

## 14 Dynamische Speicherverwaltung

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. Es kann 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.

Bei der dynamischen Speicherverwaltung gibt es diese Probleme nicht mehr. Zur dynamischen Speicherverwaltung wird ein Zeiger mithilfe der Funktion `malloc()` verwendet. Mit dieser Funktion geben Sie an, wie viel Speicherplatz reserviert werden soll. Der Zeiger verweist 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 dieser 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 zu Werke gehen, haben Sie schnell eine Zugriffsverletzung mit einem Zeiger verursacht. Ebenfalls zu Problemen kann es 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.

### 14.1 Das Speicherkonzept

Bevor gezeigt wird, wie Speicher dynamisch reserviert werden kann, folgt ein Exkurs über das Speicherkonzept von laufenden Programmen. Ein Programm besteht aus den vier Speicherbereichen, die in Tabelle 14.1 aufgeführt sind.

**Tabelle 14.1** Verschiedene Speicherbereiche in einem Programm

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

#### Code-Speicher

Der Code-Speicher wird in den Arbeitsspeicher geladen, und von dort aus werden die Maschinenbefehle der Reihe nach in den Prozessor (genauer gesagt in die Prozessor-Register) geschoben und ausgeführt.

## Daten-Speicher

Im Daten-Speicher befinden sich alle statischen Daten, die bis zum Programmende verfügbar sind (globale und statische Variablen).

## Stack-Speicher

Im Stack-Speicher werden die Funktionsaufrufe mit ihren lokalen Variablen verwaltet. In Abschnitt 9.20.1, »Exkurs: Stack«, bin ich schon näher auf den Stack eingegangen.

## Heap-Speicher

Dem Heap-Speicher gebührt in diesem Kapitel das Hauptinteresse. Über ihn wird die dynamische Speicherreservierung mit Funktionen wie `malloc()` erst realisiert. Der Heap funktioniert ähnlich wie der Stack. Bei einer Speicheranforderung erhöht sich der Heap-Speicher, und bei einer Freigabe wird er wieder verringert. Wenn ein Speicher 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.

## 14.2 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>
```

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

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

```
/* malloc1.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p;

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

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

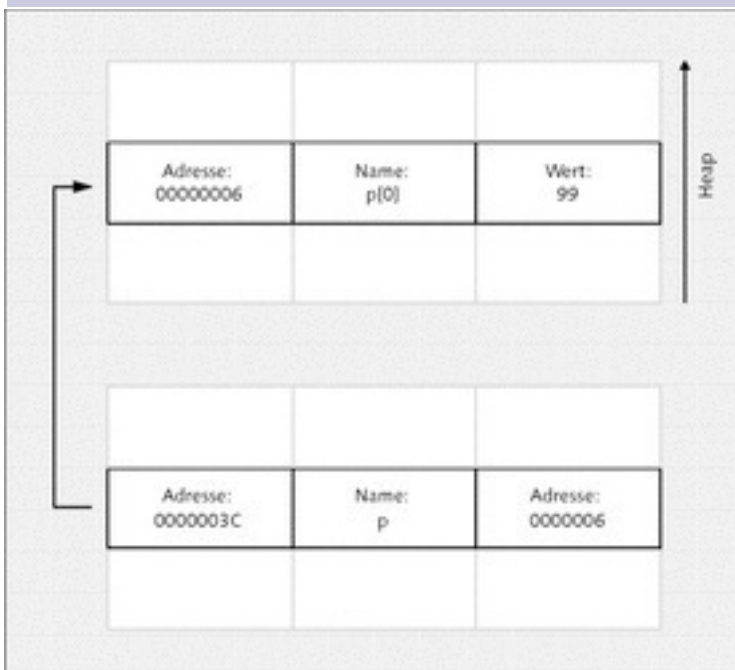
```
p = malloc(sizeof(int));
```

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. Abbildung 14.1 verdeutlicht den Programmablauf anhand einer Grafik.

### Hinweis

Ein Type-Casting 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++ (hier beispielsweise die Version 2008) z. B. brauchen Sie nur die Eigenschaftsseite mit **Alt** + **F7** aufrufen und über Konfigurationseigenschaften • C/C++ • Erweitert die Option Kompilierungsart auf Als C-Code kompilieren einstellen. Bei anderen Compilern ist dies häufig einfacher, weil man gleich ein reines C-Projekt erstellen kann.



**Abbildung 14.1** Dynamisch reservierter Speicher vom Heap

Na gut, ich denke, das beeindruckt Sie nicht besonders. Zur Laufzeit eines Programms Speicherplatz für einen `int`-Wert reservieren mit solch einem Aufwand? 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. Hier sehen Sie das Beispiel als Listing:

```
/* malloc2.c */
#include <stdio.h>
#include <stdlib.h>

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

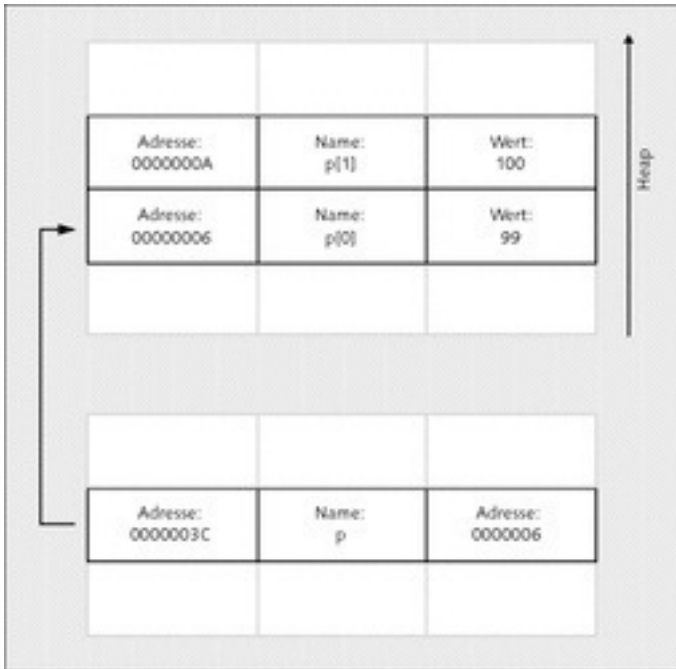
    if(p != NULL) {
        *p=99;          /* alternativ auch p[0] = 99 */
    }
}
```

```

    *(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 14.2 soll den Sachverhalt veranschaulichen.



**Abbildung 14.2** Speicher für mehrere Elemente dynamisch anfordern

Der Sachverhalt, warum `*p` und `p[0]` oder `*(p+1)` und `p[1]` auf dasselbe Element zugreifen, wurde in Kapitel 12, »Zeiger (Pointer)«, geklärt. Blättern Sie notfalls einfach zu den Tabellen am Ende von Kapitel 12 zurück.

## 14.3 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.

### 14.3.1 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 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 jetzt `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 `defines` 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_ptr1.c */
#include <stdio.h>
#include <stdlib.h>

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

    if(ptr1 != NULL) {
        /* ... */
    }
    if(ptr2 != 0) {
        /* ... */
    }
    return EXIT_SUCCESS;
}
```

Bei einem Funktionsargument allerdings ist ein solcher Zeiger-Kontext nicht unbedingt feststellbar. Hier kann ein Compiler eventuell nicht feststellen, ob der Programmierer hier 0 (als Zahl) oder eben den `NULL`-Zeiger meint. Hier ein Beispiel, das zeigt, worauf ich hinaus will:

```
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 hierbei allerdings nicht feststellen, ob der Programmierer hier 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 Argumentenliste (wie sie hier mit `exec1()` 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« Funktionsprototypen 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.

### 14.3.2 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 verzichten kann (wozu soll man eine `0` hinter `NULL` verstecken) – andere wiederum entgegenen dem, 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_ptr2.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *ptr = NULL;
    int i = NULL;      // falsch

    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.

Irgendwie bringen dann die Anfänger noch das Terminierungszeichen `\0` (`NULL`) mit in diese Geschichte ein (ich begreife zwar nicht wie, aber es geschieht ...). Vielleicht wird dabei 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).

### 14.3.3 Zusammengefasst

Aufgrund der häufigen Nachfragen zu `NULL` habe ich mich bewusst entschieden, dass Thema etwas breiter aufzurollen. Vielleicht ist der eine oder andere jetzt total 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 Funktionsaufrufes, wenden Sie am besten die von der Funktion erwartete explizite Typumwandlung an.

## 14.4 Speicherreservierung und ihre Probleme

Bei einem Aufruf der Funktion `malloc()` muss die Größe des zu reservierenden Speichers in Bytes 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:

- als numerische Konstante:
  - `p = malloc(sizeof(2));`
  - Hiermit werden vier Bytes (!) reserviert, auf deren Anfangsadresse der Zeiger `p` verweist. Es werden nicht – wie vielleicht irrtümlicherweise angenommen – zwei Bytes 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 Bytes (`int`). Somit kann die Verwendung einer numerischen Konstante sehr verwirrend sein.
- 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));`
- Sie können auch den dereferenzierten Zeiger selbst für den `sizeof`-Operator verwenden:
  - `p = malloc(sizeof(*p));`
  - Aber Achtung: Wehe, Sie vergessen den Dereferenzierungsoperator (\*) wie im folgenden Listing:

```
/* malloc3.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 und gibt die richtigen Zahlen aus. Das wäre aber 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 acht Bytes. 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*.

## 14.5 »free()« – Speicher wieder freigeben

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

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

```
void free (void *p)
```

Der Speicher wird übrigens auch ohne einen Aufruf von `free()` freigegeben, wenn sich das Programm beendet.

### Hinweis

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 ist nicht vom ANSI/ISO-Standard gefordert.

Somit hängt dieses Verhalten also von der Implementation der Speicherverwaltung des Betriebssystems ab.

Ein Beispiel zu `free()`:

```
/* free1.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)
        free(p);
    return EXIT_SUCCESS;
}
```

Es wird hier 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. Hier sehen Sie das Beispiel:

```
/* free2.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);
if(p != NULL)
    free(p);
printf("nach free() *p = %d\n", *p);
return EXIT_SUCCESS;
}

```

## Hinweis

Da der Heap üblicherweise aus Performance-Gründen nicht wieder reduziert wird, konnten Sie wie hier im Beispiel eventuell auf den freigegebenen Speicherplatz und dessen Inhalt wieder zugreifen. Aber dieses Verhalten ist nicht »portabel« und wird auch nicht vom ANSI-C-Standard gefordert. Sofern Sie also vorhaben, so etwas absichtlich in der Praxis auszuführen (warum auch immer), ist das Verhalten undefiniert.

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

Die Speicherverwaltung merkt sich die Größe eines jeden Speicherblocks, der von Ihnen angefordert wurde – daher 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 allerdings auch keinen portablen Weg, um zu erfahren, wie groß dieser Speicherblock denn tatsächlich ist.

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:

```

/* malloc4.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;
}

```

Wobei erwähnt werden muss, 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 Buchs in Abschnitt 16.25, »Ein fortgeschrittenes Thema«, werden Sie erfahren, wie Sie viel effektiver dynamischen Speicher für Text anfordern. Denn bei diesem Beispiel hätten Sie es ja gleich beim `char`-Array `puffer` belassen können.

## 14.6 Die Freispeicherverwaltung

Ohne mich hier zu sehr in die Details der Freispeicherverwaltung zu verstricken, soll dieses Thema kurz behandelt werden. Als Programmierer 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, wäre manches Mal ein wenig Hintergrundwissen wünschenswert.

Außer dem Hauptspeicher und dem Festplattenspeicher gibt es noch weitere Möglichkeiten, Daten zu speichern und zu verarbeiten. Es wird dabei von einer Speicherhierarchie gesprochen, die sich in die folgenden sechs Schichten aufteilt:

- Prozessor-Register
- First Level Cache der CPU (On-chip-Cache, 8–64 KB)
- Second Level Cache der CPU (128–512 KB)
- Hauptspeicher (RAM, z. B. 128–512 MB)
- Sekundärspeicher (Festplatte, 10–120 GB)
- Tertiärspeicher (Magnetband, 20–160 GB)

Als C-Programmierer bedienen Sie sich allerdings vorwiegend vom Hauptspeicher. Das Betriebssystem verwendet dabei das sogenannte *Paging* zur Verwaltung des Speichers. Als *Paging* wird die Unterteilung des virtuellen Speichers in Seiten (*Pages*) und des physischen Speichers in Seitenrahmen (*Page Frames*) bezeichnet. Die Größe einer Seite beträgt bei den gängigen Betriebssystemen 512 KB oder 1024 KB. Ein virtuelles Speichersystem ist erforderlich, damit auch mehrere Programme laufen können, die nicht alle in den physischen Speicher (echter vorhandener Speicher, RAM) passen würden. Dafür stellt Ihnen das Betriebssystem eine sogenannte Adresskonvention zur Verfügung, mit der aus einer virtuellen Adresse wieder eine physische Adresse wird. Denn mit virtuellen Adressen allein könnte kein Programm laufen.

## Hinweis

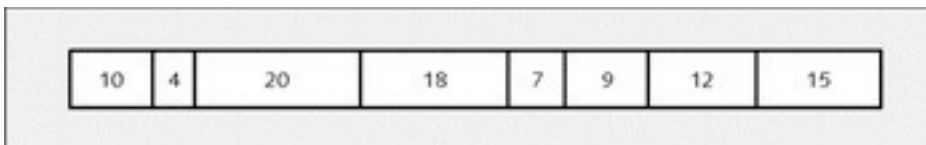
Jedem Prozess steht ein eigener virtueller Adressraum und darin eine eigene Speicherverwaltung zur Verfügung. Meistens wird hier auch noch das Swapping benutzt. Dabei wird ein Prozess in den Hauptspeicher geladen und läuft eine gewisse Zeit, bis er anschließend auf die Festplatte ausgelagert wird. Dieses Swapping findet z. B. statt, wenn nicht mehr genügend Speicherplatz vorhanden ist, um einen Prozess ausführen zu können.

In Abschnitt 14.1 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 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.

### 14.6.1 Prozessinterne Freispeicherverwaltung

Durch einen Aufruf von `malloc()` sucht das Betriebssystem jetzt 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 Beispiel dienen freie Speicherbereiche, die so im System angeordnet sind, wie Sie es in Abbildung 14.3 sehen.



**Abbildung 14.3** Freie Speicherbereiche im System

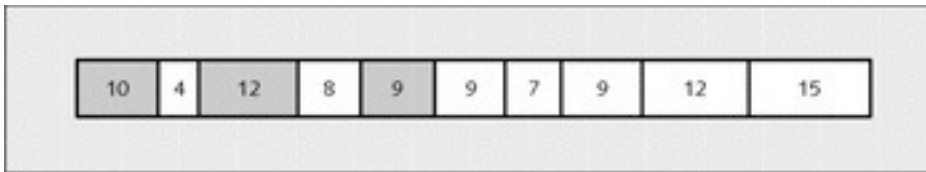
Sie fordern jetzt von diesen freien Speicherbereichen mit der Funktion `malloc()` folgende Speicherblöcke an:

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

Anhand des freien Speicherbereichs (siehe Abbildung 14.3) und den drei Speicheranforderungen will ich Ihnen die einzelnen Verfahren erklären.

#### First-Fit-Verfahren

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 14.4 gezeigt.



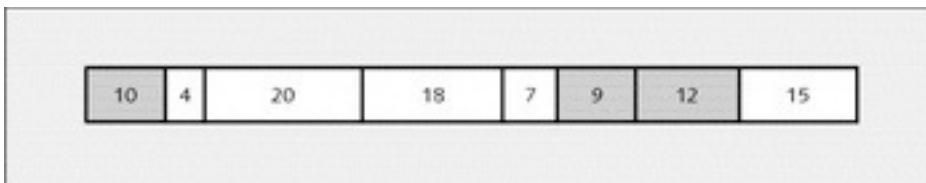
**Abbildung 14.4** First-Fit-Verfahren

### 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.

### Best-Fit-Verfahren

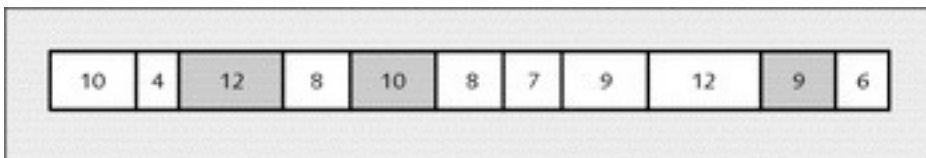
Beim Best-Fit-Verfahren wird die gesamte Speicherliste durchsucht, bis ein kleinstmögliches Loch gefunden wird. Mit diesem Verfahren wird eine optimale Speicherausnutzung garantiert.



**Abbildung 14.5** 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, und dieser wird verwendet.



**Abbildung 14.6** Worst-Fit-Verfahren

### Quick-Fit-Verfahren

Das Quick-Fit-Verfahren unterhält getrennte Listen für freie Bereiche gebräuchlicher Größe.

### 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 diesem Block entspricht, wird ein Block mit der nächsten Zweierpotenz verwendet. Die Blöcke werden außerdem dahingehend markiert, ob sie zur Anwendung frei sind. Ist bei Speicheranforderung kein gewünschter Block frei, wird ein Block in zwei gleich große Blöcke aufgeteilt.

## Hinweis

Solche Strategien der Freispeicherverwaltung haben natürlich auch ihren Sinn. 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.

Ein zweiter Grund für die 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 nicht an das Betriebssystem zurückgegeben, sondern in die Freispeicherliste eingehängt.

Nach diesem Ausflug, der schon mehr in Richtung Programmierung von Betriebssystemen ging, nun kehren wir wieder zur Praxis zurück.

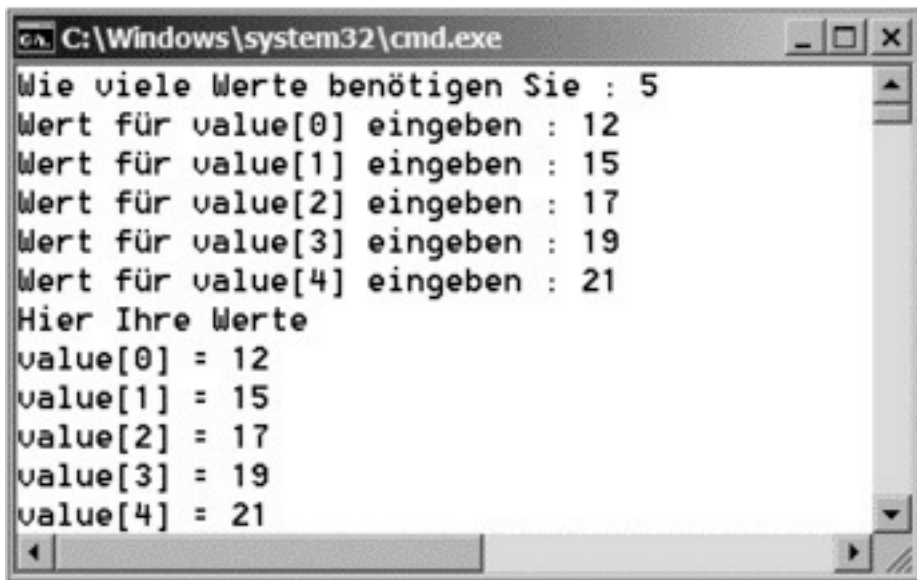
## 14.7 Dynamische Arrays

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 (lückenlos) zur Verfügung gestellt wird. In dem folgenden Beispiel wird ein solches dynamisches Array erzeugt:

```
/* dyn_array1.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;
}
```



```
C:\Windows\system32\cmd.exe
Wie viele Werte benötigen Sie : 5
Wert für value[0] eingeben : 12
Wert für value[1] eingeben : 15
Wert für value[2] eingeben : 17
Wert für value[3] eingeben : 19
Wert für value[4] eingeben : 21
Hier Ihre Werte
value[0] = 12
value[1] = 15
value[2] = 17
value[3] = 19
value[4] = 21
```

**Abbildung 14.7** 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_array2.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;
}
```

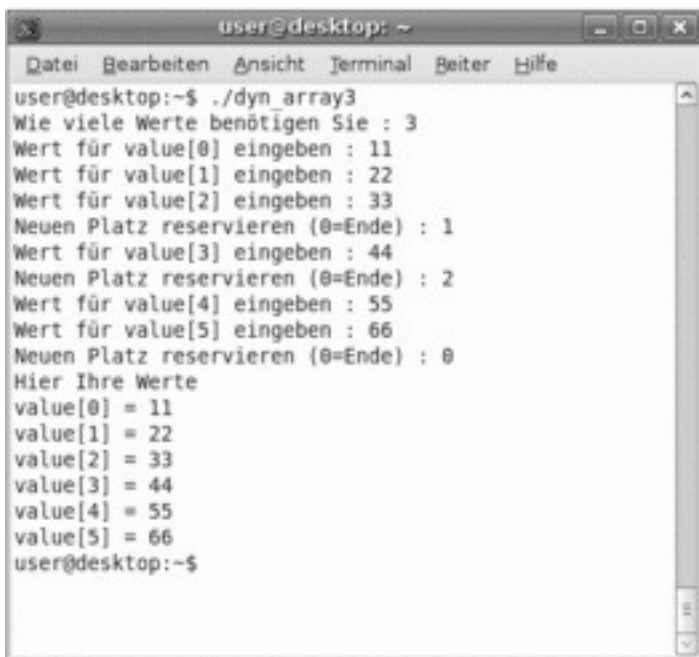
Da `*value`, `value[0]` und `*(value+1)`, `value[1]` immer auf dieselbe Speicheradresse verweisen, ist es egal, wie darauf zugegriffen wird.

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. Hier sehen Sie das Beispiel:

```
/* dyn_array3.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 = (int *)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;
}
```



```
user@desktop: ~  
Datei Bearbeiten Ansicht Terminal Beiter Hilfe  
user@desktop:~$ ./dyn_array3  
Wie viele Werte benötigen Sie : 3  
Wert für value[0] eingeben : 11  
Wert für value[1] eingeben : 22  
Wert für value[2] eingeben : 33  
Neuen Platz reservieren (0=Ende) : 1  
Wert für value[3] eingeben : 44  
Neuen Platz reservieren (0=Ende) : 2  
Wert für value[4] eingeben : 55  
Wert für value[5] eingeben : 66  
Neuen Platz reservieren (0=Ende) : 0  
Hier Ihre Werte  
value[0] = 11  
value[1] = 22  
value[2] = 33  
value[3] = 44  
value[4] = 55  
value[5] = 66  
user@desktop:~$
```

**Abbildung 14.8** 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. Ihnen das jetzt anhand eines `char`-Arrays (Strings) zu demonstrieren, erspare ich mir zunächst.

## 14.8 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 Code-Konstrukt mit `malloc()`:

```
ptr = malloc(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` Bytes 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 seinen Wert, und der neue Teil wird einfach hinten angehängt. Dieser angehängte Wert ist aber wie bei `malloc()` undefiniert. Hier sehen Sie ein kleines Beispiel dafür, wie ein Array mit der Funktion `realloc()` dynamisch erstellt wird:

```
/* realloc1.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");
    /* Endlosschleife */
    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 Bytes)\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;
}

```

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 auf `char`-Arrays umschreiben. Das folgende Listing demonstriert die dynamische Erweiterung eines Strings:

```

/* dyn_string1.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>");
    /* Endlosschleife */
    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;
}
```

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 Stringende-Zeichen berücksichtigt wird (+1). Ansonsten ist der Vorgang recht simpel: String einlesen, Zeichen zählen, erneut Speicher reservieren und hinten anhängen.

## 14.9 Speicher vom Stack anfordern mit »`alloca()`« (nicht ANSI C)

Die Funktion `alloca()` ist nicht vom ANSI-C-Standard vorgeschrieben. Da diese Funktion bei vielen Compilern vorhanden ist, soll sie hier kurz erwähnt werden. Die Syntax zu `alloca()` lautet:

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

Bei Linux/UNIX befindet sich `alloca()` in der Headerdatei `<stdlib.h>`, und unter MS-DOS/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 dieser automatisch beim Verlassen der Funktion freigegeben wird. Ganz im Gegenteil, es darf hierbei nicht einmal »`gefree()`« 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()`.

## 14.10 »`free()`« – Speicher wieder freigeben

Die Syntax der Funktion zur Freigabe von Speicher lautet:

```
#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 katastrophale Folgen haben. Die ganze Speicherverwaltung kann so aus dem Tritt gebracht werden. 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, wenn der Prozess beendet wurde, geht der Speicher wieder zurück an den Kern.

Beim Allokieren 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 sicherlich nur Anfängern? Das ist leider ganz und gar nicht so. Nicht umsonst verdienen sich viele Softwarehersteller 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. Mehr zu Memory Leaks finden Sie im gleichnamigen Abschnitt 27.2.

## 14.11 Zweidimensionale dynamische Arrays

In Abschnitt 12.9 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. */

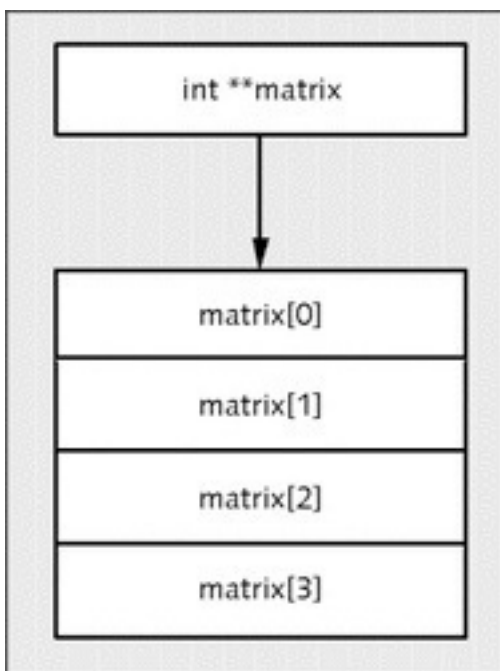
/* Spalten der i-ten Zeile freigeben */
for(i = 0; i < zeile; i++)
    free(matrix[i]);
/* Jetzt können die leeren Zeilen freigegeben werden. */
free(matrix);
return EXIT_SUCCESS;
}

```

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

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

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



**Abbildung 14.9** Reservierung des Speichers für die Zeile (erste Dimension)

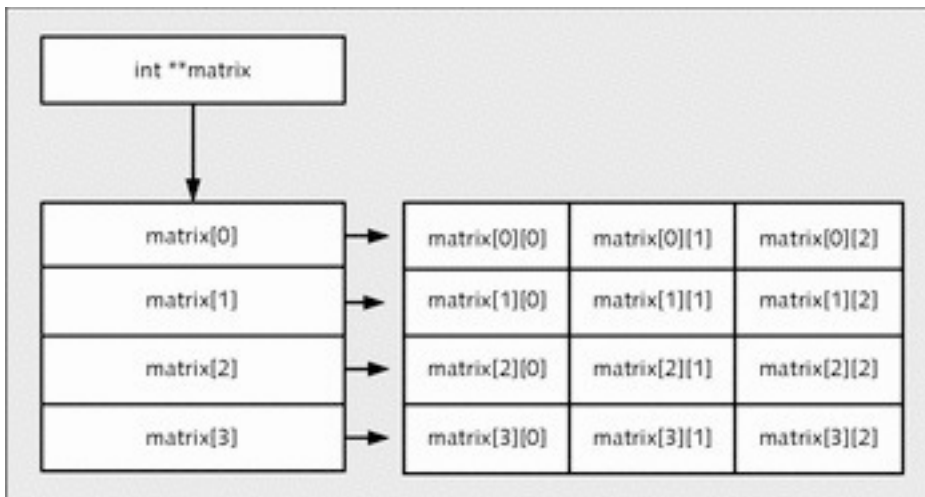
Nachdem Sie den Speicher für die einzelnen Zeilen reserviert haben, können Sie als Nächstes Speicher für die einzelnen Spalten reservieren.

```

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 14.10.



**Abbildung 14.10** 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 12, »Zeiger (Pointer)«. Auch bei den Zeigern auf Zeiger und den zweidimensionalen Arrays gibt es einige äquivalente Fälle. Sie finden sie in Tabelle 14.2 aufgelistet.

**Tabelle 14.2** Äquivalenz zwischen Zeigern auf Zeiger und mehrdimensionalen Arrays

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

## 14.12 Wenn die Speicherallokation fehlschlägt

In den vergangenen Abschnitten wurde die Speicherallokation folgendermaßen verwendet:

```
/* nomem1.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;
}
```

Auf den ersten Blick scheint dieser Code auch in Ordnung zu sein. Es wird überprüft, ob die Funktion `malloc()` erfolgreich Speicher allozieren konnte. 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 noch

Speicherplatz für den gesamten Text, bevor dieser in eine Datei abgespeichert werden kann. Jetzt, in diesem Moment, schreibt das Programm die Fehlerausgabe auf den Bildschirm und beendet sich. Der Text ist futsch und die Nerven des Anwenders auch. Es ist also kein guter Stil, ein Programm einfach zu beenden, wenn die Speicherallokation fehlschlägt. Hier folgen jetzt einige theoretische Tipps dazu, was Sie tun können, wenn eine Speicheranforderung nicht erfüllt werden konnte.

### 14.12.1 Speicheranforderung reduzieren

Kann partout kein Speicherblock einer bestimmten Größe 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:

```
/* red_mem.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;
}
```

Gelingt die Speicheranforderung hierbei 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.

### 14.12.2 Speicheranforderungen aufteilen

So einfach wie im Beispiel eben 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  $n/2$  Bytes 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:

```
/* more_mem.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; /* Speichieranforderung */

    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;
}
```

Dieses Listing erweist sich als ein hartnäckiger Fall der Speicherallokation. Im String `buffer` soll zusätzlicher Speicherplatz von 8192 Bytes 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.

### 14.12.3 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. Ein einfaches Beispiel ist das Kopieren zweier Dateien.

### 14.12.4 Zwischenspeichern auf Festplatte vor der Allokation

Wenn möglich, sollten Sie vor zeitkritischen oder umfangreichen Speicherallokationen Daten auf die Festplatte zwischenspeichern. 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.

#### 14.12.5 Nur so viel Speicher anfordern wie nötig

Um eine optimale Speicherausnutzung zu erhalten, sollten Sie mit dem Speicher geizen wie Dagobert Duck mit seinen Talern. Wenn immer nur der benötigte Speicher oder kleine Speicherblöcke angefordert werden, erreichen Sie außerdem die beste Performance. Speicher sparen können Sie schon bei der Auswahl des Datentyps bei der Entwicklung des Programms. Benötigen Sie zum Beispiel 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)`).

## 15 Strukturen

### 15.1 Struktur deklarieren

Als Beispiel dient hier ein Programm zur Verwaltung von Adressdaten mit folgenden Variablen:

```
char vname[20];
char nname[20];
long PLZ;
char ort[20];
int geburtsjahr;
```

Bei Ihrem jetzigen Kenntnisstand müsste jeder einzelne Parameter extra bearbeitet werden, sei es das Einlesen, Bearbeiten oder Ausgeben von Daten. Die Entwickler der Programmiersprache C haben zum Glück auch daran gedacht. Sie müssen einfach alle Variablen in eine Struktur verpacken. Bei den Adressdaten sieht dies dann so aus:

```
struct adres {
    char vname[20];
    char nname[20];
    long PLZ;
    char ort[20];
    int geburtsjahr;
} adressen;
```

Alle Daten wurden in einer Struktur (`struct`) namens `adres` zusammengefasst. Die Sichtbarkeit und die Lebensdauer von Strukturen entsprechen exakt der Sichtbarkeit und Lebensdauer von einfachen Variablen. Der Inhalt der Struktur `adres` wird in geschweiften Klammern zusammengefasst. Am Ende der geschweiften Klammern steht der Variablen-Bezeichner (`adressen`), mit dem auf die Struktur zugegriffen wird.

Zur Deklaration einer Struktur in C dient folgende Syntax:

```
struct typNAME {
    Datentyp1;
    Datentyp2;
```