dekrementiert (also vermindert) wird, nachdem ein Wert auf dem Stack abgelegt wurde. Auf dieses sogenannte *Post-Dekrement*-Verfahren hat das Betriebssystem keinen Einfluss, wohl aber auf die Verwaltung des Heaps. Was ist aber jetzt der Vorteil an einem solchen Speicherverwaltungsverfahren? Der Vorteil ist, dass Sie stets Speicher zur Laufzeit nachreservieren können, ohne dass sich Stack und Heap jemals in die Quere kommen.

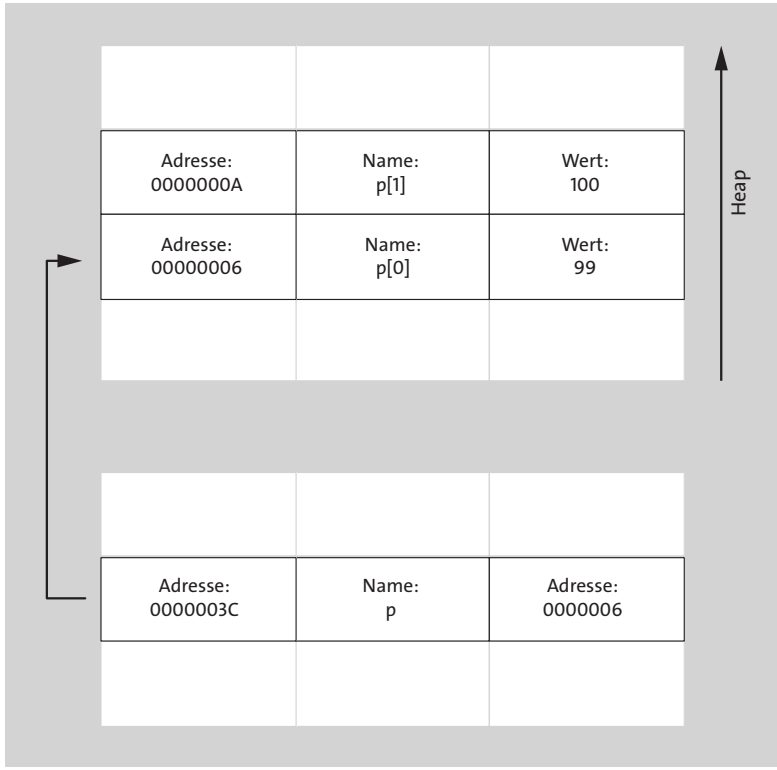

Abbildung 15.2 Speicher für mehrere Elemente dynamisch anfordern

15.3 Das NULL-Mysterium

Ein `NULL`-Zeiger wird zurückgeliefert, wenn `malloc()` nicht mehr genügend zusammenhängenden Speicher finden kann. Der `NULL`-Zeiger ist ein vordefinierter Zeiger, dessen Wert sich von einem regulären Zeiger unterscheidet. Er wird vorwiegend bei Funktionen zur Anzeige und Überprüfung von Fehlern genutzt, die einen Zeiger als Rückgabewert zurückgeben.

15.3.1 NULL für Fortgeschrittene

Sicherlich haben Sie sich schon einmal gefragt, was es mit NULL auf sich hat. Wenn Sie dann in einem Forum nachgefragt haben, könnten Sie beispielsweise dreierlei Antworten zurückbekommen haben:

- NULL ist ein Zeiger auf (die Adresse) 0.
- NULL ist ein typenloser Zeiger auf 0.
- Es gibt keinen NULL-Zeiger, sondern nur NULL, und das ist eben 0.

Sicherlich gibt es noch einige Antworten mehr hierzu. Aber es ist doch ziemlich verwirrend, ob NULL eben 0 ist oder ein Zeiger auf 0 – und wo ist dabei eigentlich der Unterschied?

Ein integraler konstanter Ausdruck mit dem Wert 0 wird zu einem NULL-Zeiger, wenn dieser einem Zeiger zugewiesen oder auf Gleichheit mit einem Zeiger geprüft wird. Damit ergeben sich die folgenden möglichen define-Anweisungen für NULL:

```
#define NULL 0
#define NULL 0L
#define NULL (void *) 0
```

Am häufigsten sieht man das Makro NULL als (void *) 0 implementiert, was den Vorteil hat, dass einem gegebenenfalls bei der Übersetzung die Arbeit abgenommen wird. Allerdings sollten Sie – egal wie NULL nun implementiert ist – auf eine Typumwandlung von NULL verzichten, denn schließlich kann NULL ja auch nur 0 oder 0L sein.

Der Compiler muss zur Übersetzungszeit selbst feststellen, wann ein NULL-Zeiger benötigt wird, und eben den entsprechenden Typ dafür eintragen. Somit ist es ohne Weiteres möglich, folgende Vergleiche zu verwenden (beide Versionen erfüllen denselben Zweck):

```
/* null_ptr_1.c */
#include<stdio.h>
#include<stdlib.h>

int main(void) {
   char *ptr1 = NULL;
   char *ptr2 = 0;

   if(ptr1 == NULL){
      printf("ptr1 ist ein Nullzeiger.\n");
   }
   if(ptr2 == 0) {
      printf("ptr2 ist ein Nullzeiger.\n");
```

```
    }
    return EXIT_SUCCESS;
}
```

Listing 15.1 »null_ptr_1.c« demonstriert die Verwendung des NULL-Zeigers.

Bei einem Funktionsargument allerdings ist ein solcher Zeiger-Kontext nicht unbedingt feststellbar. Ein Compiler kann eventuell nicht feststellen, ob der Programmierer 0 (als Zahl) oder den `NULL`-Zeiger meint. Das folgende Beispiel zeigt, worauf ich hinauswill:

```
execl ("/bin/sh", "sh", "-c", "ls", 0);
```

Der (UNIX-)Systemaufruf `execl()` erwartet eine Liste variabler Länge mit Zeigern auf `char`, die mit einem `NULL`-Zeiger abgeschlossen werden. Der Compiler kann hier allerdings wiederum nicht feststellen, ob der Programmierer mit `0` den `NULL`-Zeiger meint; daher wird sich der Compiler in diesem Fall für die Zahl 0 entscheiden. In diesem Fall müssen Sie unbedingt eine Typumwandlung nach `char*` durchführen – oder eben den `NULL`-Zeiger verwenden:

```
execl ("/bin/sh", "sh", "-c", "ls", (char *)0);
// ... oder ...
execl ("/bin/sh", "sh", "-c", "ls", NULL);
```

Dieser Fall ist besonders bei einer variablen Argumentliste (wie sie hier mit `execl()` vorliegt) wichtig. Denn hier funktioniert alles (auch ohne ein Type-Casting) bis zum Ende der explizit festgelegten Parameter. Alles, was danach folgt, wird nach den Regeln für die Typenerweiterung behandelt, und es wird eine ausdrückliche Typumwandlung erforderlich. Da es sowieso kein Fehler ist, den `NULL`-Zeiger als Funktionsargument einer expliziten Typumwandlung zu unterziehen, ist man bei regelmäßiger Verwendung (eines expliziten Casts) immer auf der sicheren Seite, wenn dann mal Funktionen mit einer variablen Argument-Anzahl verwendet werden.

Bei einem »normalen« Funktionsprototyp hingegen werden die Argumente weiterhin anhand ihrer zugehörigen Parameter im Prototyp umgewandelt. Verwenden Sie hingegen keinen Funktionsprototyp im selben Gültigkeitsbereich, wird auch hier eine explizite Umwandlung benötigt. Das ist jetzt vielleicht etwas verwirrend, aber im weiteren Verlauf wird vieles, was jetzt unklar erscheint, noch geklärt. Sie haben ja die Hälfte noch vor sich.

15.3.2 Was jetzt – NULL, 0 oder \0 ...?

Ob Sie jetzt `NULL` oder `0` verwenden, ist eine Frage des Stils und des Programmierers. Viele Programmierer halten `NULL` für eine sinnlose Neuerung, auf die man gern ver-

zichten kann (wozu soll man eine 0 hinter NULL verstecken). Andere wiederum entgegnen, dass man mit NULL besser unterscheiden kann, ob denn nun ein Zeiger gemeint ist oder nicht. Denn Folgendes wird der Compiler bemängeln, wenn NULL als (void *)0 implementiert ist:

```
/* null_ptr_2.c */
#include<stdio.h>
#include<stdlib.h>

int main(void) {
   int *ptr = NULL;
   int i = NULL; // falsch, hier erscheint eine Fehlermeldung
   return EXIT_SUCCESS;
}
```

Intern nach dem Präprozessorlauf würde aus der Zeile

```
int i = NULL;
```

Folgendes gemacht:

```
int i = (void *) 0;
```

Dies ist ein klarer Regelverstoß und somit unzulässig. Leider gilt diese Aussage nicht für C++, denn hier ist NULL schlicht 0 (weil es mit #define NULL 0L definiert wird), und im oberen Beispiel würde auch der Variablen i der Wert 0 zugewiesen.

Zusätzlich bringen Anfänger auch noch das Terminierungszeichen \0 (NUL) und NULL durcheinander. Es wird also oft das ASCII-Zeichen NUL mit NULL verwechselt. Allerdings kann man \0 nicht mit NULL vergleichen. \0 garantiert beispielsweise laut Standard, dass alle Bits auf 0 gesetzt sind; NULL tut das nicht.

15.3.3 Zusammengefasst

Aufgrund der häufigen Nachfragen zu NULL habe ich mich entschieden, das Thema etwas breiter aufzurollen. Vielleicht ist der eine oder andere jetzt verwirrt. Daher folgen hier zwei Regeln zum Umgang mit NULL, die Sie einhalten sollten, damit Sie auf der sicheren Seite sind:

- Wollen Sie im Quelltext einen NULL-Zeiger verwenden, dann können Sie entweder eben diesen Null-Zeiger mit der Konstante 0 oder eben das Makro NULL verwenden.
- Verwenden Sie hingegen 0 oder NULL als Argument eines Funktionsaufrufs, wenden Sie am besten die von der Funktion erwartete explizite Typumwandlung an.

15.4 Speicherreservierung und ihre Probleme

Bei einem Aufruf der Funktion `malloc()` muss die Größe des zu reservierenden Speichers in Byte angegeben werden. Damit ist die Größe des Speicherobjekts gemeint, das durch einen Zeiger referenziert werden soll. Für die dynamische Speicherzuweisung haben Sie folgende drei Möglichkeiten:

- Angabe des Datentyps mithilfe des `sizeof`-Operators:

  ```
  p = malloc(sizeof(int));
  ```

  Diese Möglichkeit hat einen Nachteil. Was ist, wenn Sie statt `int`-Werten auf einmal `double`-Werte benötigen? Dann müssen Sie mühsam alle Speicherzuweisungen ändern in:

  ```
  p = malloc(sizeof(double));
  ```

- als numerische Konstante:

  ```
  p = malloc(sizeof(2));
  ```

  Hiermit werden 4 Byte (!) reserviert, auf deren Anfangsadresse der Zeiger `p` verweist. Es werden nicht – wie vielleicht irrtümlicherweise angenommen – 2 Byte reserviert, sondern es wird eben so viel Speicher reserviert, wie es dem Datentyp im `sizeof`-Operator auf dem jeweiligen System entspricht. Der Wert 2 entspricht gewöhnlich auf 32-Bit-Rechnern 4 Byte (`int`). Somit kann die Verwendung einer numerischen Konstante sehr verwirrend sein.

- Sie können auch den dereferenzierten Zeiger selbst für den `sizeof`-Operator verwenden:

  ```
  p = malloc(sizeof(*p));
  ```

  Aber Achtung: Wenn Sie den Dereferenzierungsoperator (*) vergessen, wie im folgenden Listing, dann geht dies garantiert schief:

```
/* malloc_3.c */
#include<stdio.h>
#include<stdlib.h>

int main(void) {
   double *p1,*p2;

   p1 = malloc(sizeof(p1)); // Fehler
   p2 = malloc(sizeof(p2)); // Fehler

   if(p1 != NULL && p2 != NULL) {
         *p1 = 5.15;
```

```
            printf("p1 = %f\n",*p1);
            *p2 = 10.99;
            printf("p2 = %f\n",*p2);
        }
    return EXIT_SUCCESS;
}
```

Wenn Sie »Glück« haben, stürzt das Programm ab. Im schlimmsten Fall funktioniert das Programm aber und gibt die richtigen Zahlen aus. Das wäre jedoch purer Zufall, denn ohne den Dereferenzierungsoperator wird nicht ein `double`-Wert an `malloc()` übergeben, sondern die Größe des Zeigers. Und diese beträgt immer vier (bei 32-Bit-Rechnern) statt der erforderlichen 8 Byte. Wenn jetzt ein anderer Wert an diese Adresse gelangt, ist der weitere Verlauf des Programms nicht absehbar. Es kommt zu einer sogenannten *Überlappung der Speicherbereiche*, und diese ist unsicher.

15.5 »free()« – Speicher wieder freigeben

Wenn Sie Speicher vom Heap angefordert haben, sollten Sie ihn auch wieder zurückgeben. Der allozierte Speicher wird mit folgender Funktion freigegeben:

```
#include <stdlib.h> // Für free muss zusätzlich stdlib.h eingebunden werden.
void free (void *p)
```

Der Speicher wird übrigens auch ohne einen Aufruf von `free()` freigegeben, wenn sich das Programm beendet. Leider gilt dies nicht für ältere Systeme.

Hinweis: Speicherfreigabe bei Beendigung eines Programms

Zwar wird generell behauptet (und es ist auch meistens der Fall), dass bei der Beendigung eines Programms das Betriebssystem den reservierten und nicht mehr freigegebenen Speicher selbst organisiert und somit auch wieder freigibt, aber dies wird nicht vom ANSI/ISO-Standard gefordert.

Somit hängt dieses Verhalten also von der Implementierung der Speicherverwaltung des Betriebssystems ab. Unter Windows-Versionen vor XP wurden z. B. Speicherinhalte von Grafikkontexten (HDCs) beim Beenden von GUI-Anwendungen nicht automatisch wieder freigegeben. Auf der sicheren Seite sind Sie also, wenn Sie den benutzten Speicher vor dem Beenden des Programms stets wieder freigeben.

Hier sehen Sie ein Beispiel zu `free()`:

```
/* free_1.c */
#include<stdio.h>
#include<stdlib.h>
```

```
int main(void) {
   int *p = malloc(sizeof(int));

   if(p != NULL) {
      *p=99;
      printf("Allokation erfolgreich ... \n");
      }
   else {
      printf("Kein virtueller RAM mehr verfügbar ... \n");
      return EXIT_FAILURE;
   }
   if(p != NULL) {
      printf("Speicher wurde wieder freigegeben.\n");
      free(p);

 }
 return EXIT_SUCCESS;
}
```

Listing 15.2 »free_1.c« zeigt Ihnen, wie Sie mit »malloc()« reservierten Speicher wieder freigeben.

Es wird auch überprüft, dass nur wirklich reservierter Speicherplatz wieder freigegeben wird. Der mit `free()` freigegebene Speicherplatz wird danach zwar als frei markiert, aber `p` zeigt immer noch auf die ursprüngliche Speicherstelle. Das folgende Beispiel zeigt, dass dies wirklich so ist:

```
/* free_2.c */
#include<stdio.h>
#include<stdlib.h>

int main(void) {
   int *p = malloc(sizeof(int));

   if(p != NULL) {
         *p=99;
         printf("Allokation erfolgreich ... \n");
   }
   else {
      printf("Kein virtueller RAM mehr verfügbar ... \n");
      return EXIT_FAILURE;
   }
   printf("vor free() *p = %d\n", *p); // 99
   if(p != NULL)
```

```
    free(p);
  printf("nach free() *p = %d\n", *p); // evtl. Unsinn
  return EXIT_SUCCESS;
}
```

Listing 15.3 »free_2.c« zeigt, dass mit »free()« freigegebener Speicher nicht die Inhalte löscht.

Hinweis: Zugriffe auf zuvor freigegebene Speicherinhalte

Da der Heap üblicherweise aus Performance-Gründen nicht wieder reduziert wird, konnten Sie wie im Beispiel eventuell auf den freigegebenen Speicherplatz und dessen Inhalt wieder zugreifen. Vielleicht hat das Programm sogar zweimal `99` ausgegeben – oder auch nicht. Dieses Verhalten ist leider nicht portabel und wird auch nicht vom ANSI-C-Standard gefordert. Sie haben ja oben schon gesehen, dass die Verwaltung des Heaps vom Betriebssystem abhängig ist. Sofern Sie also vorhaben, so etwas absichtlich in der Praxis auszuführen (warum auch immer), ist das Verhalten undefiniert. Es gibt natürlich auch Hacks, die auf dem Heap Schadcode ablegen und ausführen. Dabei werden sogenannte *Memory Leaks* ausgenutzt. Mehr Informationen dazu gibt es später in Kapitel 27, »Sicheres Programmieren«.

Wenn Sie absolut sicher sein wollen, dass der Zeiger nichts mehr zurückgibt, dann übergeben Sie dem Zeiger nach der Freigabe von Speicher einfach den `NULL`-Zeiger:

```
free(p);
p = NULL;
```

Dies können Sie wie folgt in ein Makro verpacken:

```
#define my_free(x)  free(x); x = NULL
```

Internes zur Speicherverwaltung

Die Speicherverwaltung merkt sich die Größe eines jeden Speicherblocks, der von Ihnen angefordert wurde. Deswegen ist es auch nicht nötig, die Größe des Blocks (als echte Byte-Größe) anzugeben, um den Speicher mit `free()` wieder freizugeben. Leider gibt es daher jedoch keinen portablen Weg, um zu erfahren, wie groß dieser Speicherblock denn tatsächlich ist. Unter Windows können Sie hierfür die Windows-API benutzen. Allerdings laufen Ihre Programme dann nicht unter Linux.

Was `malloc()` und die weiteren Speicherallokationsfunktionen so bedeutend und wichtig macht, ist die Möglichkeit, von jedem beliebigen Datentyp Speicher anfordern zu können – sind es nun einfache Datentypen wie Strings, Arrays oder komplexe Strukturen.

Natürlich können Sie auch Speicherplatz für ein char-Array zur Laufzeit anfordern. Das folgende Beispiel demonstriert dies:

```
/* malloc_4.c */
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define BUF 80

int main(void) {
   char puffer[BUF];
   char *dyn_string;

   printf("Ein Text mit max. 80 Zeichen: ");
   fgets(puffer, BUF, stdin);

   dyn_string = malloc(strlen(puffer) + 1);
   if(dyn_string != NULL)
      strncpy(dyn_string, puffer, strlen(puffer) + 1);
   else {
      printf("Konnte keinen Speicherplatz reservieren\n");
      return EXIT_FAILURE;
   }
   printf("%s",dyn_string);
   free(dyn_string);
   return EXIT_SUCCESS;
}
```

Listing 15.4 »malloc_4.c« demonstriert, wie ein String dynamisch zur Laufzeit erzeugt werden kann.

Es muss jedoch erwähnt werden, dass diese Art, dynamisch Speicher für einen Text zu reservieren, noch recht unflexibel ist. Mit

```
dyn_string = malloc(strlen(puffer) + 1);
```

wird exakt so viel Speicher angefordert, wie zuvor mit fgets() in den String puffer eingelesen wurde. Im Verlauf des Buches werden Sie erfahren, wie Sie viel effektiver dynamischen Speicher für Text anfordern. Denn bei diesem Beispiel hätten Sie es ja auch gleich beim char-Array puffer belassen können. Aber auch in anderen Situationen ist nicht immer klar, wie viel Speicher man benötigen wird, z. B. in einem Online-Computerspiel mit mehreren Spielern, bei dem nicht von Anfang an klar ist, wie viele Personen mitspielen. Eine effektive Möglichkeit, dynamisch Speicher für mehrere Spieler anzulegen, sind z. B. die verketteten Listen. Auch diese werden später noch ausführlich besprochen.

15.5.1 Einige ergänzende Punkte zur Freispeicherverwaltung

Ohne mich zu sehr in die Details der Freispeicherverwaltung verstricken zu wollen, soll dieses Thema kurz behandelt werden, denn einige der folgenden Begriffe können immer wieder in Internetforen oder ergänzender Literatur auftauchen. Sie können diesen Unterpunkt auch gerne überspringen, denn für die Benutzung von `malloc()` ist es nicht unbedingt nötig, noch mehr über Prozessorarchitektur und Register zu erfahren. Allerdings wird vieles einfacher, wenn man weiß, welche unterschiedlichen Möglichkeiten es gibt, Daten abzuspeichern. Als Anfänger kann es Ihnen im Prinzip egal sein, wie ein Betriebssystem seinen Speicher reserviert, aber wenn Sie irgendwann professionelle Programme schreiben, die häufig Speicher vom Heap anfordern (dies ist z. B. bei GUI-Anwendungen der Fall), wäre manches Mal ein wenig Hintergrundwissen wünschenswert.

Außer dem Hauptspeicher und dem Festplattenspeicher gibt es noch weitere Möglichkeiten, Daten zu speichern und zu verarbeiten. Dabei wird von einer *Speicherhierarchie* gesprochen (oder manchmal auch von *Levels*), die sich in die folgenden sechs Schichten aufteilt:

- Prozessor-Register (Dies sind die schnellsten Zugriffseinheiten.)
- *First Level Cache* der CPU (On-Chip-Cache, 8–512 KB)
- *Second Level Cache* der CPU (128–4096 KB)
- Hauptspeicher (RAM, beispielsweise 8 GB, bei Gaming-PCs 8–64 GB, bei Servern mehrere Terabyte)
- Sekundärspeicher (Festplatte, vielleicht 1000 GB, bei Servern und Gaming-PCs auch mehr)
- Tertiärspeicher (Magnetband, ca. 20–160 GB, kaum noch verwendet)

Als C-Programmierer nutzen Sie allerdings vorwiegend den Hauptspeicher und die Prozessorregister. Moderne Betriebssysteme verwenden dabei das sogenannte *Paging* zur Verwaltung des Speichers. Als *Paging* wird die Unterteilung des Speichers in Seiten (*Pages*) einer festen Größe bezeichnet. Die Logik zur Verwaltung von Speicherseiten steckt als *MMU* (Memory Management Unit) im Prozessor und ist inzwischen so komplex, dass man für die Erklärung der Mechanismen ein weiteres Buch benötigt, z. B. »Moderne Betriebssysteme« von Andrew S. Tanenbaum.

Die Größe einer Seite beträgt bei den gängigen Betriebssystemen 512 KB oder 1024 KB, kann aber bei älteren UNIX-Systemen auch 1024 Byte oder 4 KB betragen. Die *MMU* kann auch sogenannte *virtuelle Adressen* bilden, die sich nicht einmal im Hauptspeicher befinden müssen. Ein virtuelles Speichersystem ist erforderlich, damit auch mehrere Programme laufen können, die nicht alle in den physischen Speicher (d. h. in den tatsächlich vorhandenen Speicher, also das RAM) passen würden. Dafür stellt Ihnen das Betriebssystem sogenannten *Swapspace* (also Speicher, der auf die Festplatte ausgelagert werden kann) zur Verfügung sowie einen Mecha-

nismus, mit dem aus einer virtuellen Adresse wieder eine physische Adresse gemacht werden kann (denn mit virtuellen Adressen allein könnte kein Programm laufen). Sie sehen also, dass das Thema »Speicherverwaltung« inzwischen sehr komplex ist und in diesem Buch nicht ausführlich behandelt werden kann.

Hinweis: Adressraum von Prozessen

Jedem Prozess steht ein eigener virtueller Adressraum und darin eine eigene Speicherverwaltung zur Verfügung. Meistens wird zusätzlich das Swapping benutzt. Dabei wird ein Prozess in den Hauptspeicher geladen und läuft eine gewisse Zeit, bis er anschließend auf die Festplatte ausgelagert wird. Dieses Swapping findet z. B. statt, wenn nicht mehr genügend Speicherplatz vorhanden ist, um einen Prozess ausführen zu können. Leider kann man das Swapping nicht wirklich steuern, sondern man kann nur die Größe der Swap-Partition ändern (Linux) oder sich in die Tiefen der Systemsteuerung einarbeiten (Windows). Meistens müssen Sie das Swapping aber auch nicht direkt steuern, denn in 95 Prozent der Fälle erledigt das Betriebssystem diese Aufgabe perfekt. Eine Faustregelt gilt aber dennoch: Je mehr RAM Sie haben, desto weniger oft werden Speicherinhalte auf die Festplatte ausgelagert; deshalb laufen PCs mit mehr RAM als nötig flüssiger als PCs mit knapp bemessenem RAM. Da RAM relativ billig ist, ist es immer besser, nicht mit Speicherriegeln zu geizen.

In Abschnitt 15.1.4 haben Sie bereits etwas über den Heap erfahren. Sie wissen, dass der Heap ein zusammenhängender Speicherplatz im Arbeitsspeicher ist, von dem Sie als Programmierer Speicher anfordern (allozieren) können. Das Betriebssystem verwaltet diesen Speicherbereich als eine Kette von freien Speicherblöcken, die nach aufsteigenden Speicheradressen sortiert ist. Jeder dieser Blöcke enthält Informationen wie die Gesamtlänge oder den nächsten freien Block. Benötigen Sie jetzt Speicherplatz, durchläuft das Betriebssystem diesen Speicherblock nach verschiedenen Verfahren. Dabei wird von einer *prozessinternen Freispeicherverwaltung* gesprochen.

15.5.2 Prozessinterne Freispeicherverwaltung

Durch einen Aufruf von `malloc()` sucht das Betriebssystem einen zusammenhängenden Speicherblock, der den Anforderungen entspricht. Auf der Suche nach diesem Speicherplatz gibt es verschiedene Strategien bzw. Algorithmen, die im Betriebssystem (genauer: im Kernel) implementiert sind.

Als sehr einfaches Beispiel dienen freie Speicherbereiche (Größen in Byte), die so im System angeordnet sind, wie Sie es in Abbildung 15.3 sehen.

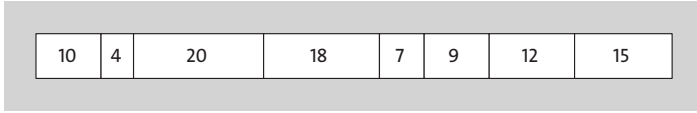

Abbildung 15.3 Freie Speicherbereiche im System

Sie fordern jetzt von diesen freien Speicherbereichen mit der Funktion `malloc()` folgende Speicherblöcke an:

```
ptr1 = malloc(10);
ptr2 = malloc(12);
ptr3 = malloc(9);
```

Anhand des freien Speicherbereichs (siehe Abbildung 15.4) und den drei hier gezeigten Speicheranforderungen will ich Ihnen im Folgenden die einzelnen Verfahren erklären.

First-Fit-Verfahren

Das First-Fit-Verfahren ist die einfachste Variante, die auch schon unter DOS oder auf noch älteren Systemen wie dem Amiga eingesetzt wurde. Beim First-Fit-Verfahren durchläuft die Speicherverwaltung die Liste der Reihe nach und alloziert den erstbesten freien Bereich, der groß genug ist. Somit sieht die Speicherbelegung im First-Fit-Verfahren so aus wie in Abbildung 15.4 gezeigt (die dunklen Bereiche sind die neu hinzugekommenen Speicherblöcke).

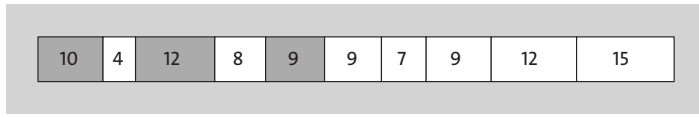

Abbildung 15.4 First-Fit-Verfahren

Next-Fit-Verfahren

Das Next-Fit-Verfahren funktioniert wie First-Fit, nur merkt sich das Next-Fit-Verfahren die aktuelle Position und fährt bei der nächsten Suche nach freiem Speicher von dieser Position aus fort. Es gibt zahlreiche Dateisysteme, die dieses Verfahren benutzen, um freie Festplattenblöcke anzufordern. Leider entstehen hierdurch viele ungenutzte Lücken und ursprünglich zusammenhängende Daten zerfallen unter Umständen in mehrere Teile. Bei Dateisystemen sagt man, das Dateisystem ist fragmentiert.

Best-Fit-Verfahren

Beim Best-Fit-Verfahren wird die gesamte Speicherliste durchsucht, bis ein kleinstmögliches Loch gefunden wird (siehe Abbildung 15.5). Mit diesem leider langsamen Verfahren wird eine optimale Speicherausnutzung garantiert.

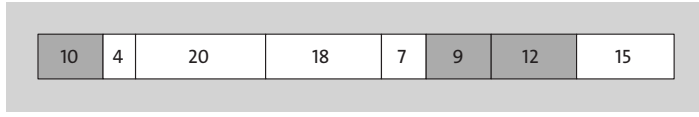

Abbildung 15.5 Best-Fit-Verfahren

Worst-Fit-Verfahren

Das Worst-Fit-Verfahren ist das Gegenteil von Best-Fit. Dabei wird in der Liste nach dem größten verfügbaren freien Bereich gesucht (siehe Abbildung 15.6).

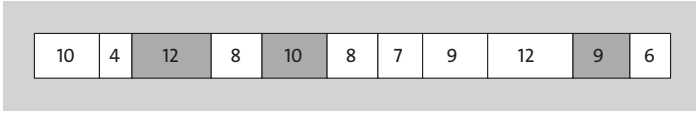

Abbildung 15.6 Worst-Fit-Verfahren

Quick-Fit-Verfahren

Das Quick-Fit-Verfahren unterhält getrennte Listen für freie Bereiche gebräuchlicher Größe. Viele Dateisysteme verwenden auch zusätzlich dieses Verfahren, aber dadurch wird die Blockverwaltung komplexer und anfälliger für Fehler.

Buddy-Verfahren

Das Buddy-Verfahren verwendet für jede Speichergröße eine eigene Liste. Die Zeiger auf die Listenköpfe werden dabei in einem Array zusammengefasst. Bei diesem Verfahren werden nur Blöcke von Zweierpotenzen verwendet (1 Byte, 2 Byte, 4 Byte, 8 Byte, 16 Byte, 32 Byte, 64 Byte, ... , 512 Byte, 1 KB, ..., 512 KB, 1 MB ...). Wird Speicher angefordert, der nicht exakt einem dieser Blockgrößen entspricht, wird ein Block mit der nächsthöheren Zweierpotenz verwendet. Die Blöcke werden außerdem markiert, um anzuzeigen, ob sie zur Anwendung frei sind. Ist bei Speicheranforderung kein gewünschter Block frei, wird ein Block in zwei gleich große Blöcke aufgeteilt. Leider wird hierdurch die Blockverwaltung komplexer und fehleranfälliger.

Hinweis: Unterschiedliche Strategien der Speicherverwaltung

Jede dieser vorgestellten Strategien der Freispeicherverwaltung hat natürlich ihren Sinn, und meistens wird eine Mischung mehrerer Verfahren verwendet. Vorwiegend dienen solche Verfahren dazu, einen Verschnitt des Speichers zu vermeiden. Das bedeutet, dass der Speicher schlecht ausgenutzt wird, wenn sehr viele unterschiedliche Stücke verwaltet werden müssen. So kann es passieren, dass einzelne Fragmente des Speichers wahrscheinlich nicht mehr verwendet werden können, obwohl sie mit `free()` freigegeben wurden.

Ein zweiter Grund für die verschiedenen Strategien der Freispeicherverwaltung ist natürlich die Geschwindigkeit. Je schneller wieder auf einen Speicherblock zurückgegriffen werden kann, umso besser ist es.

Freigabe von Speicher

Der Speicherplatz, der wieder freigegeben wird, wird meist nicht direkt an das Betriebssystem zurückgegeben, sondern in die Freispeicherliste eingehängt. Irgendwann kommt

dann der *Garbage Collector* (also die Müllabfuhr) vorbei und räumt die Liste auf. Auch auf diesen Vorgang hat man keinen Einfluss, weil er unter anderem vom Betriebssystem abhängig ist. Manche Programmiersprachen (z. B. Java) haben auch ihren eigenen Garbage Collector. Nach diesem Ausflug, der schon mehr in Richtung Programmierung von Betriebssystemen ging, kehren wir nun wieder zur Praxis zurück.

15.6 Dynamische Arrays

Wenn mit der Funktion `malloc()` ein zusammenhängender Speicherbereich reserviert werden kann, dann muss es auch möglich sein, Speicher für ein Array während der Laufzeit zu reservieren. Bei einem zusammenhängenden Speicher können Sie davon ausgehen, dass dieser in einem Block (d. h. lückenlos) zur Verfügung gestellt wird. In dem folgenden Beispiel wird ein solches dynamisches Array erzeugt:

```
/* dyn_array_1.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(void) {
   int *value;
   int size, i = 0;

   printf("Wie viele Werte benötigen Sie : ");
   scanf("%d", &size);
   value = malloc(size*sizeof(int));
   if( NULL == value ) {
      printf("Fehler bei malloc....\n");
      return EXIT_FAILURE;
   }
   while( i < size ) {
      printf("Wert für value[%d] eingeben : ", i);
      scanf("%d", &value[i]);
      i++;
   }
   printf("Hier Ihre Werte\n");
   for(i=0; i < size; i++)
      printf("value[%d] = %d\n", i, value[i]);
   return EXIT_SUCCESS;
}
```

Listing 15.5 »dyn_array_1.c« reserviert zur Laufzeit Speicherplatz für ein Array

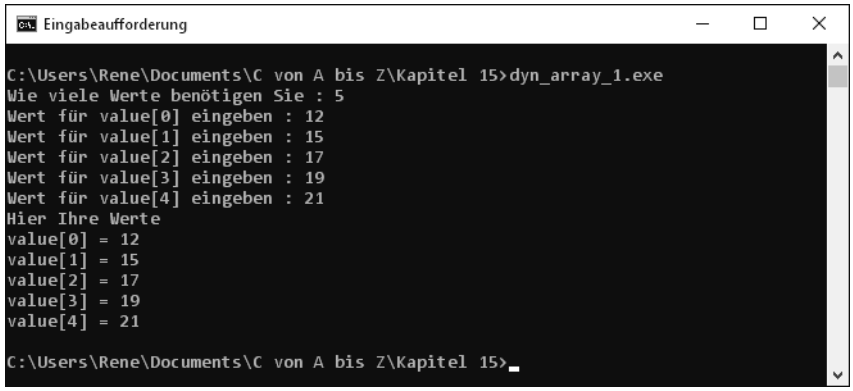

Abbildung 15.7 Dynamisch erzeugtes Array

Mit

```
value = malloc(size*sizeof(int));
```

wird ein zusammenhängender Speicherbereich mit `size` `int`-Werten reserviert. Danach werden mit

```
while(i < size) {
   printf("Wert für value[%d] eingeben : ", i);
   scanf("%d", &value[i]);
   i++;
}
```

diesem Speicherbereich Werte zugewiesen. Zum besseren Verständnis zeige ich hier dasselbe Programm nochmals, aber statt mit Arrays nun mit Zeigern:

```
/* dyn_array_2.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(void) {
   int *value;
   int size, i=0;

   printf("Wie viele Werte benötigen Sie : ");
   scanf("%d", &size);

   value = malloc(size*sizeof(int));
   if(NULL == value) {
      printf("Fehler bei malloc...!!\n");
      return EXIT_FAILURE;
```

```
   }
   while(i < size) {
      printf("Wert für value[%d] eingeben : ",i);
      scanf("%d",(value+i));
      i++;
   }
   printf("Hier Ihre Werte\n");
   for(i=0; i<size; i++)
      printf("value[%d] = %d\n", i, *(value+i));
   return EXIT_SUCCESS;
}
```

Listing 15.6 »dyn_array_2.c« erzeugt ein dynamisches Array mithilfe von Zeigern.

Da `*value`, `value[0]` und `*(value+1)`, `value[1]` immer auf dieselbe Speicheradresse verweisen, ist es egal, wie darauf zugegriffen wird.

Das Programm ist jetzt etwas unflexibel. Was ist, wenn Sie für fünf weitere Elemente Speicherplatz benötigen? Mit der Funktion `realloc()` wäre dies recht einfach zu realisieren. Aber diese Funktion steht jetzt noch nicht zur Debatte. Die Speicherzuweisung ist auch mit `malloc()` möglich, wenn auch etwas umständlicher, beispielsweise wie folgt:

```
/* dyn_array_3.c */
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int main(void) {
   int *value,*temp;
   int i=0, more;
   int size, merker = 0;

   printf("Wie viele Werte benötigen Sie : ");
   scanf("%d", &size);
   value = malloc(size*sizeof(int));
   if(NULL == value) {
      printf("Fehler bei malloc...!! n");
      return EXIT_FAILURE;
   }
   do {
      while(merker < size) {
         printf("Wert für value[%d] eingeben : ",merker);
         scanf("%d",&value[merker]);
         merker++;
      }
```

```
      printf("Neuen Platz reservieren (0=Ende) : ");
      scanf("%d",&more);
      temp = malloc(size*sizeof(int));
      if(NULL == temp) {
         printf("Kann keinen Speicher mehr reservieren!\n");
         return EXIT_FAILURE;
      }
      for(i=0; i<size; i++)
         temp[i]=value[i];
      size+=more;
      value = malloc(size * sizeof(int));
      if(NULL == value) {
         printf("Kann keinen Speicher mehr reservieren!\n");
         return EXIT_SUCCESS;
      }
      for(i=0; i<size; i++)
         value[i]=temp[i];
   }while(more!=0);
   printf("Hier Ihre Werte\n");
   for(i=0; i<size; i++)
      printf("value[%d] = %d\n" ,i ,value[i]);
   return EXIT_SUCCESS;
}
```

Listing 15.7 »dyn_array_3.c« zeigt Ihnen, wie Sie ein dynamisches Array zur Laufzeit vergrößern können.

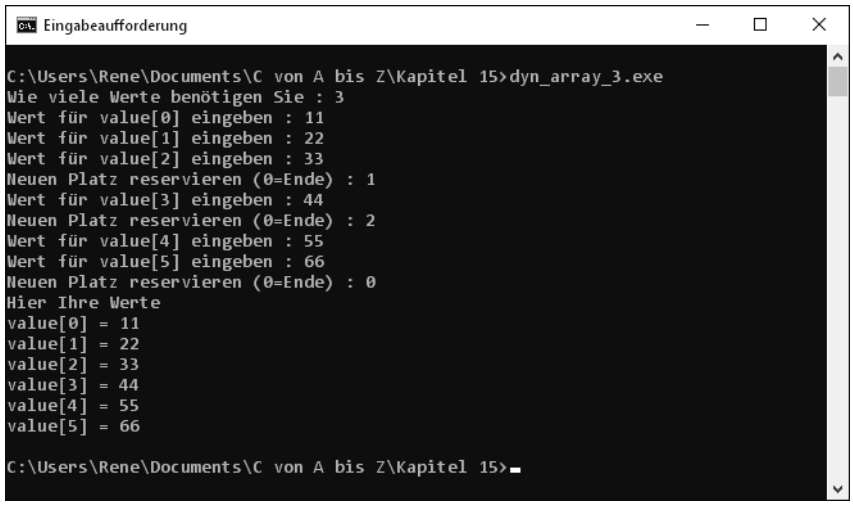

Abbildung 15.8 Ein dynamisch reserviertes Array dynamisch erweitern

Bevor Sie für das bereits dynamisch reservierte Array erneut Speicherplatz reservieren können, müssen Sie die bereits eingegebenen Werte erst einmal in ein temporär alloziertes Array zwischenspeichern. Danach kann neuer Speicherplatz für das Array reserviert werden, in den anschließend die Werte aus dem temporären Array zurückkopiert werden. Das alles ist ziemlich aufwendig., wir ersparen es uns zunächst, das anhand eines `char`-Arrays (Strings) zu demonstrieren.

15.7 Speicher dynamisch reservieren mit »realloc()« und »calloc()«

In der Headerdatei *<stdlib.h>* sind noch zwei weitere Funktionen zum dynamischen Reservieren von Speicher deklariert. Hier sehen Sie die Syntax zu diesen Funktionen:

```
void *calloc(size_t anzahl, size_t groesse);
void *realloc(void *zgr, size_t neuegroesse);
```

Die Funktion `calloc()` ist der Funktion `malloc()` sehr ähnlich, nur dass es bei der Funktion `calloc()` nicht einen, sondern zwei Parameter gibt. Im Gegensatz zu `malloc()` können Sie mit `calloc()` noch die `anzahl` von Speicherobjekten angeben, die reserviert werden soll. Wird z. B. für 100 Objekte vom Typ `int` Speicherplatz benötigt, so erledigen Sie dies mit `calloc()` folgendermaßen:

```
int *zahlen;

zahlen = calloc(100,sizeof(int));
```

Außerdem werden mit der Funktion `calloc()` alle Werte des allozierten Speicherbereichs automatisch mit dem Wert `0` initialisiert. Bei `malloc()` hat der reservierte Speicherplatz zu Beginn einen undefinierten Wert. Allerdings können Gleitpunkt- und Zeiger-Nullen auch ganz anders dargestellt werden, weshalb man sich auf solchen Feldern nicht auf die Nullen verlassen kann. Gleichwertig zu `calloc()` verhält sich außerdem folgendes Codekonstrukt mit `malloc()`:

```
ptr = calloc(100, sizeof(int));

// Alternative dafür mit malloc(); erfüllt denselben Zweck
ptr = malloc(100 * sizeof(int));
memset(ptr, 0, 100 * sizeof(int));
```

Da `calloc()` außer den beiden eben genannten Unterschieden genauso funktioniert wie die Funktion `malloc()`, gehe ich nicht mehr näher darauf ein.

Interessanter ist dagegen die dynamische Speicherreservierung mit der Funktion `realloc()`. Mit dieser Funktion ist es möglich, während des laufenden Programms so viel Speicher zu reservieren, wie Sie benötigen. Des Weiteren können Sie sich darauf

verlassen, dass ein neuer Pool mit `malloc()` erstellt wird und die ganzen Ergebnisse herüberkopiert werden, wenn im aktuellen Speicherblock nicht mehr genügend freier Speicher vorhanden ist.

Mit `realloc()` ist es noch einfacher, z. B. dynamische Arrays zu programmieren. Die Anfangsadresse des dynamischen Arrays ist diejenige, auf die der Zeiger (`zgr`) zeigt. Der Parameter `neuegroesse` dient dazu, einen bereits zuvor allozierten Speicherplatz auf `neuegroesse` Byte zu vergrößern. Die Funktion `realloc()` ermöglicht es auch, den Speicherplatz zu verkleinern. Dazu wird einfach der hintere Teil des Speicherblocks freigegeben, während der vordere Teil unverändert bleibt. Bei einer Vergrößerung des Speicherplatzes mit `realloc()` behält der vordere Teil auf jeden Fall seine Werte, und der neue Teil wird einfach hinten angehängt. Dieser angehängte Teil ist aber wie bei `malloc()` undefiniert. Hier sehen Sie ein kleines Beispiel dafür, wie ein Array mit der Funktion `realloc()` dynamisch erstellt wird:

```
/* realloc_1.c */
#include<stdio.h>
#include<stdlib.h>

int main(void) {
   int n=0, max=10, z,i;
   int *zahlen=NULL;

   /* Wir reservieren Speicher für 10 int-Werte mit calloc. */
   zahlen = calloc(max, sizeof(int));
   if(NULL == zahlen) {
      printf("Kein virtueller RAM mehr vorhanden ... !");
      return EXIT_FAILURE;
   }
   printf("Zahlen eingeben --- Beenden mit 0\n");
   /* Endlossschleife */
   while(1) {
      printf("Zahl (%d) eingeben : ", n+1);
      scanf("%d", &z);
      if(z==0)
         break;
      /* Reservierung von Speicher während der Laufzeit
       * des Programms mit realloc */
       if(n >= max) {
          max += max;
          zahlen = realloc(zahlen,max*sizeof(int));
          if(NULL == zahlen) {
             printf("Kein virtueller RAM mehr vorhanden ... !");
```

```
            return EXIT_FAILURE;
         }
         printf("Speicherplatz reserviert "
                " (%d Byte)\n", sizeof(int) * max);
      }
      zahlen[n++] = z;
   }
   printf("Folgende Zahlen wurden eingegeben ->\n\n");
   for(i = 0; i < n; i++)
      printf("%d ", zahlen[i]);
   printf("\n");
   free(zahlen);
   return EXIT_SUCCESS;
}
```

Listing 15.8 »realloc_1.c« verwendet die Funktion »realloc()«, um ein dynamisches Array zur Laufzeit zu vergrößern.

Den benötigten Speicherbedarf könnten Sie in diesem Beispiel auch einzeln allozieren. Die einfache Anwendung dieser Funktion soll nicht darüber hinwegtäuschen, dass auch hier erst der alte Speicherbereich temporär zwischengespeichert werden muss, so wie bei der Funktion `malloc()`. In diesem Fall ist es aber einfacher, da Sie sich nicht mehr selbst darum kümmern müssen.

Im Beispiel wurde der Speicherplatz nach jedem erneuten Allozieren mit `calloc()` gleich verdoppelt (`max += max`). Dies ist nicht optimal. Benötigt ein Programm z. B. täglich 500 `double`-Werte, wäre es am sinnvollsten, erst nach 500 `double`-Werten neuen Speicher zu allozieren. Somit müsste das Programm nur einmal am Tag neuen Speicher bereitstellen.

Dasselbe Beispiel lässt sich recht ähnlich und einfach auch für `char`-Arrays umschreiben. Das folgende Listing demonstriert die dynamische Erweiterung eines Strings:

```
/* dyn_strings_1.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define BUF 255

int main(void) {
   size_t len;
   char *str = NULL;
   char puffer[BUF];
```

```
   printf("Ein dynamisches char-Array für Strings\n");
   printf("Eingabe machen : ");
   fgets(puffer, BUF, stdin);
   str = malloc(strlen(puffer)+1);
   if(NULL == str) {
      printf("Kein virtueller RAM mehr vorhanden ... !");
      return EXIT_FAILURE;
   }
   strcpy(str, puffer);
   printf("Weitere Eingabe oder beenden mit \"END\"\n>");
   /* Endlossschleife */
   while(1) {
      fgets(puffer, BUF, stdin);
      /* Abbruchbedingung */
      if(strcmp(puffer,"end\n")==0 || strcmp(puffer,"END\n")==0)
         break;
       /* aktuelle Länge von str zählen für realloc */
       len = strlen(str);
       /* neuen Speicher für str anfordern */
       str = realloc(str,strlen(puffer)+len+1);
       if(NULL == str) {
          printf("Kein virtueller RAM mehr vorhanden ... !");
          return EXIT_FAILURE;
       }
       /* Hinten anhängen */
       strcat(str, puffer);
   }
   printf("Ihre Eingabe lautete: \n");
   printf("%s", str);
   free(str);
   return EXIT_SUCCESS;
}
```

Listing 15.9 »dyn_strings_1.c« demonstriert die dynamische Erweiterung von Strings zur Laufzeit.

Beim `char`-Array läuft es so ähnlich ab wie schon im Beispiel mit den `int`-Werten zuvor. Sie müssen allerdings immer darauf achten, dass bei erneuter Speicheranforderung mit `realloc()` das String-Ende-Zeichen berücksichtigt wird (`+1`). Ansonsten ist der Vorgang recht simpel: String einlesen, Zeichen zählen, erneut Speicher reservieren und hinten anhängen.

15.8 Speicher vom Stack anfordern mit »alloca()« (nicht ANSI C)

Die Funktion `alloca()` ist nicht vom ANSI-C-Standard vorgeschrieben und wird mit dem C23-Standard verschwinden. Deshalb bespreche ich sie hier nur kurz. Die Syntax zu `alloca()` lautet:

```
void *alloca(size_t size);
```

Bei Linux befindet sich `alloca()` in der Headerdatei *<stdlib.h>*, und unter Windows sollte sich diese Funktion in der Headerdatei *<malloc.h>* befinden.

`alloca()` kann bezüglich der Verwendung mit `malloc()` verglichen werden, aber mit dem Unterschied, dass `alloca()` den Speicherplatz nicht vom Heap, sondern vom Stack anfordert. Die Funktion `alloca()` vergrößert den Stack-Bereich (Stack Frame) der aktuellen Funktion.

`alloca()` hat außerdem den Vorteil, dass der Speicherplatz nicht extra mit `free()` freigegeben werden muss, da er automatisch beim Verlassen der Funktion freigegeben wird. Ganz im Gegenteil, es darf hierbei nicht einmal `free()` benutzt werden, da der `alloca()`-Block nicht von dem `malloc()`-Pool kommt, sondern eben vom Stack. Ein `free()` hätte dabei böse Folgen. Die Funktion `alloca()` wird ansonsten genauso verwendet wie die Funktion `malloc()`.

Ich rate Ihnen, `alloca()` in neuen Projekten nicht mehr zu verwenden, da sie eben ab C23 nicht mehr Teil des Standards sein wird. Außerdem kann `alloca()` sowohl gut angegriffen als auch für Angriffe benutzt werden, die bewusst Stack Overflows auslösen. Auf diese Weise kann z. B. eine Anwendung dazu gezwungen werden, fremden Code auszuführen. Die Sache ist wirklich ganz einfach: Der Angreifer muss dem Prozessor nur direkt vor der Rückkehr aus einer Funktion eine andere Rücksprungadresse auf dem Stack unterjubeln. Wenn dann dort plötzlich Schadcode steht, merkt der Prozessor nicht einmal, dass er übers Ohr gehauen wurde.

15.9 Ergänzende Anmerkungen zu »free()«

Wie Sie bereits wissen, lautet die Syntax der Funktion zur Freigabe von Speicher wie folgt:

```
#include <stdlib.h>

void free(void *zeiger);
```

`free()` wurde bereits des Öfteren verwendet. Diese Funktion dient zur Freigabe von Speicher, der zuvor mit Funktionen wie `malloc()`, `calloc()` oder `realloc()` angefordert wurde. Folgendes sollten Sie bei dieser Funktion aber noch beachten:

- Falls ein Speicherbereich freigegeben wird, der nicht zuvor mit `malloc()`, `calloc()` oder `realloc()` alloziert wurde, kann dies unvorhersehbare Folgen haben. Die ganze Speicherverwaltung kann so aus dem Tritt gebracht werden, zumindest unter Systemen wie DOS, das den Speicher nicht schützt. Daher sollten Sie darauf achten, dass wirklich nur Speicherplatz freigegeben wird, der auch alloziert wurde.
- Speicher, den Sie mit `free()` freigeben, wird während der Laufzeit des Prozesses nicht wirklich an den Kern zurückgegeben, sondern in einem sogenannten `malloc()`-Pool gehalten, um bei Bedarf während des laufenden Prozesses wieder darauf zurückgreifen zu können. Erst dann, wenn der Prozess beendet wurde, geht der Speicher wieder zurück an den Kern.

Beim Allozieren des Speichers mit `malloc()` wird der Aspekt, den Speicher wieder freizugeben, häufig vernachlässigt. In den Beispielen dieses Buchs dürfte ein vergessenes `free()` nicht allzu tragisch sein, da ein Programm, das sich beendet, seinen Speicherplatz gewöhnlich automatisch wieder freigibt (bei einem gut programmierten Betriebssystem). Schlimmer dürfte der Fall aber bei sogenannten *Server-Programmen* sein, die oft wochen- bzw. jahrelang laufen müssen. Das Programm wird zwangsweise immer langsamer. Man spricht dabei von *Memory Leaks* (Speicherlecks). Das passiert aber doch sicherlich nur Anfängern? Das ist leider nicht so, denn viele professionelle Angriffe auf Server nutzen Memory Leaks aus. Nicht umsonst verdienen sich viele Softwarehersteller wie z. B. *Kaspersky Labs* eine goldene Nase mit Programmen, die solche und andere Programmierfehler entdecken.

Memory Leaks gehören neben Buffer Overflows zu den Fehlern, die C-Programmierer am häufigsten machen, und dies sind nicht nur Anfänger. Sie können dies im Endeffekt mit dem Autofahren vergleichen: Bei einem Unfall sind es auch immer wieder dieselben Fahrfehler, die zu einem Unfall führen, aber nicht immer sind nur Anfänger an den Unfällen beteiligt. Mehr zu Memory Leaks finden Sie im gleichnamigen Abschnitt 27.3. Was können Sie aber an dieser Stelle konkret tun? Achten Sie erstens darauf, dass Sie sämtlichen nicht mehr benötigten Speicher direkt wieder freigeben, und initialisieren Sie zweitens jeden Zeiger gleich von Anfang an mit `NULL`.

15.10 Zweidimensionale dynamische Arrays

In Abschnitt 13.10 haben Sie gelesen, dass das Anwendungsgebiet von Zeigern auf Zeiger unter anderem das dynamische Erstellen von Matrizen ist. Ich will Sie jetzt nicht quälen und als Thema die Matrizenberechnung nehmen, sondern ich werde nur einfache Speicherreservierungen mit Zeilen und Spalten vornehmen:

```
int matrix[zeile][spalte];
```

Um also für ein zweidimensionales Array mit beliebig vielen Zeilen und Spalten Speicher zu reservieren, benötigen Sie zuerst Platz für die Zeile. Und zu jeder dieser Zeilen

wird nochmals Platz für die Spalte benötigt. Beim Freigeben des Speichers muss dies in umgekehrter Reihenfolge geschehen.

Hier folgt das vollständige Listing dazu:

```
/* 2D_dyn_array.c */
#include<stdio.h>
#include<stdlib.h>
#define BUF 255

int main(void) {
   int i, j, zeile, spalte;
   /* Matrix ist Zeiger auf int-Zeiger. */
   int ** matrix;

   printf("Wie viele Zeilen : ");
   scanf("%d", &zeile);
   printf("Wie viele Spalten: ");
   scanf("%d", &spalte);

   /* Speicher reservieren für die int-Zeiger (=zeile) */
   matrix = malloc(zeile * sizeof(int *));
   if(NULL == matrix) {
      printf("Kein virtueller RAM mehr vorhanden ... !");
      return EXIT_FAILURE;
   }
   /* jetzt noch Speicher reservieren für die einzelnen Spalten
    * der i-ten Zeile */
   for(i = 0; i < zeile; i++) {
      matrix[i] = malloc(spalte * sizeof(int));
         if(NULL == matrix[i]) {
            printf("Kein Speicher mehr fuer Zeile %d\n",i);
            return EXIT_FAILURE;
         }
   }
   /* mit beliebigen Werten initialisieren */
   for (i = 0; i < zeile; i++)
      for (j = 0; j < spalte; j++)
         matrix[i][j] = i + j;      /* matrix[zeile][spalte] */

   /* Inhalt der Matrix entsprechend ausgeben */
   for (i = 0; i < zeile; i++) {
      for (j = 0; j < spalte; j++)
         printf("%d ",matrix[i][j]);
```

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

   /* Speicherplatz wieder freigeben.
    * Wichtig! In umgekehrter Reihenfolge. */

   /* Spalten der i-ten Zeile freigeben */
   for(i = 0; i < zeile; i++)
      free(matrix[i]);
   /* Jetzt können die leeren Zeilen freigegeben werden. */
   free(matrix);
   return EXIT_SUCCESS;
}
```

Listing 15.10 »2D_dyn_array.c« zeigt Ihnen, wie Sie ein zweidimensionales dynamisches Array realisieren.

Zugegeben, das Listing hat es in sich. Einigen dürfte es etwas undurchsichtig erscheinen, wie aus `**matrix` nun `matrix[zeile][spalte]` wird. Am besten sehen Sie sich einfach mal an, was bei der folgenden Speicherreservierung geschehen ist:

```
matrix = malloc(zeile * sizeof(int));
```

Als Beispiel soll eine 4×3-Matrix erstellt werden, also vier Zeilen und drei Spalten.

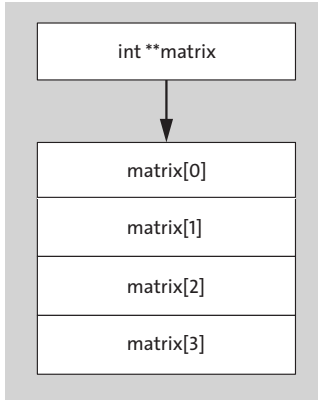

Abbildung 15.9 Reservierung des Speichers für eine Zeile (erste Dimension)

Nachdem Sie den Speicher für die einzelnen Zeilen reserviert haben wie in Abbildung 15.9, können Sie als Nächstes Speicher für die einzelnen Spalten reservieren:

```
   for(i = 0; i < zeile; i++) {
      matrix[i] = malloc(spalte * sizeof(int));
         if(NULL == matrix[i]) {
```

```
            printf("Kein Speicher mehr fuer Zeile %d\n",i);
            return EXIT_FAILURE;
        }
    }
```

Somit ergibt sich im Speicher dann das finale Bild aus Abbildung 15.10.

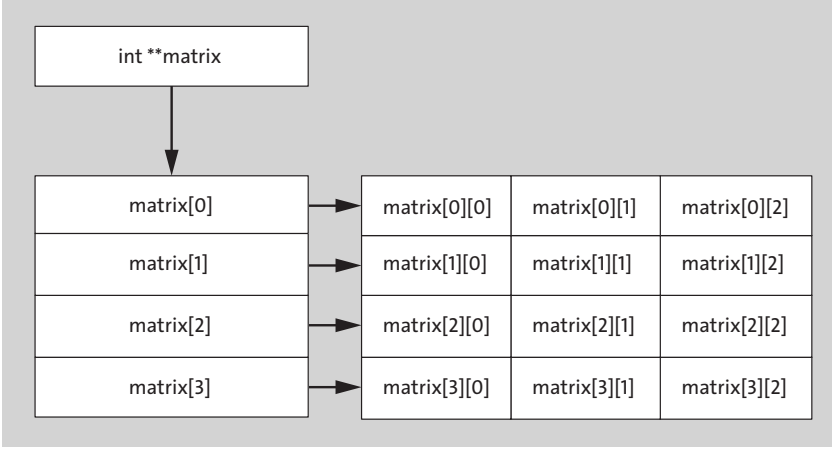

Abbildung 15.10 Nach der Reservierung des Speichers für die Spalte

Sicherlich erinnern Sie sich noch an die Demonstration des gleichwertigen Zugriffs auf ein Speicherobjekt mithilfe eines Zeigers und eines Arrays in Kapitel 13, »Zeiger (Pointer)«. Auch bei den Zeigern auf Zeiger und den zweidimensionalen Arrays gibt es einige äquivalente Fälle. Sie finden sie in Tabelle 15.2 aufgelistet.

Zugriff auf ...	Möglichkeit 1	Möglichkeit 2	Möglichkeit 3
1. Zeile, 1. Spalte	`**matrix`	`*matrix[0]`	`matrix[0][0]`
i. Zeile, 1. Spalte	`**(matrix+i)`	`*matrix[i]`	`matrix[i][0]`
1. Zeile, i. Spalte	`*(*matrix+i)`	`*(matrix[0]+i)`	`matrix[0][i]`
i. Zeile, j. Spalte	`*(*(matrix+i)+j)`	`*(matrix[i]+j)`	`matrix[i][j]`

Tabelle 15.2 Äquivalenz zwischen Zeigern auf Zeiger und mehrdimensionalen Arrays

15.11 Was muss man tun, wenn die Speicherallokation fehlschlägt?

In den vergangenen Abschnitten wurde die Speicherallokation folgendermaßen verwendet:

```
/* no_memory_1.c */
#include<stdio.h>
#include<stdlib.h>

int main(void) {
   int *ptr;
   ptr = malloc(100);

   if(NULL == ptr) {
      printf("Kein virtueller RAM mehr vorhanden ... !");
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}
```

Listing 15.11 »no_memory_1.c« behandelt den Fall, dass kein Speicher mehr frei ist.

Auf den ersten Blick scheint dieser Code in Ordnung zu sein. Es wird überprüft, ob die Funktion `malloc()` erfolgreich Speicher allozieren konnte, und wenn dies nicht der Fall ist, dann ist der zurückgegebene Zeiger `NULL`.

Stellen Sie sich jetzt vor, Sie arbeiten an einem Textverarbeitungsprogramm und haben ein paar Seiten Text zusammengestellt, den es jetzt abzuspeichern gilt. Das Programm alloziert zusätzlich Speicherplatz für den gesamten Text, bevor dieser in eine Datei abgespeichert werden kann. Jetzt, in diesem Moment, ist der Heap-Speicher aufgebraucht und das Programm schreibt eine Fehlerausgabe auf den Bildschirm und beendet sich. Der Text wird nicht abgespeichert, weil das Programm ja gar nicht mehr bis zu diesem Punkt kommt!

Es ist also kein guter Stil, ein Programm einfach zu beenden, wenn die Speicherallokation fehlschlägt. Daher folgen jetzt einige Tipps dazu, was Sie tun können, wenn eine Speicheranforderung nicht erfüllt werden konnte.

15.11.1 Speicheranforderung reduzieren

Kann kein Speicherblock einer bestimmten Größe mehr angefordert werden, sollten Sie die Speicheranforderung ein wenig reduzieren. Vielleicht kann das System einfach keinen großen zusammenhängenden Block finden. Wie Sie die erneute Speicheranforderung reduzieren, bleibt Ihnen selbst überlassen. Eine Möglichkeit wäre es, den angeforderten Speicher durch zwei zu teilen. Ein Beispiel dazu:

```
/* reduced_memory_allocation.c */
#include<stdio.h>
#include<stdlib.h>
#define MIN_LEN 256
```

```
int main(void) {
   int *ptr;
   char jn;
   static size_t len = 8192;  /* Speicheranforderung */

   do {
      ptr = malloc(len);
      /* Speicher konnte nicht alloziert werden. */
      if(ptr == NULL) {
         len /= 2;  /* Versuchen wir es mit der Hälfte. */
         ptr = malloc(len);
         if(ptr == NULL) {
            printf("Konnte keinen Speicher allozieren. "
                   " Weiter versuchen? (j/n): ");
            scanf("%c", &jn);
            fflush(stdin);
         }
      }
      else
         /* Erfolg. Speicherreservierung - Schleifenende */
         break;
   /* so lange weiterprobieren, bis 'n' gedrückt wurde oder
    * len weniger als MIN_LEN beträgt */
   } while(jn != 'n' && len > MIN_LEN);

   if(len <= MIN_LEN)
      printf("Speicheranforderung abgebrochen!!\n");
   return EXIT_SUCCESS;
}
```

Listing 15.12 »reduced_memory_allocation.c« demonstriert eine oft verwendete Methode, um bei Speichermangel die Speicheranforderung zu reduzieren.

Gelingt die Speicheranforderung nicht, wird der angeforderte Speicher um die Hälfte reduziert. Bei einem erneuten Versuch und eventuellem Scheitern wird der angeforderte Speicher wieder um die Hälfte reduziert. Dies setzt sich so lange fort, bis `MIN_LEN` Speicherplatzanforderung unterschritten wird oder der Anwender zuvor mit dem Buchstaben `'n'` abbrechen will. Dies ist natürlich nur eine von vielen Strategien, die Sie anwenden können.

15.11.2 Speicheranforderungen aufteilen

So einfach wie im letzten Beispiel werden Sie es aber höchstwahrscheinlich nicht haben. Was ist, wenn die Länge eines Strings oder die Größe einer Struktur bereits

feststeht? Sie können nicht für einen String der Länge *n* einfach *n/2* Byte Speicherplatz anfordern. Schließlich soll ja nicht nur der halbe Text gespeichert werden. Wenn es Ihnen dabei nicht allzu sehr auf die Geschwindigkeit ankommt, könnten Sie die Funktion `realloc()` verwenden (Sie erinnern sich sicherlich daran, dass diese Funktion Speicherblöcke automatisch aufteilen kann):

```
/* more_memory.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define BUF 8192
int main(void) {
   char *buffer;
   int reserviert=0;
   int i;
   static size_t len = BUF; /* Speicheranforderung */

   buffer = malloc(sizeof("Hallo Welt"));
   strcpy(buffer, "Hallo Welt");

   while(reserviert != BUF && len != 0) {
      buffer = realloc(buffer, len);
      /* Speicher konnte nicht alloziert werden. */
      if(buffer == NULL) {
         len /= 2;  /* Versuchen wir es mit der Hälfte. */
      }
      else {
         reserviert += len;
      }
   }

   for(i = 0; i < reserviert; i++)
      buffer[i] = 'x';
   buffer[i]='\0';
   printf("\n%s\n",buffer);
   return EXIT_FAILURE;
}
```

Listing 15.13 »more_memory.c« zeigt die Reduzierung der Speicheranforderung mit »realloc()«.

Dieses Listing erweist sich als ein hartnäckiger Fall der Speicherallokation. Im String `buffer` soll zusätzlicher Speicherplatz von 8192 Byte reserviert werden. Gelingt dies nicht, teilt das Programm diesen Happen in zwei Teile auf und versucht es erneut.

Diese Aufteilung geht so weit, dass eventuell byteweise Speicherplatz reserviert wird. Damit Sie auch eine Abbruchbedingung im Programm haben, wird die Anzahl des erfolgreich reservierten Speichers mitgezählt. Die Funktion `realloc()` wird dazu verwendet, dass der neu allozierte Speicher jeweils hinten angefügt wird.

15.11.3 Einen Puffer konstanter Größe verwenden

Das Problem hat vielleicht nichts mit der dynamischen Speicherallokation zu tun, aber manches Mal ist die dynamische Speicherreservierung fehl am Platze. Überdenken Sie das Programm dahingehend, ob es nicht sinnvoller wäre, ein `char`-Array konstanter Größe zu verwenden. Dies könnte sinnvoll sein, wenn Sie z. B. einen Audiopuffer fester Größe reservieren müssen oder Bilddaten einer vorher festgelegten Auflösung schnell in den Framebuffer zurückschreiben wollen. Hier kommt es einfach nicht darauf an, ob ein Programm beim Start vorher eventuell einige Megabyte mit Nullen beschreiben muss. Begehen Sie aber auf keinen Fall den Fehler, riesige Datenmengen global zu reservieren, wie ich es z. B. bei dem folgenden »Loader« für 3DS-Max-Dateien gesehen habe:

```
float WorldData[0x10000000]; // Dies geht meistens schief!
```

15.11.4 Vor der Allokation auf eine Festplatte zwischenspeichern

Wenn möglich, sollten Sie vor zeitkritischen oder umfangreichen Speicherallokationen Daten auf die Festplatte zwischenspeichern. In den nächsten Kapiteln werden Sie noch genau sehen, wie Sie mit Dateien arbeiten. Sie könnten zum Beispiel sehr viel früher im Programm Speicher dafür allozieren. Bevor eine umfangreiche Allokation für kritische Daten erfolgt, können Sie diesen Speicher verwenden, um Daten darin zwischenzuspeichern und auf die Festplatte zu schreiben. Im Allgemeinen gilt es, nur so viele Daten im virtuellen Speicher (RAM) zu beherbergen, wie auch wirklich nötig sind. Eine weitere Strategie ist es, vor einer Speicherallokation einen bereits reservierten Speicherbereich auf die Festplatte zu schreiben (temporäre Datei) und diesen Speicherblock für die nachfolgende Allokation freizugeben. Womit wir auch gleich beim nächsten Punkt wären.

15.11.5 Nur so viel Speicher anfordern wie nötig

Um eine optimale Speicherausnutzung zu erhalten, sollten Sie mit dem Speicher so geizen wie Dagobert Duck mit seinen Talern. Denn irgendwie ist es doch so: Trotz schneller Gaming-PCs mit 16 GB RAM hat man doch immer zu wenig Speicher – genauso, wie Dagobert Duck eigentlich niemals genug Geld hat. Es ist schon verhext! Wenn also immer nur der benötigte Speicher oder möglichst kleine Speicherblöcke angefordert werden, erreichen Sie die beste Performance.

Speicher sparen können Sie schon bei der Auswahl des Datentyps während der Entwicklung des Programms. Benötigen Sie z. B. unbedingt einen `double`-Wert im Programm? Reicht ein `float` nicht aus? Bei umfangreichen Strukturen sollten Sie sich fragen, ob alle Strukturelemente wirklich erforderlich sind. Müssen Berechnungen zwischengespeichert werden? Ein Beispiel ist der Matrixzugriff von `matrix[x][y]`. Das Ergebnis müsste dabei nicht gespeichert werden. Sie können auch einen Funktionsaufruf vornehmen, der Ihnen das berechnet (`matrix(x, y)`).

»Okay«, werden Sie jetzt denken, »dann baue ich eben noch 16 GB RAM ein, ist doch gar nicht so teuer.« Aber irgendwann werden Sie vielleicht in die Verlegenheit kommen, Mikrocontroller programmieren zu müssen, die nur 16 KB RAM haben. Und dann werden Sie froh sein, dass Sie sich diesen Abschnitt zu Herzen genommen haben.

Kapitel 27
Sicheres Programmieren

In diesem Kapitel werden zwei Themen angesprochen, die vielleicht auf den ersten Blick nicht allzu interessant erscheinen: Buffer Overflows und Memory Leaks. Da diese beiden Probleme jedoch häufiger in Erscheinung treten, als man denkt, sollte sich jeder Programmierer zumindest einmal mit ihnen auseinandersetzen.

Ein Aspekt, der manchmal leider übersehen wird, ist die sichere Programmierung. Programmierer setzen manchmal Funktionen ein, von denen sie zwar wissen, dass diese nicht ganz sicher sind, aber sie wissen nicht, welche Alternativen es gibt. Dies liegt natürlich auch teilweise daran, dass das Thema Sicherheit inzwischen sehr komplex ist und viele Programmierer bei dem Begriff »Sicherheit« sofort an Kryptografie denken. Da Kryptografie in der Tat ein sehr kompliziertes Thema ist, ist es verständlich, dass mancher Programmierer dann lieber »nicht ganz sichere Funktionen« verwendet. Manchmal muss es dann auch schnell gehen (vor allem vor Abschluss eines Projekts), und in diesem Fall siegt dann die gewohnte Routine. Dies öffnet natürlich wieder Tür und Tor für Angriffe, z. B. durch die gefürchteten Buffer Overflows, die ich im weiteren Verlauf noch näher beschreibe.

Sie sehen also, dass es im Bereich Sicherheit einige Begriffsverwirrungen gibt, die ich an dieser Stelle erst auflösen möchte. Sie werden danach feststellen, dass dieses Kapitel einiges von seinem Schrecken verliert. In der IT gibt es nämlich zwei wichtige Bereiche:

Die *IT-Sicherheit* beschäftigt sich in der Tat mit der Absicherung von Anwendungen und Netzwerken durch kryptografische Verfahren. Ein Anwendungsfall der IT-Sicherheit ist z. B. der sichere Datenaustausch zwischen dem Elster-Programm des Finanzamtes und Ihrem Steuerberater. Das hierfür verwendete Verfahren ist der RSA-Algorithmus. Der zweite Bereich sind die Techniken des *sicheren Programmierens*. Dieser Bereich beschäftigt sich mit dem Schutz von Programmen vor ernsten Fehlern und unvorhergesehenem Verhalten. Durch solche Fehler können Sicherheitslücken entstehen, und genau darin besteht die Überschneidung mit dem Bereich der IT-Sicherheit. Das heißt, dass es in diesem Kapitel nicht noch einmal um Kryptografie geht, sondern um Techniken, wie Sie ernste Programmierfehler vermeiden können.

Sie sehen also, dass es sich durchaus lohnt, das Thema sichere Programmierung aufzugreifen und eventuell schon bei der Entwicklung von Programmen zu berücksichtigen. Denn mit sicherem Programmieren und einem modernen Projektmanagement, das die Qualitätssicherung schon mit in die Planung einer Anwendung integriert, werden bereits 50 % aller möglichen ernsten Fehler von vornherein vermieden. Zumindest kann man durch ein gutes Controlling (dies ist Teil des Projektmanagements) die Folgen solcher ernsten Fehler sehr abmildern. Im Folgenden werden einige sicherheitskritische Dinge besprochen, die beim Programmieren auftreten können.

27.1 Buffer Overflow (Speicherüberlauf)

Eines der bekanntesten und am häufigsten auftretenden Sicherheitsprobleme ist der *Buffer Overflow* (dt. *Speicherüberlauf*, *Pufferüberlauf*), häufig auch als Buffer Overrun bezeichnet. Geben Sie einmal in einer Internetsuchmaschine den Begriff »Buffer Overflow« ein, und Sie werden angesichts der enormen Anzahl von Ergebnissen überrascht sein. Es gibt unzählige Programme, die für einen Buffer Overflow anfällig sind und auch angegriffen werden können. Das Ziel des Angreifers ist es dabei, den Buffer Overflow auszunutzen, um in das System Schadcode einzuschleusen.

Die Aufgabe dieses Abschnitts ist es nicht, Ihnen beizubringen, wie Sie Programme hacken können, sondern zu erklären, was ein Buffer Overflow ist, wie dieser ausgelöst wird und was Sie als Programmierer beachten müssen, damit Ihr Programm nicht anfällig dafür ist. Ich habe allerdings in dieser Auflage den Assembler-Code, der hinter den C-Programmen steckt, ausgespart. Ich programmiere zwar selbst sehr gern in Assembler, habe die Beispiele aber aus der 5. Auflage herausgenommen und damit auf das Feedback regiert, dass ich Lesern, die reines C lernen wollen, die Assembler-Beispiele ersparen sollte.

27.1.1 Was verursacht Buffer Overflows?

Für den Buffer Overflow ist immer der Programmierer selbst verantwortlich. Der Overflow kann überall dort auftreten, wo Daten von der Tastatur, dem Netzwerk oder einer anderen Quelle aus in einen Speicherbereich mit statischer Größe ohne eine Längenüberprüfung geschrieben werden. In Listing 27.1 sehen Sie ein solches Negativbeispiel.

```
/* bufferoverflow_1.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
```

```
int main(void) {
   char *str = "0123456789012";
   char buf[10];

   strcpy(buf, str);
   printf("%s",buf);
   return EXIT_SUCCESS;
}
```

Listing 27.1 Das Listing verursacht einen Pufferüberlauf durch Beschreiben eines Arrays mit zu vielen Werten.

Mit der Funktion `strcpy()` wurde ein Buffer Overflow erzeugt. Es wird dabei versucht, in das `char`-Array, das Platz für zehn Zeichen reserviert hat, mehr als diese zehn Zeichen zu kopieren (siehe Abbildung 27.1).

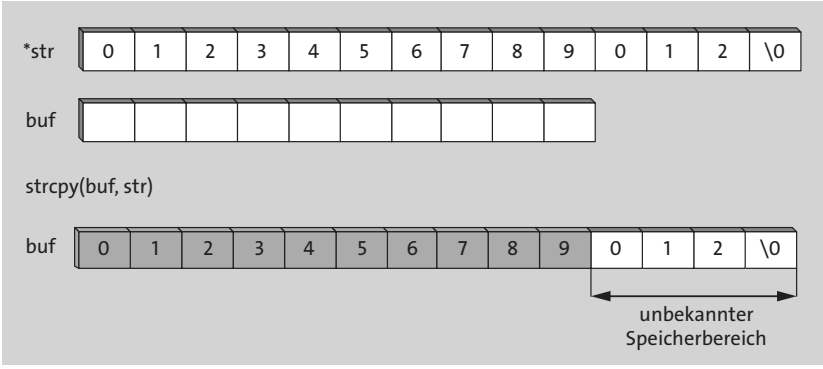

Abbildung 27.1 Pufferüberlauf mit der Funktion »strcpy()«

27.2 Warum sind Buffer Overflows kritisch für die Sicherheit?

Die Auswirkungen eines Buffer Overflows sind stark vom Betriebssystem abhängig, aber oft wirken sich diese eben direkt auf die Sicherheit aus – genau hier gibt es dann auch oft Überschneidungen mit der IT-Sicherheit. Häufig stürzt dabei das Programm ab, weil Variablen mit irgendwelchen Werten überschrieben wurden. Manches Mal bekommen Sie aber auch nach Beendigung des Programms eine Fehlermeldung zurück, etwa Speicherzugriffsfehler (Segmentation Fault). Diese Meldung wird unter Linux ausgegeben, wenn z. B. die Rücksprungadresse des Programms überschrieben wurde und das Programm irgendwo in eine unerlaubte Speicheradresse springt.

Wenn Sie aber bewusst diese Rücksprungadresse manipulieren und auf einen speziell von Ihnen erstellten Speicherbereich verweisen, der echten Code enthält, haben Sie einen sogenannten Exploit erstellt. Ein *Exploit* ist ein bewusstes Ausnutzen von

Sicherheitslücken, um eigenen Code (z. B. ein Virus) einzuschleusen. Um aber vollständig zu verstehen, wie Sicherheitslücken durch Buffer Overflows zustande kommen, müssen Sie ein wenig mehr über die Speicherverwaltung erfahren.

27.2.1 Speicherverwaltung von Programmen

Ein Programm besteht aus drei Speichersegmenten, die im Arbeitsspeicher liegen. Der Prozessor (CPU) holt sich die Daten und Anweisungen aus diesem Arbeitsspeicher, genauer gesagt aus dem Codesegment. Der Speicherbereich, den eine Anwendung nutzen kann, wird dabei noch in zusätzliche andere Segmente aufgeteilt. In Abbildung 27.2 sind die einzelnen Segmente schematisch dargestellt.

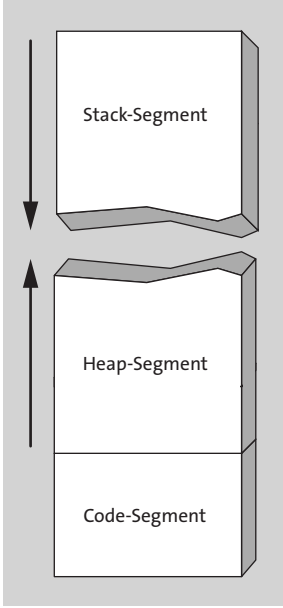

Abbildung 27.2 Die einzelnen Segmente der Speicherverwaltung

- Im Codesegment (Textsegment) befinden sich die Maschinenbefehle, die vom Prozessor eingelesen werden, oder einfacher gesagt der Programmcode selbst. Das Codesegment lässt sich nicht manipulieren, hat eine feste Größe und ist gegen Überschreiben geschützt.
- Im Heap-Segment (Datensegment) liegen die Variablen (`extern`, `static`), Felder (Arrays) und Tabellen des Programms. Maschinenbefehle, die diese Daten einlesen können, greifen auf dieses Segment zu.
- Im Stacksegment befinden sich dynamische Variablen und Rücksprungadressen von Funktionen. Dieser Bereich dient auch dem schnellen Zwischenspeichern von Daten und Parameterübergaben.

Es sei noch erwähnt, dass der Stackbereich nach unten und der Heap nach oben anwächst (es gibt aber auch Betriebssysteme, bei denen sich die Sache umgekehrt verhält). Der Stack ist auch das Angriffsziel für einen Buffer Overflow.

27.2.2 Der Stack Frame

Für jede Funktion steht ein sogenannter *Stack Frame* zur Verfügung, in dem die lokalen Variablen gespeichert werden. Wichtiger noch: Im Stack befinden sich Registerinhalte des Prozessors, die vor dem Funktionsaufruf gesichert wurden. Diese Sicherung ist nötig, um bei Beendigung der Funktion zur aufrufenden Funktion zurückspringen zu können, ohne dass diese plötzlich andere Registerinhalte sieht als vor dem Aufruf. Im nächsten Beispiel, das diesen Sachverhalt demonstriert, wird in der `main()`-Funktion die Funktion `my_func(wert1, wert2)` aufgerufen:

```
/* stackframe.c */
#include<stdio.h>
#include<stdlib.h>

void my_func(int wert1, int wert2) {
   int summe;

   summe = wert1+wert2;
   printf("Summe: %d \n",summe);
}

int main(void) {
   my_func(10,29);
   return 0;
}
```

Dies geschieht – ohne zu sehr ins Detail zu gehen – in folgenden Schritten auf dem Stack:

- Mit einem bestimmten Assembler-Befehl, z. B. `PUSH`, werden die Parameter `wert1` und `wert2` auf den Stack geschoben.
- Mit einem bestimmten Assembler-Befehl, z. B. `CALL`, wird die aktuelle Position des Programmzeigers auf dem Stack gesichert, damit bei Beendigung der Funktion `my_func()` wieder in die `main()`-Funktion zurückgesprungen werden kann. Bei Intel-Prozessoren wird diese Adresse mithilfe des Befehlszeigers (IP), des Codesegments (CS) und des Base Pointers (BP) erzeugt. Dies ist die Rücksprungadresse, die in den folgenden Beispielen mit CS:IP und BP dargestellt wird. Bei anderen Prozessoren, z. B. bei ARM-Systemen, ist dies nicht so. Dort werden die Segmente vom Betriebssystem emuliert, weil ARM-Prozessoren den Speicher linear adressieren.

- Jetzt werden die lokalen Variablen der Funktion `my_func()` eingerichtet (genauer gesagt auf dem Stack zwischengespeichert), und die Funktion arbeitet die einzelnen Befehle ab.
- Am Schluss, wenn die Funktion beendet ist, springt sie wieder zur `main()`-Funktion zurück. Dies geschieht z. B. mit dem Assembler-Befehl `RET`, der an die vorher auf dem Stack abgelegte Adresse zurückspringt.

27.2.3 Manipulation der Rücksprungadresse

In diesem Abschnitt folgt ein Beispiel, das zeigt, wie die Rücksprungadresse manipuliert werden kann. Es ist nicht Ziel und Zweck, Ihnen eine Schritt-für-Schritt-Anleitung zur Programmierung eines Exploits an die Hand zu geben und bewusst einen Buffer Overflow zu erzeugen, sondern ich will Ihnen vor Augen führen, wie schnell und unbewusst kleine Unstimmigkeiten im Quellcode Hackern Tür und Tor öffnen können – einige Kenntnisse der Funktionsweise von Assemblern vorausgesetzt.

Zur Demonstration des folgenden Beispiels werden der Compiler GCC und der Disassembler objdump verwendet. Das Funktionieren dieses Beispiels ist nicht auf allen Systemen garantiert, da bei den verschiedenen Betriebssystemen zum Teil unterschiedlich auf den Stack zugegriffen wird.

Es sei gegeben:

```
/* bufferoverflow_2.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void overflow(void) {
   char zeichen[5];
   strcpy(zeichen, "1234567"); /*Überlauf*/
}

int main(void) {
   printf("Mein 1.Buffer Overflow\n");
   overflow();
   return EXIT_SUCCESS;
}
```

Listing 27.2 »bufferoverflow_2.c« erzeugt bewusst einen Stapelüberlauf.

Da Sie jetzt wissen, wie Sie an die Rücksprungadresse eines Programms herankommen, können Sie nun ein Programm schreiben, bei dem der Buffer Overflow, der ja durch die Funktion `strcpy()` ausgelöst wird, zum Ändern der Rücksprungadresse

genutzt wird. Es wird dabei im Fachjargon von *Buffer Overflow Exploit* gesprochen. Bei dem folgenden Beispiel soll nun die Rücksprungadresse manipuliert werden:

```
/* bufferoverflow_3.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void funktion(int temp,char *array) {
   char puffer[5];

   strcpy(puffer, array);
   printf("%s\n",puffer);
}

int main(void) {
   int wert;

   wert=0;
   funktion(7,"hallo");
   wert=1;
   printf("%d\n",wert);
}
```

Das Ziel soll es nun sein, die Funktion `funktion()` aufzurufen und die Rücksprungadresse zu `wert=1;` zu überspringen, sodass `printf()` als Wert `0` statt `1` ausgibt. Nach dem Funktionsaufruf sieht der Stack so aus wie in Abbildung 27.3.

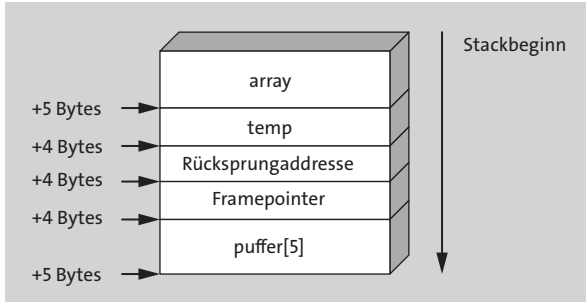

Abbildung 27.3 Der aktuelle Zustand des Stacks in diesem Beispiel

Wie kommen Sie nun am einfachsten zur Rücksprungadresse? Mit einem Zeiger. Also benötigen Sie zuerst einen Zeiger, der auf diese Rücksprungadresse verweist. Anschließend manipulieren Sie die Adresse der Rücksprungadresse, auf die der Zeiger zeigt, und zwar so, dass die Wertzuweisung `wert=1` übersprungen wird (siehe Listing 27.3).

```
/* bufferoverflow_4.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void funktion(int tmp,char *array) {
   char puffer[5];
   int *pointer;

   strcpy(puffer, array);
   printf("%s\n",puffer);
   /* Pointer auf dem Stack um 4 Byte zurücksetzen.
      Sollte jetzt auf die Rücksprungadresse zeigen. */
   pointer=&tmp-1;
   /*Rücksprungadresse, auf die Pointer zeigt, 10 Byte weiter*/
   *pointer=*pointer+10;
}

int main(void) {
   int a;

   a=0;
   funktion(7,"hallo");
   a=1;
   printf("wert = %d\n",a);
   return EXIT_SUCCESS;
}
```

Listing 27.3 »bufferoverflow_4.c« manipuliert nun direkt die Rücksprungadresse.

Die einfachste Möglichkeit, auf die Rücksprungadresse zurückzugreifen, besteht darin, die Speicheradresse der Variablen `temp` auf einen negativen Wert zu setzen und damit in der Funktion quasi rückwärts zu springen. Meistens sind nämlich Sprungadressen relative Adressen. Das bedeutet natürlich auch, dass z. B. auf einigen RISC-Maschinen, die absolute Sprungadressen verwenden, dieser Angriff nicht funktioniert. Wenn Ihr Prozessor jedoch relative Sprungadressen benutzt, können Sie sich erst einmal die Adresse der Variablen `tmp` holen und von dieser `1` abziehen:

```
pointer=&tmp-1;
```

Jetzt können Sie die Rücksprungadresse manipulieren, auf die die Variable `pointer` zeigt (siehe Abbildung 27.4):

```
*pointer=*pointer+10;
```

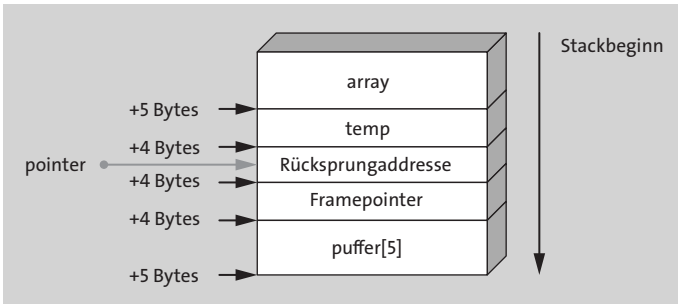

Abbildung 27.4 Der Zeiger verweist auf die Rücksprungadresse.

Ich habe die Rücksprungadresse also um 10 Byte erhöhen müssen, aber dies funktioniert wieder nur auf Intel-Maschinen unter Windows. Bei ARM-Prozessoren verhält sich die Sache anders, weil dort die OP-Codes an Adressen ausgerichtet werden müssen, die durch 4 teilbar sind. Auch auf dem Pi ist dies so, was dazu führt, dass das letzte Beispiel nicht auf dem Pi läuft. Auch die Tatsache, dass der Maschinencode in einer Variablen auf dem Stack abgelegt und die Rücksprungadresse auf die Startadresse eines fremden Programmcodes gesetzt wird, führt dazu, dass das letzte Beispiel nur unter Windows funktioniert. Exploits müssen also oft für bestimmte Betriebssysteme und Prozessoren programmiert werden.

Wenn der Exploit korrekt funktioniert, beendet sich die vorher aufgerufene Funktion durch `RET`; der Prozessor springt zu der Rücksprungadresse, die Sie vorher manipuliert haben, und Sie können nun Ihr untergeschobenes Programm ausführen. Ihnen dies detailliert zu demonstrieren, würde natürlich den Umfang des Kapitels bei Weitem sprengen und vor allem am Thema vorbeigehen. Zum anderen würde dies neben der gründlichen Kenntnis von C auch gute Kenntnisse im Assembler-Bereich (und unter Linux unter anderem auch der Shellprogrammierung) erfordern. Ich möchte Sie nur etwas sensibilisieren und Ihnen einige Gefahren aufzeigen, die bei unbedachtem Programmieren entstehen können. Zusammengefasst lassen sich Buffer Overflows für folgende Manipulationen ausnutzen:

Inhalte von Variablen, die auf dem Stack liegen, können verändert werden. Stellen Sie sich das einmal bei einer Funktion vor, die ein Passwort vom Anwender abfragt.

- Die Rücksprungadresse wird manipuliert, sodass das Programm an einer beliebigen Stelle im Speicher mit der Maschinencodeausführung fortfährt. Meistens ist dies die Ausführung des vom Angreifer präparierten Codes. Für die Ausführung von fremdem Code werden wiederum die Variablen auf dem Stack, eventuell auch auf dem Heap verwendet.
- Dasselbe Schema lässt sich auch mit Zeigern auf Funktionen anwenden. Dabei ist theoretisch nicht einmal ein Buffer Overflow erforderlich, sondern es reicht die Speicheradresse, an der sich diese Funktion befindet. Die Daten, die für die Aus-

führung von fremdem Code nötig sind, werden vorzugsweise wieder in einer Variablen gespeichert.

27.3 Wie man Buffer Overflows vermeidet

Steht Ihr Projekt in den Startlöchern, haben Sie Glück. Wenn Sie den nächsten Abschnitt durchgelesen haben, ist die Gefahr recht gering, dass Sie während der Programmerstellung eine unsichere Funktion implementieren.

Die meisten Buffer Overflows können mit den Funktionen der Standard-Bibliothek erzeugt werden. Das Hauptproblem dieser Funktionen ist, dass sie teilweise schon sehr alt sind und z. B. keine Längenüberprüfung der Ein- bzw. Ausgabe vornehmen. Daher wird empfohlen, sofern diese auf Ihrem System vorhanden sind, alternative Funktionen zu verwenden, die diese Längenüberprüfung durchführen. Falls es in Ihrem Programm auf Performance ankommt, muss jedoch erwähnt werden, dass die Funktionen mit der n-Alternative (etwa `strcpy -> strncpy`) langsamer sind als die Standardfunktionen. Auf Mikrocontrollern existieren deshalb oft nur die Standardvarianten.

Es folgt in Tabelle 27.1 bis Tabelle 27.7 ein Überblick zu einigen anfälligen Funktionen und geeigneten Gegenmaßnahmen, die getroffen werden können.

27.3.1 Unsicheres Einlesen von Eingabe-Streams

Unsichere Funktion	Gegenmaßnahme
`gets(puffer);`	`fgets(puffer, MAX_PUFFER, stdin);`
Bemerkung: Auf Linux-Systemen gibt der Compiler bereits eine Warnmeldung aus, wenn die Funktion `gets()` verwendet wird. Mit `gets()` lesen Sie von der Standardeingabe bis zum nächsten Enter einen String in einen statischen Puffer ein. Als Gegenmaßnahme wird die Funktion `fgets()` empfohlen, da diese nicht mehr als den im zweiten Argument angegebenen Wert bzw. das angegebene Zeichen einliest.	

Tabelle 27.1 Unsichere Funktion »gets()«

Unsichere Funktion	Gegenmaßnahme
`scanf("%s",str);`	`scanf("%10s",str);`
Bemerkung: Auch `scanf()` nimmt bei der Eingabe keine Längenprüfung vor. Die Gegenmaßnahme dazu ist recht simpel. Sie verwenden einfach eine Größenbegrenzung bei der Formatangabe (`%`|**`SIZE`**|`s`). Das Gleiche gilt natürlich auch für `fscanf()`.	

Tabelle 27.2 Unsichere Funktion »scanf()«

27.3.2 Unsichere Funktionen zur Stringbearbeitung

Unsichere Funktion	Gegenmaßnahme
`strcpy(buf1, buf2);`	`strncpy(buf1, buf2, SIZE);`

Bemerkung: Bei der Funktion `strcpy()` wird nicht auf die Größe des Zielpuffers geachtet, mit `strncpy()` hingegen schon. Trotzdem kann mit `strncpy()` bei falscher Verwendung ebenfalls ein Buffer Overflow ausgelöst werden:
`char buf1[100]='\0';`
`char buf2[50];`
`fgets(buf1, 100, stdin);`
`/* buf2 hat nur Platz für 50 Zeichen */`
`strncpy(buf2, buf1, sizeof(buf1));`	

Tabelle 27.3 Unsichere Funktion »strcpy()«

Unsichere Funktion	Gegenmaßnahme
`strcat(buf1 , buf2);`	`strncat(buf1, buf2, SIZE);`
Bemerkung: Bei der Funktion `strcat()` wird nicht auf die Größe des Zielpuffers geachtet, mit `strncat()` hingegen schon. Trotzdem kann mit `strncat()` bei falscher Verwendung wie schon bei `strncpy()` ein Buffer Overflow ausgelöst werden.	

Tabelle 27.4 Unsichere Funktion »strcat()«

Unsichere Funktion	Gegenmaßnahme
`sprintf(buf, "%s", temp);`	`snprintf(buf, 100, "%s", temp);`
Bemerkung: Mit `sprintf()` ist es nicht möglich, die Größe des Zielpuffers anzugeben, daher empfiehlt sich auch hier die n-Variante `snprintf()`. Das Gleiche gilt übrigens für die Funktion `vsprintf()`, wo Sie sich ebenfalls zwischen der Größenbegrenzung und `vsnprintf()` entscheiden können.	

Tabelle 27.5 Unsichere Funktion »sprintf()«

27.3.3 Unsichere Funktionen zur Bildschirmausgabe

Unsichere Funktion	Gegenmaßnahme
`printf("%s", argv[1]);`	`printf("%100s",argv[1]);`
Bemerkung: Die Länge der Ausgabe von `printf()` ist nicht unbegrenzt. Auch hier würde sich eine Größenbegrenzung gut eignen. Das Gleiche gilt für `fprintf()`.	

Tabelle 27.6 Unsichere Funktion »printf()«

27.3.4 Weitere unsichere Funktionen im Überblick

Unsichere Funktion	Bemerkung
`getenv()`	Diese Funktion lässt sich ebenfalls für einen Buffer Overflow verwenden.
`system()`	Diese Funktion sollte möglichst vermieden werden, insbesondere dann, wenn der Anwender den String selbst festlegen darf.

Tabelle 27.7 Unsichere Funktionen »getenv()« und »system()«

Abhängig vom Betriebssystem und vom Compiler gibt es noch eine Menge mehr solcher unsicheren Funktionen. Die wichtigsten habe ich aber erwähnt. Generell sollten Sie immer alle `printf()`- und `scanf()`-Funktionen mit Vorsicht und Bedacht verwenden. Häufig lässt sich beispielsweise wesentlich sicherer mit `fwrite()` oder `fread()` arbeiten. Wenigstens sollten Sie aber einen Frame (also einen sicheren Bezugsrahmen) um die unsichereren Funktionen herum bauen, die entsprechende Längenüberprüfungen durchführen. Listing 27.4 soll die Vorgehensweise bei der Erstellung eines solchen Frames verdeutlichen.

```
/* check_sprintf.c */
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAX 10

void check_bevore_sprintf(char *quelle, int max) {
   if(strlen(quelle) < MAX)
      return;
   else
      abort(); /* abort zum Debugger */
}

int main(void) {
   char *ptr1 = "123456789";
   char *ptr2 = "1234567890";
   char string[MAX];

   check_bevore_sprintf(ptr1, MAX);
   sprintf(string, "%s", ptr1);
   printf("string: %s\n", string);
```

```
   /* Hier wurde eine Unsicherheit festgestellt */
   check_bevore_sprintf(ptr2, MAX);
   sprintf(string, "%s", ptr2);
   printf("string: %s\n", string);

   return EXIT_SUCCESS;
}
```

Listing 27.4 »check_sprintf.c« sichert die Funktion »sprintf()« durch einen sicheren Bezugsrahmen ab (Funktion »check_before_sprintf()«).

Einige Programmierer gehen an dieser Stelle sogar so weit, dass sie alle `printf()`- und `scanf()`-Funktionen aus ihren fertigen Programmen verbannen, und scheuen auch nicht die Arbeit, hierzu eigene Funktionen (bzw. eigene Bibliotheken) zu schreiben, die die Benutzereingaben scannen. Leider birgt die Taktik, das Rad an einigen Stellen bewusst neu zu erfinden, wieder das Risiko, dass die eigenen, von keinem Komitee geprüften Funktionen wieder neue Sicherheitslücken enthalten.

27.4 Gegenmaßnahmen zum Buffer Overflow, wenn das Programm fertig ist

Wenn das Programm bereits fertig ist und Sie es noch nicht der Öffentlichkeit zugänglich gemacht haben, können Sie sich die Suchen-Funktion des Compilers zunutze machen oder eine eigene Funktion schreiben. Im Folgenden sehen Sie einen solchen Ansatz. Listing 27.5 gibt alle gefährlichen Funktionen, die in der Stringtabelle *danger* eingetragen sind, auf dem Bildschirm aus.

```
/* danger.c */
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAX 255

char *danger[] = {
   "scanf", "sscanf", "fscanf",
   "gets", "strcat", "strcpy",
   "printf", "fprintf", "sprintf",
   "vsprintf", "system", NULL
   /* usw. */
};

int main(int argc, char **argv) {
```

```
   FILE *fp;
   char puffer[MAX];
   int i, line=1;

   if(argc < 2) {
      printf("Anwendung: %s <datei.c>\n\n", argv[0]);
      return EXIT_FAILURE;
   }
   if ( (fp=fopen(argv[1], "r+")) == NULL) {
      printf("Konnte Datei nicht zum Lesen oeffnen\n");
      return EXIT_FAILURE;
   }
   while( (fgets(puffer, MAX, fp)) != NULL) {
      i=0;
      while(danger[i] != NULL) {
         if( (strstr(puffer,danger[i])) !=0 )
            printf("%s gefunden in Zeile %d\n",
               danger[i],line);
         i++;
      }
      line++;
   }
   fclose(fp);
   return EXIT_SUCCESS;
}
```

Listing 27.5 »danger.c« verwendet eine Stringtabelle, um alle unsicheren Funktionen, die in einem C-Quellcode auftreten, auszugeben.

Eine weitere Möglichkeit ist es, eine sogenannte *Wrapper-Funktion* zu schreiben. Eine Wrapper-Funktion können Sie sich als Strumpf vorstellen, den Sie einer anfälligen Funktion überziehen. Als Beispiel dient in Listing 27.6 die Funktion `gets()`.

```
/* wrap_gets.c */
#include<stdio.h>
#include<stdlib.h>
#define MAX  10
/* Damit es keine Kollision mit gets aus stdio.h gibt. */
#define gets(c) Gets(c)

void Gets(char *z) {
   int ch;
   int counter=0;
```

```
   while((ch=getchar()) != '\n') {
      z[counter++]=ch;
      if(counter >= MAX)
         break;
   }
   z[counter] = '\0';     /* Terminieren */
}

int main(int argc, char **argv) {
   char puffer[MAX];

   printf("Eingabe : ");
   gets(puffer);
   printf("puffer = %s\n",puffer);
   return EXIT_SUCCESS;
}
```

Listing 27.6 Das Listing zeigt, wie Sie eine Hülle (Wrapper) um eine unsichere Funktion legen können, wenn Ihr Projekt schon fast fertiggestellt ist.

Zuerst musste vor dem Compiler-Lauf die Funktion `gets()` mit

```
#define gets(c) Gets(c)
```

ausgeschaltet werden. Jetzt kann statt der echten `gets()`-Version die Wrapper-Funktion `Gets()` verwendet werden. Genauso kann dies bei den anderen gefährlichen Funktionen gemacht werden – beispielsweise mit der Funktion `strcpy()` (siehe Listing 27.7).

```
/* wrap_strcpy.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAX  10
 /* Damit es keine Kollision mit strcpy in string.h gibt */
#define strcpy Strcpy
#define DEBUG
/* #undef DEBUG */

void Strcpy(char *ziel, char *quelle) {
   int counter;
#ifdef DEBUG
   /* DEBUG-INFO */
   size_t size = strlen(quelle)+1;
```

```
   if( size > MAX )
      printf("DEBUG-INFO: Pufferueberlaufversuch\n");
   /* DEBUG-INFO Ende */
#endif

   for(counter=0; quelle[counter] != '\0' && counter < MAX-1;
     counter++)
      ziel[counter]=quelle[counter];
   /* terminieren */
   ziel[counter] = '\0';
}

int main(int argc, char **argv) {
   char puffer[MAX];

   strcpy(puffer, "0123456789012345678");
   printf("puffer = %s\n",puffer);
   return EXIT_SUCCESS;
}
```

Listing 27.7 »wrap_strcpy.c« legt einen Wrapper um die unsichere Funktion »strcpy()«.

Hier wird z. B. noch eine Debug-Info mit ausgegeben, falls dies erwünscht ist. Ansonsten muss einfach die Direktive undef auskommentiert werden.

Hinweis zu Microsoft Visual Studio

Unter Microsoft Visual Studio ab der Version 2014 sind schon einige Wrapper für unsichere Funktionen vorinstalliert. Wenn der Compiler richtig eingestellt ist, weigert sich Visual Studio sogar, Ihr Programm zu übersetzen, wenn Sie unsichere Funktionen benutzen. In diesem Fall müssen Sie an den Funktionsnamen `_s` anhängen. So wird beispielsweise aus `scanf()` `scanf_s()` und aus `printf()` `printf_s()`. Wenn Sie die Meldungen für unsichere Funktionen abschalten wollen, müssen Sie am Anfangs Ihres Programms folgende Zeile einfügen:

```
#define crt_secure_no_warnings
```

Benutzen Sie diese Zeile aber nur in Ausnahmefällen, z. B. wenn Sie alte Programme erst einmal übersetzen wollen, um eventuelle Fehler auszubügeln.

27.4.1 Programme und Tools zum Buffer Overflow

Es gibt z. B. auf dem Linux-Sektor zwei gute Bibliotheken, StackShield und StackGuard. Beide Bibliotheken arbeiten etwa nach demselben Prinzip. Beim Aufruf einer Funktion greifen diese Bibliotheken ein und sichern die Rücksprungadresse. Dafür wird

natürlich extra Code am Anfang und Ende des Funktionsaufrufes eingefügt. Wird hierbei versucht, die Rücksprungadresse zu manipulieren, schreibt das Programm eine Warnung in das Syslog (also in die Datei */etc/systemd/syslogd/system.log*) und beendet sich.

Die Voraussetzung dafür, dass Sie eine der beiden Bibliotheken verwenden können, ist, dass Sie im Besitz des Quellcodes des Programms sind, das Sie vor einem Buffer Overflow schützen wollen. Das Programm muss also mit den Bibliotheken StackShield und StackGuard neu übersetzt werden.

Einen anderen Weg geht die Bibliothek *libsafe*. Sie entfernt gefährliche Funktionsaufrufe und ersetzt sie durch sichere Versionen. Diese besitzen zusätzlich einen Schutz vor dem Überschreiben des Stack Frames.

Hinweis für den Raspberry Pi

Unter Raspbian und einigen anderen Debian-Varianten funktioniert libsafe korrekt, ist aber nicht vorinstalliert. Genauso verhält es sich mit StackShield und StackGuard. Um diese drei Bibliotheken auf dem Raspberry Pi benutzen zu können, geben Sie im Terminal folgende Kommandos ein:

```
sudo apt-get install libsafe
sudo apt-get install libstackshield
sudo apt-get install libstackguard
```

Firmen mit einem etwas größeren Geldbeutel benutzen z. B. das Programm Insure++ von Parasoft. Ich erwähne dieses Tool deshalb, weil noch einige Firmen damit arbeiten. Das Programm lässt sich auch als Testversion einige Zeit kostenlos ausprobieren. Das Programm ist für alle gängigen Systeme erhältlich und kann außer dem Buffer Overflow noch eine Menge weiterer Fehler aufdecken.

Einige davon sind:

- Speicherfehler
- Speicherlecks
- Speicherreservierungsfehler
- Verwendung uninitialisierter Variablen
- falsche Variablendefinitionen
- Zeigerfehler
- Bibliothekenfehler
- logische Fehler

Leider habe ich festgestellt, dass Insure++ nicht mehr oft aktualisiert wird, deshalb rate ich vom Einsatz ab. Denn ich denke, dass besonders bei sicherheitskritischen Programmen regelmäßige Updates Pflicht sind.

27.4.2 Ausblick

Buffer Overflows werden wohl auch in Zukunft noch vielen Programmierern Probleme bereiten und eines der häufigsten Angriffsziele von Hackern darstellen. Daher

lohnt es, sich mit diesem Thema zu befassen und immer auf dem neuesten Stand zu sein. Auch wenn moderne Betriebssysteme, wie z. B. Windows 10, einige solcher Probleme von selbst erkennen und ausgrenzen, finden clevere Programmierer immer wieder Wege, die Schutzmechanismen auszuhebeln.

Hinweis: Unterschied zwischen Hacker und Cracker

Um es richtigzustellen: Der Hacker findet Fehler in einem System heraus und meldet diese auch oft dem Hersteller des Programms. Entgegen der in den Medien verbreiteten Meinung ist ein Hacker kein »Bösewicht«. Die »Bösewichte« werden oft Cracker genannt, weil sie z. B. Lizenzschlüssel brechen und damit Raubkopien Tür und Tor öffnen. Genau dies ist für Unternehmen wie Microsoft viel schlimmer als Sicherheitslücken und deren Behebung durch regelmäßige Updates.

27.5 Stack Overflow (Stapelüberlauf)

Im letzten Punkt sind Sie ja schon mit dem Stack in Berührung gekommen. Oft überschneiden sich die Begriffe Buffer Overflow und Stack Overflow auch, nämlich, wenn einer Funktion Strukturelemente oder auch Puffer über den Stack übergeben werden müssen. Aber auch, wenn überhaupt keine Puffer im Spiel sind, die überlaufen können, kann es trotzdem bei Rekursionen zu Problemen kommen.

27.6 Was verursacht Stack Overflows?

Hier geht es also um die Gefahr rekursiver Funktionen. Beispielsweise kann eine rekursive Funktion, die sich nicht rechtzeitig beendet, versehentlich Variablen überschreiben, die in einem zum Stack benachbarten Speicherbereich liegen. Man spricht auch von einer *tiefen Rekursion* (*Deep Recursion*). Dort stehen dann nicht mehr die Variablen, die dort eigentlich stehen sollten, sondern Rücksprungadressen zu der rekursiven Funktion. Selbstverständlich kommt es hier zu Fehlern. Meistens wird dann die Anwendung beendet, aber natürlich können auch Sicherheitslücken auftreten, wenn bestimmte Speicherbereiche nicht richtig geschützt sind.

27.7 Warum ist ein Stapelüberlauf kritisch für die Sicherheit?

Sie haben es wahrscheinlich an dieser Stelle schon geahnt: Auch bei einem Stapelüberlauf kann man selbstverständlich wieder die Rücksprungadressen manipulieren. Beliebte Angriffsziele sind oft Grafikroutinen, die nicht selten rekursiv sind. Das bekannteste Beispiel ist wahrscheinlich das Ausfüllen einer Fläche mit `FloodFill()`. Wenn man nicht aufpasst, läuft nach einiger Zeit gerne mal der Stack über, und

genau hier kann ein Angreifer ansetzen und die Rücksprungadressen des übergelaufenen Stapels manipulieren. Ein weiterer bekannt gewordener Kandidat für Angriffe ist die Laderoutine für JPEG-Bilder im Firefox-Browser. Auch hier war ein Stapelüberlauf einer rekursiven Funktion schuld, der unter anderem dazu führte, dass Angreifer Viren harmlosen Fotos anhängen konnten. Auch die zuletzt entdeckten Sicherheitslücken in der Open-SSL-Bibliothek hatten ihre Ursachen unter anderem in einem nicht abgesicherten Stackbereich.

Ich will an dieser Stelle aber nicht so weit ausholen und die letzten Angriffe auf die JPEG-Bibliothek ausführlich schildern. Hierfür gibt es geeignetere Literatur. Ziel dieses Abschnitts ist wieder, Ihnen Hinweise zu geben, worauf Sie beim Programmieren achten sollten.

27.8 Wie man Stack Overflows verhindert

Wenn doch ein voll- oder leergelaufener Puffer das Problem ist, verfahren Sie, wie im letzten Abschnitt über Buffer Overflows. Schauen Sie z. B. mit libstackshield, libstackguard oder einem Tool ihrer Wahl nach Pufferüberläufen. Führen Sie Ihr Programm eventuell auch schrittweise mit einem Debugger aus, vor allem die rekursiven Funktionen. Meistens haben Sie in einer IDE wie Visual Studio auch die Möglichkeit, sich einen Trace anzeigen zu lassen. Ein Trace ist eine Auflistung aller lokalen Variablen samt Typ und Wert, inklusive der letzten 100 Stackpositionen. Hier können Sie meistens schon erkennen, ob sich auf dem Stack unerwünschte Dinge befinden, die dort nicht hingehören. Einige Angriffe lassen sich schon auf diese Weise abblocken oder zumindest erkennen.

Wenn Sie ein Projekt gerade beginnen, haben Sie die besten Chancen, die meisten ernsten Schwierigkeiten mit Stacküberläufen zu vermeiden, wenn Sie folgende grundlegende Dinge beachten:

- Wenn Sie eine Alternative zu einer Rekursion haben, die genauso schnell oder effizient ist, verwenden Sie am besten diese.
- Wenn Sie keine Alternative zu einer Rekursion haben, verwenden Sie eine sinnvolle Abbruchbedingung, falls mal etwas schiefgeht. Im Fall von `FloodFill()` kann die Rekursion z. B. stoppen, wenn bereits mehr als 1 Million Pixel gesetzt worden sind. In diesem Fall ist die Farbe garantiert aus dem Füllbereich »ausgelaufen«. Eine solche Variante habe ich auch in Kapitel 28, »Wie geht's jetzt weiter?«, verwendet.
- Gehen Sie bei größeren Projekten sämtliche Ihrer eigenen Funktionen zuerst mit einem Debugger durch, bevor Sie diese freigeben. Natürlich sollten so auch sämtliche Mitarbeiter verfahren.
- Testen Sie Ihre Funktionen am besten auch mit »unmöglichen« Eingabeparametern, die Sie so niemals machen würden. Auf diese Weise können Sie z. B. tiefe Rekursionen und problematische Abbruchbedingungen am besten aufspüren.

Im nächsten Abschnitt möchte ich noch einmal auf Speicherlecks eingehen. Auch diese Art Bugs können sehr unangenehme Folgen für die Sicherheit haben.

27.9 Memory Leaks (Speicherlecks)

Wie im Fall von Buffer Overflows oder Stack Overflows sind auch Memory Leaks in den meisten Fällen durch Programmierfehler zu erklären. Der erste Verdacht, es könnte sich bei Memory Leaks um Hardwareprobleme handeln, täuscht.

Ein Memory Leak entsteht, wenn ein Programm dynamisch Speicher alloziert (`malloc()`, `realloc()`, ...) und diese Speicherressourcen nicht mehr an das System zurückgibt (mittels `free()`). Es steht natürlich nicht unendlich viel Speicher vom Heap dafür zur Verfügung, und genau daraus können sich Probleme ergeben.

Programme wie das in Listing 27.8 folgende erzeugen keine Probleme, wenn der Speicher nicht mehr an den Heap zurückgegeben wird.

```
/* memory_leak_1.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(void) {
   char *p;

   p = malloc(sizeof("Hallo Welt\n"));
   if(NULL == p) {
      fprintf(stderr, "Abbruch: Speichermangel !!\n");
      return EXIT_FAILURE;
   }
   strcpy(p, "Hallo Welt\n");
   printf("%s",p);
   return EXIT_SUCCESS;
}
```

Listing 27.8 »memory_leak_1.c« verursacht meist keine Probleme.

Hier bekommt der Heap seinen Speicher bei Beendigung des Programms sofort wieder zurück.

Was ist aber mit Programmen, die dauerhaft im Einsatz sein müssen? Ein gutes Beispiel sind Telefongesellschaften, die jedes laufende, eingehende und ausgehende Gespräch nach dem FIFO-Prinzip (*First In, First Out*) auf dem Heap ablegen und ständig für diese Datensätze Speicher auf dem Heap allozieren bzw. für ältere Datensätze

wieder freigeben müssen. Es folgt in Listing 27.9 das Beispiel einer Variante, die keinen Schutz gegen ein Memory Leak bietet.

```
/* memory_leak_2.c */
#include<stdio.h>
#include<stdlib.h>

int main(void) {
   char *p;

   while(p = malloc(64000)) {
      if(NULL == p) {
         fprintf(stderr, "Speicherplatzmangel!!\n");
         return EXIT_FAILURE;
      }
      /* Tu was mit dem reservierten Speicher. */
   }
   return EXIT_SUCCESS;
}
```

Listing 27.9 »memory_leak_2.c« ist unsicher, weil es viel Speicher reserviert, ohne ihn irgendwann wieder freizugeben.

Dieses Programm wird wohl eine Weile ohne Probleme laufen. Doch je länger das Programm läuft, desto mehr Speicher wird vom Heap angefordert. Dies wird sich auf Dauer schlecht auf die Performance des Systems auswirken. Denn der Heap ist ja nicht nur für ein Programm allein da. Die anderen Programme, die ebenfalls Ressourcen benötigen, werden immer langsamer. Am Ende bleibt einem nichts anderes mehr übrig, als das System neu zu starten (abhängig vom Betriebssystem und der Art der Anwendung). Leider ist das Programm aber meistens längst fertiggestellt, wenn ein Speicherleck gefunden wird. Dann kann guter Rat teuer werden, wenn Sie sich nicht auskennen.

Eine primitive Möglichkeit, sofern Sie im Besitz des Quellcodes sind, ist es, sogenannte *Wrapper-Makros* für speicherallozierte und speicherfreigebende Funktionen zu schreiben, beispielsweise für die `malloc()`-Funktion:

```
#define malloc(size) \
        malloc(size);\
        printf("malloc in Zeile %ld der Datei %s (%ld Byte) \n"\
        ,__LINE__,__FILE__, size);\
        count_malloc++;
```

Diese Wrapper-Funktionen stehen in einer separaten Headerdatei. Bei Verwendung der `malloc()`-Funktion im Programm wird jetzt jeweils eine Ausgabe auf dem Bild-

schirm erzeugt, die anzeigt, in welcher Zeile und in welchem Programm die Funktion `malloc()` vorkommt und wie viel Speicher sie verwendet. Außerdem wird die Verwendung von `malloc()` mitgezählt.

Dasselbe wird anschließend auch mit der Funktion `free()` gemacht. Die Anzahl der gezählten `malloc()`- und `free()`-Aufrufe wird am Ende in eine Datei namens *DEBUG_FILE* geschrieben. Die Headerdatei, die die Wrapper-Funktionen enthält, sieht dann so aus wie in Listing 27.10.

```
/* memory_check.h */
#ifndef MEM_CHECK_H
#define MEM_CHECK_H
#define DEBUG_FILE "Debug"

static int count_malloc=0;
static int count_free  =0;
FILE *f;

#define malloc(size) \
        malloc(size);\
        printf("malloc in Zeile %d der Datei %s (%d Byte) \n"\
        ,__LINE__,__FILE__, size);\
        count_malloc++;

#define free(x)\
        free(x); \
        x=NULL;\
        printf("free in Zeile %d der Datei %s\n",\
               __LINE__,__FILE__);\
        count_free++;

#define return EXIT_SUCCESS; \
        f=fopen(DEBUG_FILE, "w");\
        fprintf(f, "Anzahl malloc : %d\n",count_malloc);\
        fprintf(f, "Anzahl free   : %d\n",count_free);\
        fclose(f);\
        printf("Datei : %s erstellt\n", DEBUG_FILE);\
        return EXIT_SUCCESS;

#endif
```

Listing 27.10 »memory_check.h« ist eine Headerdatei, die zum Überprüfen eventuell sicherheitskritischer »malloc()«-Aufrufe verwendet werden kann.

Es wurde also eine Headerdatei namens *memory_check.h* erstellt, mit der alle Aufrufe von `malloc()` und `free()` auf dem Bildschirm ausgegeben werden. Sie erfahren dadurch, in welcher Datei und in welcher Zeile sich ein Aufruf dieser Funktion befindet. Außerdem wird auch die Anzahl der `malloc()`- und `free()`-Aufrufe mitgezählt. Sind mehr `malloc()`-Aufrufe als `free()`-Aufrufe vorhanden, wurde auf jeden Fall ein Speicherleck im Programm gefunden. Hier sehen Sie ein einfaches Listing zum Testen eines Programms auf Speicherlecks:

```
/* memory_check_beispiel.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include "memory_check.h"

int main(void) {
   char *p;

   p = malloc(sizeof("Hallo Welt\n"));
   if(NULL == p) {
      fprintf(stderr, "Speichermangel!!!\n");
      return EXIT_FAILURE;
   }
   strcpy(p, "Hallo Welt\n");
   printf("%s",p);
   malloc(1024);
   free(p);
   return EXIT_SUCCESS;
}
```

In der Praxis und bei größeren Projekten ist diese Version, Memory Leaks aufzuspüren, nur bedingt geeignet. Mit dem Makro `return 0` habe ich es mir allzu leicht gemacht. Dies setzt nämlich voraus, dass ein Programm auch damit beendet wird. Oft haben Sie es aber mit dauerhaft laufenden Programmen zu tun.

Genauso sieht es mit der Zuordnung des allozierten und freigegebenen Speichers aus. Welches `malloc()` gehört zu welchem `free()`? Aber das Prinzip dürfte Ihnen klar geworden sein. Wenn Sie Fehler wie Memory Leaks finden wollen, haben Sie notfalls mit Wrapper-Makros eine gute Möglichkeit.

Meistens werden Sie schon eher auf eines der mittlerweile vielen Tools oder auf eine der Bibliotheken zurückgreifen, die zur Erkennung von Memory Leaks programmiert wurden.

Hinweis zum letzten Beispiel

Das letzte Beispiel läuft nur mit Compilern, die dem C99-Standard entsprechen, wie z. B. dem GCC-Compiler unter Linux. Unter Borland C++ beispielsweise ist das letzte Programm nicht zur Mitarbeit zu bewegen, unter Visual Studio jedoch schon.

27.10 Bibliotheken und Tools zu Memory Leaks

Es gibt mittlerweile eine unüberschaubare Menge an solchen Debugging-Tools. Daher folgt ein kleiner Überblick mit Angabe der Bezugsquellen. Meistens finden Sie auf diesen Webseiten auch gleich die Dokumentation für die Anwendung.

27.10.1 ccmalloc

Bezugsquelle: *https://github.com/librsync/librsync/blob/master/.ccmalloc*

ccmalloc wird mit dem C/C++-Programm verlinkt und gibt nach Beendigung des Programms einen Bericht über Memory Leaks aus. ccmalloc ist nicht geeignet, um festzustellen, ob versucht wurde, aus illegalen Speicherbereichen zu lesen.

27.10.2 dbmalloc

dbmalloc ist in C23 integriert. Besondere Merkmale von dbmalloc sind folgende:

- Funktionsfluss, Datei und Zeileninformationen werden mit angegeben.
- Gibt Adressen zurück (hilfreich zusammen mit Debuggern).
- Grenzbereichsüberprüfung
- Ausgabe auf die Standard-Fehlerausgabe
- Findet Memory Leaks.

Generell wird dbmalloc wie die meisten anderen Memory-Leak-Tools zu einem Programm hinzugelinkt. Sie müssen also im Besitz des Quellcodes sein, um diesen neu zu übersetzen.

27.10.3 mpatrol

Bezugsquelle: *https://sourceforge.net/projects/mpatrol/*

mpatrol ist ein leistungsfähiges Tool zum Auffinden von Memory Leaks, das sich leider auf die Performance des Programms negativ auswirkt. Folgende Funktionsmerkmale stehen Ihnen dabei zur Verfügung:

- Ein Abbild des Stacks wird bei einem Fehler angezeigt.
- Datei- und Zeileninformationen werden mit ausgegeben.

- Es ist kompatibel mit dmalloc, dbmalloc, insure und purify.
- Es ist nicht unbedingt erforderlich, neu zu übersetzen, um seine Programme mit mpatrol zu testen.
- mpatrol findet alle denkbaren Fehler auf dem Heap. Fehler auf dem Stack werden nicht gefunden.

Um ein Programm mit mpatrol zu testen, ist genau wie bei den meisten anderen Tools ein Überschreiben der speicheranfordernden und freigebenden Funktionsaufrufe notwendig. Bei mpatrol können Sie dies auf zwei Arten machen: Entweder Sie linken das Programm zu der statischen oder dynamischen Bibliothek oder Sie binden diese später durch einen Aufruf von

```
mpatrol --dynamc ./testprog -i file
```

dynamisch mit in das Programm ein. Die letzte Möglichkeit funktioniert allerdings nur, wenn das Programm schon dynamisch zur Standard-C-Bibliothek übersetzt wurde, und selbst dann nur auf einigen wenigen Systemen, die diesen Befehl unterstützen.

27.11 Tipps zu Sicherheitsproblemen

Es gibt noch eine Menge weiterer Sicherheitsprobleme, die Sie auf den ersten Blick gar nicht als solche erkennen können. Schließlich haben Sie keine Garantie, dass Ihr Programm vor allen Problemen geschützt ist, wenn Sie auf sichere Funktionen zurückgreifen.

Zum Abschluss des Kapitels folgt noch ein kleiner Leitfaden zum Thema Sicherheit. Wenn Sie diese Punkte beherzigen, sollten sich Sicherheitsprobleme auf ein Minimum reduzieren lassen:

- Vermeiden Sie Funktionen, die keine Längenprüfung der Ein- bzw. Ausgabe vornehmen (`strcpy()`, `sprintf()`, `vsprintf()`, `scanf()`, `gets()`, ...). Unter Visual Studio stehen mittlerweile sichere Varianten zur Verfügung, bei denen an den Funktionsnamen einfach `_s` angehängt wird (`strcpy_s()`, `printf_s()`, `sprintf_s()`, `scanf_s()`, ...). Visual Studio warnt auch mittlerweile, wenn Sie eine unsichere Funktion verwenden, bzw. verweigert die Kompilierung.
- Verwenden Sie bei Funktionen, die eine formatierte Eingabe erwarten (etwa `scanf()`), eine Größenangabe (etwa `%10s`, `%4d`).
- Ersetzen Sie gegebenenfalls unsichere Funktionen durch selbst geschriebene (z. B. `gets()` durch `mygets()`, ...). Dies hat natürlich den Nachteil, dass Sie das Rad an einigen Stellen neu erfinden müssen, was wiederum Sicherheitslücken verursachen kann.

- Überprüfen Sie Eingaben von der Tastatur auf eine zulässige Größe. Verlassen Sie sich nicht darauf, dass der Anwender schon das Richtige eingeben wird.
- Verwenden Sie die `exec`-Funktionen und `system()` nur mit konstanten Strings. Lassen Sie niemals den Anwender selbst einen String zusammenbasteln.
- Vergessen Sie bei der Funktion `strlen()` nicht, dass diese Funktion alle Zeichen ohne das Terminierungszeichen zählt. Beim Reservieren von Speicher müssen Sie dies berücksichtigen.
- Und der Klassiker: Die Anzahl der Elemente in einem Array und die Adressierung über den Index beginnt bei 0.

An dieser Stelle haben Sie es fast geschafft und »C von A bis Z« komplett durchgelesen. Es folgt nur noch ein kleines Kapitel mit Tipps, wie Sie jetzt weitermachen können, um ein echter Programmierprofi zu werden. Natürlich werde ich Ihnen im letzten Kapitel auch Literaturhinweise und Tipps geben, wie Sie sich weiter spezialisieren können, denn im Endeffekt müssen Sie dies tun, wenn Sie im Bereich Programmierung weitermachen wollen. Niemand weiß und kann alles, und die einzig vernünftige Antwort auf diese Tatsache ist Spezialisierung. Dies ist auch vor allem in der Berufswelt so, denn letztendlich sind heutzutage alle Spezialisten in irgendeinem Teilgebiet der IT. Natürlich kann ich Ihnen mit den Bereichen Grafikprogrammierung, Linux-Programmierung und GUI-Entwicklung nur einige Vorschläge machen, und Sie müssen später selbst entscheiden, worauf Sie sich spezialisieren wollen. Ich kann Ihnen aber trotzdem Mut machen, denn es gibt inzwischen so viele Bereiche in der IT, dass wahrscheinlich für jeden Leser etwas dabei ist.